programming pearls

by Jon Bentley

LITTLE LANGUAGES

When you say "language," most programmers think of the big ones, like FORTRAN or COBOL or Pascal. In fact, a language is any mechanism to express intent, and the input to many programs can be viewed profitably as statements in a language. This column is about those "little languages."

Programmers deal with microscopic languages every day. Consider printing a floating-point number in six characters, including a decimal point and two subsequent digits. Rather than writing a subroutine for the task, a FORTRAN programmer specifies the format `F6.2`, and a COBOL programmer defines the picture `999.99`. Each of these descriptions is a statement in a well-defined little language. While the languages are quite different, each is appropriate for its problem domain: although a FORTRAN programmer might complain that `999999.99999` is too long when `F12.5` could do the job, the COBOLer can't even express in FORTRAN such common financial patterns as `$,$$$,$$9.99`. FORTRAN is aimed at scientific computing, COBOL is designed for business.

In the good old days, real programmers would swagger to a key punch and, standing the whole time, crank out nine cards like:

```
//SUMMARY   JOB   REGION=(100K,50K)
//           EXEC  PGM=SUMMAR
//SYSIN     DD    DSNAME=REP.8601,DISP=OLD,
//                UNIT=2314,SPACE=(TRK,(1,1,1)),
//                VOLUME=SER=577632
//SYSOUT    DD    DSNAME=SUM.8601,DISP=(,KEEP),
//                UNIT=2314,SPACE=(TRK,(1,1,1)),
//                VOLUME=SER=577632
//SYSABEND DD     SYSOUT=A
```

Today's young whippersnappers do this simple job by typing

```
summarize <jan.report >jan.summary
```

Modern successors to the old "job control" languages are not only more convenient to use, they are fundamentally more powerful than their predecessors. In the June column, for instance, Doug McIlroy implemented a program to find the *K* most common words in a document in six lines of the UNIX® SHELL language.

Languages surround programmers, yet many programmers don't exploit linguistic insights. Examining programs under a linguistic light can give you a better understanding of the tools you now use, and can teach you design principles for building elegant interfaces to your future programs. This column will show how the user interfaces to half a dozen interesting programs can be viewed as little languages.

This column is built around Brian Kernighan's PIC language for making line drawings. Its compiler is implemented on the UNIX system, which is particularly supportive (and exploitative) of language processing; the sidebar on pages 714–715 shows how little languages can be implemented in a more primitive computing environment (BASIC on a personal computer).

The next section introduces PIC and the following section compares it to alternative systems. Subsequent sections discuss little languages that compile into PIC and little languages used to build PIC.

The PIC Language

If you're talking about compilers, you might want to depict their behavior with a picture:

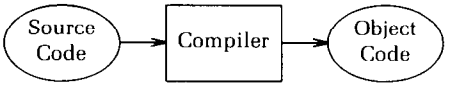
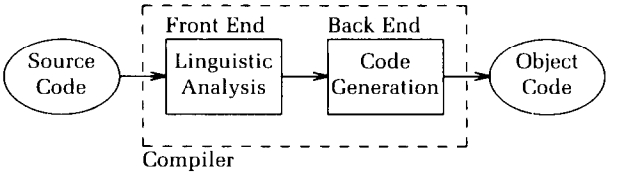


FIGURE 1. A Simple View of a Compiler

(This diagram is genuine PIC output; we'll see its input description shortly.) Or you may desire a little more detail about the internal structure:



UNIX is a trademark of AT&T Bell Laboratories.

This diagram also describes the two tasks that a program for drawing pictures must perform: a back end draws the picture while a front end interprets user commands to decide what picture to draw.

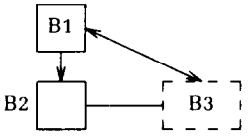
And just how does a user describe a picture? There are (broadly) three ways to do the job. An interactive program allows the user to watch the program as it is drawn, and a subroutine library adds picture primitives to the constructs in a programming language. We'll return to these approaches in the next section.

The third approach to describing pictures is the topic of this column: a little language. In Kernighan's PIC language, for instance, Figure 1 is described as

```
ellipse "Source" "Code"
arrow
box "Compiler"
arrow
ellipse "Object" "Code"
```

The first input line draws an ellipse of default size and stacks the two strings at its center. The second line draws an arrow in the default direction (moving right), and the third line draws a box with the text at its center. The implicit motion after each object makes it easy to draw the picture and convenient to add new objects to an existing picture.

Other PIC mechanisms are illustrated in this nonsense picture (which is a simple version of Figure 2):



It was drawn by

```
boxht = .25; boxwid = .25
down  # default direction
B1: box "B1"
arrow
B2: box
"B2 " at B2.w rjust
line right .4 from B2.e
B3: box dashed wid .4 "B3"
line <-> from B3.n to B1.e
```

The `boxht` and `boxwid` variables represent the default height and width of a box in inches; those values can also be explicitly set in the definition of a particular box. Text following the # character is a comment (up to the end of the line). Labels such as `B1`, `B2`, and `B3` name objects (`LongerName` is fine too); the western point of box `B2` is referred to as `B2.w`. A line of the form *string* `at` *position* places a text string at a given position; `rjust` right-justifies the string. These devices were used to draw Figure 2, which gives a yet more detailed view of a compiler.

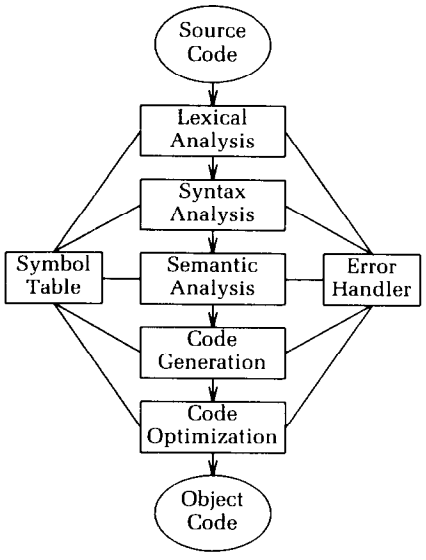
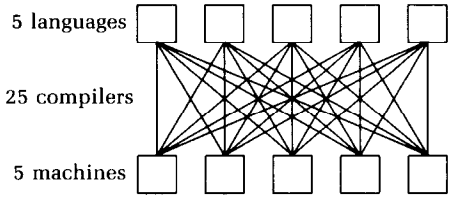


FIGURE 2. A Detailed View of a Compiler

Any particular compiler translates one source language into one object language. How can an organization run 5 different languages on 5 different machines? A brute-force approach writes 25 compilers:



An *intermediate language* circumvents much of this complexity. For each language there is a front end that translates into the intermediate language, and for each machine there is a back end that translates the intermediate language into the machine's output code.

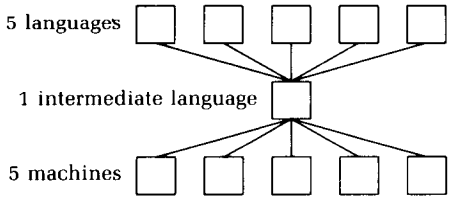


FIGURE 3. Five Languages for Five Machines

If there are L languages on M machines, the brute-force approach constructs $L \times M$ distinct compilers, while the intermediate language needs just L front ends and M back ends. (PIC compiles its output into

a picture-drawing subset of the TROFF typesetting language, which in turn produces an intermediate language suitable for interpretation on a number of output devices, from terminal display programs to laser printers to phototypesetters.)

Figure 3 uses two of PIC's programming facilities, variables and `for` loops:

```
n = 5
boxht = boxwid = .2
h = .3; w = .35
I: box at w*(n+1)/2,0
for i = 1 to n do {
   box with .s at i*w, h
   line from last box.s to I.n
   box with .n at i*w, -h
   line from last box.n to I.s
}
"1 intermediate language  " at I.w rjust
"5 languages  " at 2nd box .w rjust
"5 machines  " at 3rd box .w rjust
```

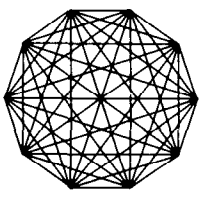
The picture of the brute-force approach is described by a single loop to draw the boxes, followed by two nested loops to make all pairwise interconnections.

The examples in this section should give you an idea of the structure of PIC, but they only hint at its complete power. I have not mentioned a number of PIC's facilities, such as built-in functions, `if` statements, macro processing, file inclusion, and a simple block structure.

Perspective

In this section we'll consider several approaches to picture-drawing programs and compare them to Kernighan's PIC language. Although the particulars are for pictures, the general lessons apply to designing user interfaces for many kinds of programs.

An interactive drawing program allows the user to enter a picture with a spatial input device (such as a mouse or a drawing pad) and displays the picture as it is drawn. Most interactive systems have a menu that includes items such as boxes, ellipses, and lines of various flavors (vertical, horizontal, dotted, etc.). Immediate feedback makes such systems quite comfortable for drawing many simple pictures, but drawing the following picture on an interactive system would require a steady hand and the patience of Job:



PIC's programming constructs allow the picture to be drawn easily:

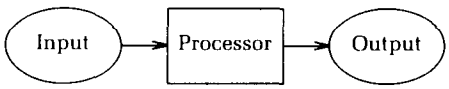
```
pi = 3.14159; n = 10; r = .5
s = 2*pi/n
for i = 1 to n-1 do {
   for j = i+1 to n do {
      line from r*cos(s*i), r*sin(s*i)\
           to   r*cos(s*j), r*sin(s*j)
   }
}
```

(The backslash character \ at the end of a line continues it on the next line.)

But handy as such features are, doesn't parsimony[1] dictate that variables and `for` loops properly belong in a full programming language? This concern is addressed by a subroutine library that adds pictures to the primitives supported by a given language. Given a subroutine `line(x1, y1, x2, y2)`, one could easily draw the last picture in Pascal:

```
pi := 3.14159; n := 10; r := 0.5;
s := 2*pi/n;
for i := 1 to n-1 do
   for j := i+1 to n do
      line (r*cos(s*i), r*sin(s*i),
            r*cos(s*j), r*sin(s*j) );
```

Unfortunately, to draw this picture



one must write, compile, and execute a program containing subroutine calls like:

```
ellipse(0.3, 0, 0.6, 0.4)
text(0.3, 0, "Input")
arrow(0.75, 0, 0.3, 0)
box(1.2, 0, 0.6, 0.4)
text(1.2, 0, "Processor")
arrow(1.65, 0, 0.3, 0)
ellipse(2.1, 0, 0.6, 0.4)
text(2.1, 0, "Output")
```

(And even such simple code may be too hard for some nonprogrammers who find PIC comfortable, such as technical typists or software managers.) The first two arguments to each routine give the x and y coordinates of the center of the object; later arguments give its width and height or a text string. These routines are rather primitive; more clever

[1] Arguments beyond taste suggest that PIC's `for` loops may be inappropriate: their syntax differs from similar loops elsewhere in the UNIX system, and PIC's `for` loops are a few orders of magnitude slower than those in other languages. Purists may write loops in other languages to generate PIC output; I am a delighted (if morally compromised) user of PIC's `for` loops—the quilts and stereograms in the exercises were easy to generate using that construct.

A Little Language for Surveys

Once a public opinion pollster knows the questions to ask in a survey, there are a number of data processing problems to be faced:

Input: Most organizations administer the survey to a respondent using a paper questionnaire; the responses are later keyed into a database. Other organizations administer questions by computers that record the responses online.

Validation: There are a number of checks for consistency and completeness, ranging from global issues (are all respondents accounted for?) to local ones (were "Democrat Only" questions administered to all and only Democrats?).

Tabulation and Output: Once the questionnaire database is complete, the responses must be tabulated and a final report prepared.

One approach to these problems is to write a new program from scratch for each task for each survey. This sidebar sketches how a single little language can solve all the problems.

Program 1 illustrates a little language I once implemented in BASIC on a personal computer. Each line that begins with a "Q" describes a question: Question 1, for instance, is stored in column 5 of each record, and asks the respondent's political party. The next three lines are the three possible responses to the questions; allowing the user to indent the responses under the question makes the file easier to read.

The single language can serve as input to several programs.

```
Q1,5 What is your political party?
  1 Democrat
  2 Republican
  3 Other
Q2,6 For whom did you vote in 1984?
  1 Reagan/Bush
  2 Mondale/Ferraro
  3 Named Other Candidate
  4 Didn't Vote
  5 Don't Know
Q3,7 Where is your polling place?
  1 Named a place
  2 Did not name a place
Q4,8 In this election, are you
  1 Very interested
  2 Somewhat interested
  3 Not interested
```

PROGRAM 1. A Description of a Survey

Input: An interactive program can administer the survey from this description and store the results in the database. If an organization uses paper questionnaires, the description is used by a "pretty-print" program to prepare the master copy and by a data-entry program to describe record formats.

Validation: From a description like Program 1, a program can ensure that all questions are answered and that all responses are in a legal range. We'll see shortly how another little language can be used to check more subtle constraints.

Tabulation and Output: The description in Program 1 provides the bulk of the input to the program that produces the final report of a survey. The user also specifies in a simple language the titles to appear on the report, which questions should be cross-tabulated, and headings for the cross-tabulations.

Just as a FORTRAN description of a program can be compiled and executed on several different kinds of computers, one description of a survey can be interpreted to perform several different tasks.

I have neglected a ton of details that complicate all survey programs. For instance, even though the questions were asked in one order, the user might want them to appear on the output in a different order (say, from greatest to least frequency of response); we'll see several other complications shortly. When I first designed the program, I sketched half a dozen bells and whistles before I realized that such was the way of folly: I could never anticipate all the options a user might desire, and any program that dealt with all options would be a rat's nest of code.

I therefore looked for a general mechanism that could handle the problems, and finally settled on a construct I called *pseudocolumns*. The "real" data was stored in columns 1 through 250 of the input record. As each record is read, the program generates pseudocolumns (defined by a little language) that start at column 251. Program 3 states that column 5 contains party information in the order Democrat, Republican, Other. To print Republicans before Democrats, one could define column 251 as follows:

```
define 251
  1 if 5 is 2   # Rep
  2 if 5 is 1   # Dem
  3 otherwise   # Other
```

(As in PIC, the # character introduces a comment.) The user can now refer to column 251 as any other column:

```
Q1,251 What is your political party?
  1 Republican
  2 Democrat
  3 Other
```

Another common task is collapsing fields. For instance, the user might wish to collapse the three age brackets 21–25, 26–30, and 31–35 into the single bracket 21–35. If column 19 contains age in 5-year clumps, one can make coarser grains in pseudocolumn 252:

```
define 252  # age, bigger lumps
  1 if 19 is 1       # below 21
  2 if 19 is 2,3,4   # 21-35
  3 if 19 is 5,6,7   # 36-50
  4 otherwise        # over 50
```

Pseudocolumns have a more sophisticated application in identifying "high-propensity" voters, who are most likely to show up at the polls:

```
define 253 # 1 if high-propensity
  1 if 6 is 1,2,3 and 7 is 1 and 8 is 1,2
  2 otherwise
```

This column is one if and only if the respondent remembered his or her 1984 candidate (column 6), could name his or her polling place (column 7), and is interested in this election (column 8). This illustrates the most complex form for a pseudocolumn; it is similar to the "conjunctive normal form" of boolean algebra.

Pseudocolumns have handled all the problems I knew about during the design phase and many others that I never would have dreamed of. Although the mechanism is quite general, it was easy to implement. The descriptions are read and stored in a data structure by 90 lines of BASIC code. The generation routine tests each value sequentially in 11 lines of BASIC (simple code was more than fast enough for this task; optimization would have been wasted).

When I first started to design a survey system to be implemented on a personal computer, I sketched an interactive program. It sounded easy at first: tell me the question, tell me the responses, now to the next question. As I explored further, though, I realized that I was designing large portions of a text editor (I want to change part of question 35. Which part? A response. Which response? 3, I think, but let me see them all. Oops, 4. Change "Smith" to "Smythe", and leave the rest alone. . .). I finally made progress by abandoning the interactive approach and thinking about the problem as designing a little language to describe surveys (and leaving the editing to the system text editor!).

routines might, for instance, have an implicit motion associated with objects.

So far I've used the term "little languages" intuitively; the time has come for a more precise definition. I'll restrict the term computer language to textual inputs (and thus ignore the spatial and temporal languages defined by cursor movements and button clicks).

> A computer language enables a textual description of an object to be processed by a computer program.

The object being described might vary widely, from a picture to a program to a tax form. Defining "little" is harder: it might imply that a first-time user can use the system in half an hour or master the language in a day, or perhaps that the first implementation took just a few days. In any case, a little language is specialized to a particular problem domain and does not include many features found in conventional languages.

PIC qualifies in my book as a little language, although admittedly a big little language. Its tutorial and user manual is 26 pages long (including over 50 sample pictures); I built my first picture in well under an hour. Kernighan had the first implementation up and stumbling within a week of putting pencil to coding form. The current version is about 4,000 lines of C code and represents several months of effort spread over five years. Although PIC has many features of big languages (variables, `for` statements, and labels), it is missing many other features (declarations, `while` and `case` statements, and facilities for separate compilation). I won't attempt a more precise definition of a little language; if the linguistic analogy gives you insight into a particular program, use it, and if it doesn't, ignore it.

We've considered three approaches to specifying pictures: interactive systems, subroutine libraries, and little languages. Which one is best? Well, that depends.

> Interactive systems are probably the easiest to use for drawing simple pictures, but a large collection of pictures may be hard to manage (given 50 pictures in a long paper, how do you make all ellipses 0.1 inches wider and 0.05 inches shorter?).
>
> If your pictures are generated by big programs, subroutine libraries can be easy and efficient. Libraries are usually uncomfortable for drawing simple pictures, though.
>
> Little languages are a natural way to describe many pictures; they can be integrated easily into document production systems to include pictures in larger documents. Pictures can be managed

using familar tools such as file systems and text editors.

I've used all three kinds of systems; each is preferable under some circumstances.[2]

PIC Preprocessors

One of the greatest advantages of little languages is that one processor's input is another processor's output. So far we've only thought of PIC as an input language. In this section we'll survey two languages for describing specialized classes of pictures; their compilers generate PIC programs as output.

We'll start with SCATTER, a PIC preprocessor that makes scatter plots from x, y data. The output of SCATTER is fed as input to PIC, which in turn feeds the TROFF document formatter.

This structure is easy to implement as a UNIX pipeline of processes:

```
scatter infile | pic | troff >outfile
```

(The UNIX SHELL program that interprets such commands is, of course, another little language. In addition to the | operator for constructing pipelines, the language includes common programming commands such as `if`, `case`, `for`, and `while`.)

PIC is a big little language, SCATTER is at the other end of the spectrum. This SCATTER input uses all five kinds of commands in the language.

```
size x 1.8
size y 1.2
range x 1870 1990
range y 35 240
label x Year
label y Population
ticks x 1880 1930 1980
ticks y 50 100 150 200
file pop.d
```

The `size` commands give the width (x) and height (y) of the frame in inches. The `range` commands tell the spread of the dimensions, and labels and ticks are similarly specified. Ranges are mandatory for both dimensions; all other specifications are optional. The description must also specify an input file containing x, y pairs. The first three lines in the file `pop.d` are

```
1880    50.19
1890    62.98
1900    76.21
```

The x-value is a year and the y-value is the United States population in millions in the census of that year. SCATTER turns that simple description of a scatter plot into a 23-line PIC program that produces Figure 4.

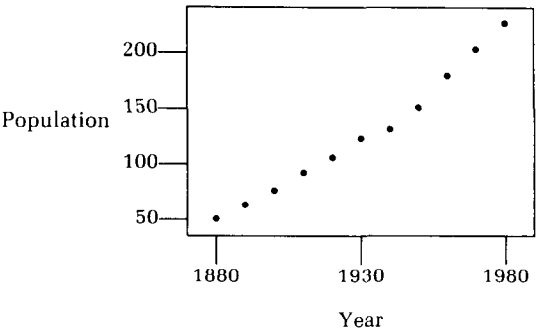


FIGURE 4. Population of the United States

SCATTER is tiny but useful. Its "compiler" is a 24-line AWK[3] program that I built in just under an hour. A companion paper in this issue of *Communications* describes GRAP, a much larger PIC preprocessor for drawing graphs, and an AWK compiler for a similar little language; see the Further Reading.

Chemists often draw chemical structure diagrams like the representation of the antibiotic penicillin G shown in Figure 5. One could in principle draw that picture in PIC, but it is more natural for a chemist to describe the structure in the CHEM language illustrated in Program 2.

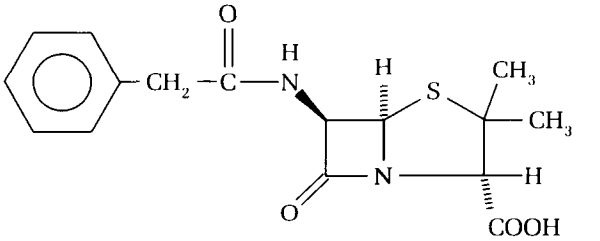


FIGURE 5. Penicillin G

[2] In terms of implementation difficulty, all three approaches have a front end for specification and a back end for picture drawing. Subroutine libraries use a language's procedure mechanism as a front end: it may be clumsy, but it's free. Little languages can use standard compiler technology for their front end; we'll see such tools shortly. Because interactive systems usually involve real-time graphics, they are typically the hardest to implement and the least portable (often with two back ends: an interactive one shows the picture as it is being drawn, and a static one writes the complete picture to a file).

[3] In many environments, SNOBOL's string-processing facilities would make it the language of choice for quickly implementing a little language. I am more comfortable with the AWK language, which was sketched in this column in June and July, 1985.

```
R1: ring4 pointing 45 put N at 2
    doublebond -135 from R1.V3 ; O
    backbond up from R1.V1 ; H
    frontbond -45 from R1.V4 ; N
    H above N
    bond left from N ; C
    doublebond up ; O
    bond length .1 left from C ; CH2
    bond length .1 left
    benzene pointing left
R2: flatring5 put S at 1 \
        put N at 4 with .V5 at R1.V1
    bond 20 from R2.V2 ; CH3
    bond 90 from R2.V2 ; CH3
    bond 90 from R2.V3 ; H
    backbond 170 from R2.V3 ; COOH
```

PROGRAM 2. CHEM Description of Penicillin G

The history of CHEM is typical of many little languages. Late one Monday afternoon, Brian Kernighan and I spent an hour with Lynn Jelinski, a Bell Labs chemist, moaning about the difficulty of writing. She described the hassles of including chemical structures in her documents: the high cost and inordinate delays of dealing with a drafting department. We suspected that her task might be appropriate for PIC, so she lent us a recent monograph rich in chemical diagrams.

That evening Kernighan and I each designed a microscopic language that could describe many of the structures, and implemented them with AWK processors (about 50 lines each). Our model of the world was way off base (the book was about polymers, so our languages were biased towards linear structures), but the output was impressive enough to convince Jelinski to spend a couple hours educating us about the real problem. By Wednesday we had built a set of PIC macros with which Jelinski could (with some pain) draw structures of genuine interest to her; that convinced her to spend even more time on the project. Over the next few days we built and threw away several little languages that compiled into those macros. A week after starting the project, the three of us had designed and implemented the rudiments of the current CHEM language, whose evolution since then has been guided by real users. The current version is about 500 lines of AWK and uses a library of about 70 lines of PIC macros.

These two brief examples hint at the power of preprocessors for little languages. PIC produces line drawings; SCATTER extends it to scatter plots, and CHEM deals with chemical structures. Each preprocessor was easy to implement by compiling into PIC; it would be much more difficult to extend interactive drawing programs to new problem domains such as graphs or chemistry.

Little Languages for Implementing PIC

In this section we'll turn from using PIC to building it. We'll study three UNIX tools that Kernighan used to construct PIC; each can be viewed as a little language for describing part of the programmer's job. This section sketches the tools; the Further Reading describes them in detail. The purpose of this section is to hint at the breadth of little languages; you may skip to the next section with impunity any time you feel overwhelmed by details.

Figure 2 illustrates the components in a typical compiler; Figure 6 shows that PIC has many, but not all, of those components. We'll first study the LEX program (which generates PIC's lexical analyzer), then turn to YACC (which performs the syntax analysis), and finally look at MAKE (which manages the 40 source, object, and header files used by PIC).

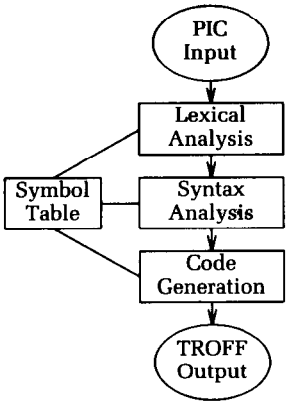


FIGURE 6. A Detailed View of PIC

A lexical analyzer (or lexer) breaks the input text into units called tokens. It is usually implemented as a subroutine; at each call it returns the next token in the input text. For instance, on the PIC input line

```
line down from B1.s
```

a lexer should return the following sequence:

```
LINE
DOWN
FROM
SYMBOL: B1
SOUTH
```

Constructing a lexer is straightforward but tedious, and therefore ideal work for a computer. Mike

Lesk's LEX language specifies a lexer by a series of pairs: when the routine sees the regular expression on the left, it performs the action on the right. Here is a fragment of the LEX description of PIC:

```
">"                 return(GT);
"<"                 return(LT);
">="                return(GE);
"<="                return(LE);
"<-"                return(HEAD1);
"->"                return(HEAD2);
"<->"               return(HEAD12);
"."(s|south)        return(SOUTH);
"."(b|bot|bottom)   return(SOUTH);
```

The regular expression (a | b) denotes either a or b. Given a description in this form, the LEX program generates a C function that performs lexical analysis.

Those regular expressions are simple: PIC's definition of a floating point number is more interesting:

```
({D}+("."?){D}*|"."{D}+)((e|E)("+"|-)?{D}+)
```

(In the spirit of this column, observe that regular expressions are a microscopic language for describing patterns in text strings.) Constructing a recognizer for that monster is tedious and error-prone work for a human; LEX does it quickly and accurately.

YACC is an acronym for "Yet Another Compiler-Compiler." Steve Johnson's program is a parser generator; it can be viewed as a little language for describing languages. Its input has roughly the same form as LEX: when a pattern (on the left-hand side) is recognized, the action on the right is performed. While LEX's patterns are regular expressions, YACC supports context-free languages. Here is part of PIC's definition of an arithmetic expression:

```
expr:
   NUMBER
 | VARNAME          { $$ = getfval($1); }
 | expr '+' expr    { $$ = $1 + $3; }
 | expr '-' expr    { $$ = $1 - $3; }
 | expr '*' expr    { $$ = $1 * $3; }
 | expr '/' expr    { if ($3 == 0.0) {
                        error("0 divide");
                        $3 = 1.0;
                      }
                      $$ = $1 / $3; }
 | '(' expr ')'     { $$ = $2; }
   ...
 ;
```

When the parser finds `expr + expr`, it returns (in `$$`) the sum of the first expression (`$1`) and the second expression (which is the third object, `$3`). The complete definition describes the precedence of operators (* binds before +), comparison operators (such as < and >), functions, and several other minor complications.

A PIC program can be viewed as a sequence of primitive geometric objects; a primitive is defined as

```
primitive:
   BOX attrlist       { boxgen($1); }
 | CIRCLE attrlist    { elgen($1); }
 | ELLIPSE attrlist   { elgen($1); }
 | ARC attrlist       { arcgen($1); }
 | LINE attrlist      { linegen($1); }
   ...
 ;
```

When the parser sees an `ellipse` statement, it parses the attribute list and then calls the routine `elgen`. It passes to that routine the first component in the phrase (the token `ELLIPSE`); `elgen` uses that token to decide whether to generate a general ellipse or a circle (a special case with length equal to width).

All PIC primitives use the same attribute list (some primitives ignore some attributes). An attribute list is either empty or an attribute list followed by an attribute:

```
attrlist:
   attrlist attr
 | /* empty */
 ;
```

And here is a small part of the definition of an attribute:

```
attr:
   DIR expr         { storefattr($1, !DEF, $2); }
 | DIR              { storefattr($1, DEF, 0.0); }
 | FROM position    { storeoattr($1, $2); }
 | TO position      { storeoattr($1, $2); }
 | AT position      { storeoattr($1, $2); }
   ...
 ;
```

As each attribute is parsed, the appropriate routine stores its value.

These tools tackle well-studied problems: the compiler book referenced under Further Reading devotes 80 pages to lexers and 120 pages to parsers. LEX and YACC package that technology: the programmer defines the lexical and syntactic structure in straightforward little languages, and the programs generate high-quality processors. Not only are the

descriptions easy to generate in the first place, they make the language very easy to modify.

Stu Feldman's MAKE program addresses a more mundane problem that is nonetheless difficult and crucial for large programs: keeping up-to-date versions of the files containing header code, source code, object code, documentation, test cases, etc. Program 3 is an abbreviated version of the file that Kernighan uses to describe the files associated with PIC.

```
OFILES = picy.o picl.o main.o print.o \
         misc.o symtab.o blockgen.o \
         ...
CFILES = main.c print.c misc.c symtab.c \
         blockgen.c boxgen.c circgen.c \
         ...
SRCFILES = picy.y picl.l pic.h $(CFILES)
pic:    $(OFILES)
        cc $(OFILES) -lm
$(OFILES): pic.h y.tab.h
memo:
        pic memo | eqn | troff -ms >memo.out
backup: $(SRCFILES) makefile pictest.a
        push safemachine $? /usr/bwk/pic
        touch backup
bundle:
        bundle $(SRCFILES) makefile README
```

PROGRAM 3. PIC's MAKE file

The file starts with the definition of three names: `OFILES` are the object files, `CFILES` contain C code, and the source files `SRCFILES` consist of the C files and the YACC description `picy.y`, the LEX description `picl.l`, and a header file. The next line states that PIC must have up-to-date versions of object files (MAKE's internal tables tell how to make object files from source files); the next line tells how to combine those into a current version of PIC. When Kernighan types `make pic`, MAKE checks the currency of all object files (`file.o` is current if its modification time is later than `file.c`), recompiles out-of-date modules, then (if needed) loads the pieces along with the appropriate libraries. The following line states that the object files depend on the two named header files.

The next two lines tell what happens when Kernighan types `make memo`: the file containing the technical memorandum is processed by TROFF and two preprocessors. The `backup` command saves on `safemachine` all modified files, and the `bundle` command wraps the named files into a package suitable for mailing. Although MAKE was originally designed with compiling in mind, Feldman's elegant general mechanism gracefully supports all these additional housekeeping functions.

Principles

Little languages are an important part of the popular Fourth- and Fifth-Generation Languages and Application Generators, but their influence on computing is much broader. Little languages often provide an elegant interface for humans to control complex programs or for modules in a large system to communicate with one another. Although most of the examples in the body of this column are large "systems programs" on the UNIX system, the sidebar on pages 714–715 shows how the ideas were used in a fairly mundane task implemented in BASIC on a microcomputer.

The principles of language design summarized below are well known among designers of big programming languages; they are just as relevant to the design of little languages.

Design Goals. Before you design a language, carefully study the problem you are trying to solve. Should you instead build a subroutine library or an interactive system? An old rule of thumb states that the first 10 percent of programming effort provides 90 percent of the functionality; can you make do with an AWK or BASIC or SNOBOL implementation that cheaply provides the first 90 percent, or do you have to use more powerful tools like LEX and YACC to get to 99.9 percent?

Simplicity. Keep your language as simple as possible. A smaller language is easier for its implementers to design, build, document, and maintain and for its users to learn and use.

Fundamental Abstractions. Typical computer languages are built around the world-view of a von Neumann computer: instructions operate on small chunks of data. The designer of a little language has to be more creative: the primitive objects might be geometric symbols, chemical structures, context-free languages, the files in a program, or the questions in a survey. Operations on objects vary just as widely, from fusing two benzene rings to recompiling a source file. Identifying these key players is old hat to programmers: the primitive objects are a program's abstract data types, and the operations are the key subroutines.[4]

Linguistic Structure. Once you know the basic ob-

[4] In the mid 1970s Bill McKeeman (now at the Wang Institute of Graduate Studies) consulted on an Automated Teller Machine project that was running out of its 28 kilobytes. Several programming tricks compromised maintainability to squeeze space, but each time additional functions consumed even more memory. After losing this battle several times, McKeeman watched a human teller perform the function. He found that the teller's job was defined by paper slips that describe various transactions (deposit, withdrawal, balance inquiry, etc.) stored in three dozen slots beneath the teller's window. McKeeman realized that the human teller could be viewed as a machine with three dozen operation codes, each defined by a separate form. He therefore designed an interpreted program with commands in a little language for banking. The new design provided three times the functionality in less memory, and maintenance was much easier.

jects and operations, there are still many ways of writing down their interactions. The infix arithmetic expression `2+3*4` might be written in postfix as `234*+` or functionally as `plus(2,times(3,4));` there is often a trade-off between naturalness of expression and ease of implementation. But whatever else you may or may not include in your language, be sure to allow indentation and comments.

Yardsticks of Language Design. Rather than preach about tasteful design, I've chosen as examples useful languages that illustrate good taste. Here are some of their desirable properties.

Orthogonality: keep unrelated features unrelated.

Generality: use an operation for many purposes.

Parsimony: delete unneeded operations.

Completeness: can the language describe all objects of interest?

Similarity: make the language as suggestive as possible.

Extensibility: make sure the language can grow.

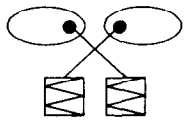
Openness: let the user "escape" to use related tools.

The Design Process. Like other great software, great little languages are grown, not built. Start with a solid, simple design, expressed in a notation like Backus-Naur form. Before implementing the language, test your design by describing a wide variety of objects in the proposed language. After the language is up and running, iterate designs to add features as dictated by real use.

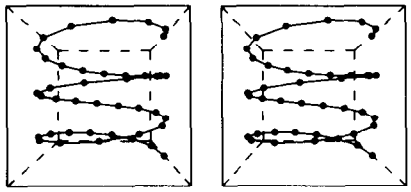
Insights from Compiler Building. When you build the processor for your little language, don't forget lessons from compilers. As much as possible, separate the linguistic analysis in the front end from the processing in the back end; that will make the processor easier to build and easier to port to a new system or new use of the language. And when you need them, use compiler-building tools like LEX, YACC, and MAKE.

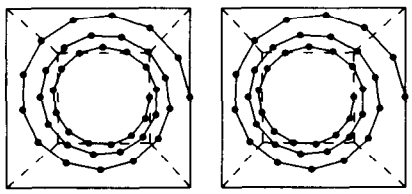
Problems

1. Most systems provide a package for sorting files; the interface is usually a little language. Evaluate the language provided by your system. Design a better language and implement it (perhaps as a preprocessor that generates commands for your system sort).
2. LEX uses a little language for regular expressions to specify lexical analyzers. What other programs on your system employ regular expressions? How do they differ, and why?
3. Study different languages for describing bibliographic references. How do the languages differ in applications such as document retrieval systems and bibliography programs in document production systems? How are little languages used to perform queries in each system?
4. Study examples of what might be the littlest languages of them all: assemblers, format descriptions, and stack languages.
5. Design and implement picture languages specialized for the following domains.
   a. Many people can perceive a three-dimensional image by crossing their eyes and fusing the two halves of stereograms:

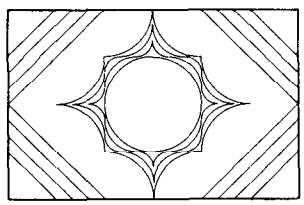


   A small survey I conducted suggests that about half the readers of *Communications* should be able to perceive these three-dimensional scenes; the other half will get a headache trying.



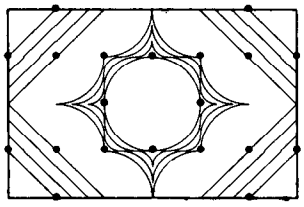


   These pictures were drawn by a 40-line PIC program.
   b. Ravi Sethi described this quilt in a 35-line PIC program.



   The quilt is a 4 × 6 array of rotations of these two squares:

The dots shown above in the corners of the two squares are also displayed in this version of the quilt:



c. Other interesting pictorial domains include the following: data structures depicted in pictures, such as arrays (see page 479 of the June *Communications*), trees, and graphs (drawing Finite State Machines is especially interesting); descriptions of musical scores (consider both rendering the score in pictures and playing it on a music generator); and pictorially scored games (such as bowling and baseball).

6. Design a little language to deal with common forms in your organization, such as expense reports for trips.
7. How can processors of little languages respond to linguistic errors? (Consider the options available to compilers for large languages.) How do particular processors respond to errors?
8. These questions deal with the survey system described in the sidebar on pages 714–715.
   a. The example assumed (falsely) that a question or a response always fits on a single line; extend the language to handle multiple-line text.
   b. Design a program to automate the administration of a survey. Describe a mechanism to ensure, for instance, that Democrat-only questions are asked only of Democrats.

Solutions to June's Problems

1. Most programs for computing the K most common words in a file spend a great deal of effort on words that occur only a few times; in the text of both May and June's columns, for instance, over half the distinct words occurred just once. A two-pass program saves time and space by reading the file twice: the first pass identifies infrequent words, and the second pass concentrates on other words. The two passes share information in an array named *Count*, which is initialized to zero. As the first pass reads word X, it increments *Count*[*Hash*(X)]; no information is stored about the words themselves. After the first pass, frequent words must have high *Counts*, but some high *Count* values could be the result of several rare words. The second pass deals with word X only if *Count*[*Hash*(X)] is appropriately large, using any of the techniques discussed in the June column.
4. Knuth assumed that most frequent words tend to occur near the front of the document; McIlroy pointed out that some frequent words may not appear until relatively late. When Knuth ran his program with reduced memory to find the 100 most common words in Section 3.5 of his *Seminumerical Algorithms*, it missed just two words that were used frequently at the end of the section.
5. For insight into this problem, see Exercise 5.24 (and the answer) in Knuth's *Sorting and Searching*.

Further Reading

You may never have heard of *Compilers: Principles, Techniques, and Tools* by Aho, Sethi and Ullman, but you'd probably recognize the cover of the "New Dragon Book" (published in 1986 by Addison-Wesley). And you *can* judge this book by its cover: it is an excellent introduction to the field of compilers, with a healthy emphasis on little languages.[5] Furthermore, the book makes extensive use of PIC to tell its story in pictures. (Most of the compiler pictures in this column were inspired by pictures in that book.)

Chapter 8 of *The UNIX Programming Environment* by Kernighan and Pike (Prentice-Hall, 1984) is the case history of a little language. It uses the UNIX tools sketched in this column to design, develop, and document a language.

The companion article by Kernighan and me on page 782 of this issue of *Communications* describes a little language in detail: the GRAP language for graphical displays of data. The references in that paper present details on PIC and several related UNIX document production tools.

[5] I first learned the importance of little languages from Mary Shaw, who edited *The Carnegie-Mellon Curriculum for Undergraduate Computer Science* (published by Springer-Verlag in 1985). Course 320 in that curriculum restructures a traditional compiler course to place substantial emphasis on little languages.

For Correspondence: Jon Bentley, AT&T Bell Laboratories, Room 2C-317, 600 Mountain Ave., Murray Hill, NJ 07974.