Design and Implementation of a Near-optimal Loglan Syntax

Submitted by Jeff Prothero, jsp@milton.u.washington.edu
May 9, 1990
editor's note: this file used to live at the late great Planned Languages FTP server, before the WWW was invented *Then at www.rickharrison.com, but the page no longer exists*

Synopsis

A near-optimal surface syntax for an artificial human language is derived, and a prototype software implementation provided.

Motivation

The Sapir-Whorf Hypothesis claims that the language you think in influences the way you think. Just as you tend to solve the same problem differently depending on whether you are coding in ForTran, C, APL or Lisp, so (the Hypothesis claims) you tend to see the world differently depending on the human language in which you think: Your language imprisons your thoughts. You cannot escape this prison -- but you can transfer to another one, by learning a new language.

The loglan (logical language) community is devoted to exploring the practical implications of the Sapir-Whorf Hypothesis by constructing and using artificial human languages. The syntax of current loglans leaves much to be desired. The root design is about thirty-five years old, predating APL, Forth, LALR(1) parsers, Unix, and much of contemporary formal language theory. This root design is nearly obscured by accumulated patches.

Current word-resolution algorithms are defined informally in English, the descriptions run over a dozen pages, and the algorithm fails on about thirty percent of word boundaries, at which point inter-word pauses are mandatory.

Current grammars contain over a hundred rules, are context-sensitive, cannot be recognized in linear time, and after a third of a century of continuous tinkering, still require fixes on a weekly basis. I've designed and implemented four parsers for various loglans: It's a pain!

Here I will propose a loglan syntax which:
* Is simple enough to be parsed by a couple of hundred lines of straightforward C. (See attached program.)
* Is simple for humans to learn and use.
* Allows for unambiguous resolution of continuous human speech.
* Offers near-optimal conciseness and simplicity.

Overview

The proposed syntax consists of:

* An alphabet. "bcdf ghjk lmnp stvz" is suggested, but the choice is not critical.
* A pronunciation scheme which makes all sequences of letters equally pronouncable, thus decoupling the rest of the language design from the details of the human vocal tract.
* A set of "affixes" (distinct strings of letters) from which to build words. (These will be used as word prefixes, roots, and suffixes. A word will be the concatenation of one or more affixes.) This affix set has the property that affixes of all lengths are available, so that frequently used affixes can be given a short representation (Zipf's Law). This affix set also has the property that any arbitrary string of letters can be decomposed into affixes in a most one way, so that arbitrary affixes may be concatenated to produce words without introducing ambiguities.
* A distinguished set of affixes which mark word boundaries.
* A simple (four-rule) grammar which allows the expression and parsing of arbitrary syntax trees composed of nodes of with zero, one or two children.

Problem definition

To make any sort of optimality argument, or indeed any sort of rational engineering decision, one needs a fairly precise characterization of the problem to be solved. We will think of a language as an encoding used by two agents to exchange information via a serial channel, which we may think of as a bitstream, although we will keep in mind that the agents have human limitations, and that we will want to support efficient phonetic and written encodings of the bitstream.

We will arbitrarily suppose that these two agents think of the world as consisting of discrete objects and relations, and wish to communicate using a finite alphabet of symbols, rather than (say) modelling the world in terms of smooth fields and communicating by smoothly varying analog signals.

We have available a number of formally equivalent ways of representing the knowledge possessed by our two agents. Suppose one agent knows that John loves Mary. We may use the predicate logic and represent this as Loves(John,Mary). We may use relations, and place a 2-tuple [John, Mary] in the two-column relation Loves. Or, we may imagine John and Mary to be two labelled nodes in a graph, and Loves to be a labelled, directed edge joining them. Since we normally think of parsetrees as directed graphs, it will be convenient to use the graph representation of knowledge

as well: We will think of the internal knowledge store of our two communicating agents as consisting of (very) large graphs with labelled directed edges, some of the nodes also having labels. Our agents wish to communicate by exchanging fragments of these graphs.

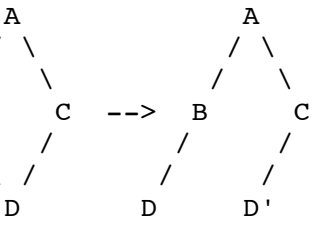
Our problem, then is to supply a good encoding scheme allowing these graph fragments to be linearized, sent through a bitstream channel, and reconstructed at the far end. We require that

1) The reconstruction be unambiguous -- no guesswork.
2) The encoding be as compact as possible -- channel bandwidth is considered a precious commodity.
3) The en/decoding system be a simple as possible.
4) The en/decoding system be human-usable -- it must cater to the specific strengths and weaknesses of human linguistic machinery.

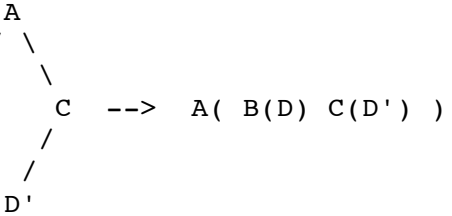
Problem analysis

So far as I can discover, the only reasonable general plan of attack is as follows:

1) Convert the graph fragment into a tree (or, more generally, a forest of trees) by selecting some node(s) as tree roots, and then duplicating nodes as necessary to satisfy the tree constraint. The duplicate nodes are given anaphoric labels so the listener will be able to merge them again. Example:



2) Linearize the tree by doing a treewalk, introducing syntactic markers sufficient to allow unambiguous reconstruction of the original tree structure. Example:



3) Encode the linearized tree in the chosen bitstream by mapping the abstract symbols to concrete written or phonetic encodings.

I'm going to skip Step 1, because I haven't found much

to say about it. The humanly usable systems are not very satisfactory from a formal pointer of view -- English uses anaphora like 'it' and 'that' to refer to previous subexpressions, and depends on pure guess-work on the listener's part to deduce which subexpression is intended. Past loglans have used formally precise schemes based on mental stacks and counting backward, schemes which are generally found to be unworkable in practice. I don't see anything profound in either class of system. It's easy to hack up a suitable instance of either.

Step two, linearizing the tree, involves basically putting all the labels on the tree in a row, and adding just enough syntactic information to allow later reconstruction of the original tree structure. How much information is this? We will suppose that nodes in the tree have zero, one, or two children. (In principle, any tree can be converted to this form, and in practice human languages seem to favor this sort of binary syntactic structure.)

For analytic simplicity, consider a strictly binary tree. Half the nodes are leafs, and half are internal nodes on the tree. We cannot possibly reconstruct the tree without being able to deduce which are which. A one-bit tag on each node is exactly necessary and sufficient to encode the distinction. Now strip all the leafs off the tree. The reduced tree is again half leafs and half internal nodes, and the argument may be iterated until only the root node is left. Thus, half the nodes in the tree need one bit of structure information, one quarter of the nodes need two bits of structure information, one eighth of the nodes need three bits of structure information, and so forth. On the average, a word needs

$$\sum_{i=1}^{\infty} 0.5^{i} * i = 2$$

bits of tree-structure information if the original tree structure of the sentence is to be recovered by the listener.

Thus, an optimal linearizing algorithm should add about two bits of information per node to the output string. If it's much more, you're probably wasting bandwidth, and it it's much less, you're probably fooling yourself. Naturally, the information doesn't necessarily have to be implemented as bit-tags -- it could be segregated as a prefix string to the sentence, or whatever. But it has to provided for in some fashion.

Step 3, providing a concrete surface representation for the abstract symbol string: For analytical simplicity, we will think of encoding the symbol

string into a bitstream, deferring the task of encoding the bitstring as inksplats, handwaving, vocal-cord twanging or whatever. The human-usability constraint basically restricts us to one fixed surface representation for each symbol -- fancy compression schemes which dynamically compute new representations for the symbol as a function of the immediately preceding speechstream are far beyond capabilities of the human listener. Thus, the best we can do on the efficiency front is to assign shorter codes to frequently-used symbols. Our unique-interpretation contraint requires us to choose a bitstream encoding technique which can be unambiguously resolved into the intended stream of symbols by the listener.

So: we need a left-to-right resolvable variable-length set of bitstring encodings selected on the basis of symbol frequencies. Huffman encodings are known to be an optimal solution to this problem: Any reasonable practical system should be either an adaptation of Huffman coding, or something which can be shown to be equally simple and effective.

A near-optimal solution

We adopt the alphabet BCDF GHJK LMNP STVZ. (It is handy to have the alphabet size be a power of two. Eight letters would be less concise, thirty-two would be tough to map onto the standard twenty-six char character set. The particular sixteen letters chosen don't matter.) To encode an arbitrary bitstream efficiently, we use these sixteen letters as a hex encoding according to the following scheme. (The capital letters in the right two columns give the intended pronunciation of each letter when used as a vowel and when used as a consonant.)

LETTER	VALUE	VOWEL	CONSONANT
b	0	bEt	Bet
c	1	shApe	SHape
d	2	dIp	Dip
f	3	fOUGHt	Fought
g	4	gUY	Guy
h	5	bOOt	THing
j	6	bOAt	aZure
k	7	kEEp	Keep
l	8	REd	THese
m	9	pREy	Mom
n	10	pRInt	priNt
p	11	pROp	Prop
s	12	tRIte	Site
t	13	tRUE	True
v	14	ROver	roVer

```
    z       15       bREEze    breeZe
```

Thus we can encode an arbitrary bitstring into letters by breaking it into groups of four bits, and replacing each group by the corresponding letter according to the above table:

```
   0001 0110 0001 0110 1101 ...
     c    j    b    j    t  ...
```

By providing both a vowel and a consonant pronunciation for each letter, and using them alternately, we can pronounce arbitrary strings of letters without difficulty. This is important: It modularizes our language design by decoupling our word-encodings from the details of the human vocal tract, letting us concentrate on other issues. Other than that, the particular letters and pronunciations chosen don't matter much, and might be changed for a non-European audience. It simplifies learning to retain alphabetical ordering, of course, along with the traditional pronunciations of letters where practical. The second eight vowels are simply the first eight with an 'r' prepended. With the suggested pronuciations, the above example "cjbjt" would be pronounced "showboat".

We will call the smallest semantically meaningful letter-sequences in our target language "affixes". These may be complete words, but experience with previous loglans suggests that it is important that one allow for internal structure in words -- we should be free construct a word out of one or more primitive roots, together perhaps with some prefixes and suffixes.

To group letters into affixes, we know we want a Huffman-style expanding-opcode sort of scheme. A simple and effective one is simply to have the number of leading 1 bits in the affix give the number of trailing letters in the affix. This gives us the following infinite set of affixes, where '?' may be any single char:

```
Length 1 (     8 affixes):   b  d  g  j  l  n  s  v
Length 2:(    64 affixes):    c?    h?    m?    t?
Length 3:(   512 affixes):       f??         p??
Length 4:(  4096 affixes):             k???
Length 5:( 32768 affixes): zb???  zd???  zg???  zj???
                           zl???  zn???  zs???  zv???
Length 6:(262144 affixes): zc???? zh???? zm???? zt????
```

... with infinitely more to follow, eight times more for each length. Since these affixes are based on a Huffman-style encoding scheme, any arbitrary sequence of these affixes may be concatenated to form a word, which word can always be uniquely resolved by the

listener into its constituent affixes. The listener would probably simply do this by intuition, being familiar with the individual affixes, but can also do this formally and consciously by reasoning, for example, that any affix beginning with 'm' must necessarily be exactly two letters long. Thus,

    mzdplzgczjv

can only be

    mz d lz g cz j v

This is important, because the listener _must_ be able to uniquely resolve pauseless speech into its constituent words and affixes if unambiguous communication is to be achieved, and we _must_ be able to use arbitrary sequences of affixes if the language design is to stay clean and simple.

We just solved the problem of uniquely resolving the bitstring into affixes: Now we are faced with the similar problem of uniquely resolving the resulting affix-stream into words. How should this be done?

We have established that words must carry an average overhead of two bits each of tree-structure information, so we certainly want our words to average about eight bits each at a minimum, just to amortize the overhead decently. Humans can't reasonably learn a vocabulary of more than one hundred thousand words, so the average word length should certainly be less than seventeen bits. Eight to sixteen bits per word is thus the reasonable design goal.

Suppose we mark word boundaries by using a reserved bitstring. How many bits long should such a bitstring be, in order to minimize verbosity? If we make the word-boundary marker too long, we will clearly make our language more verbose. Less obviously, if we make the word-boundary marker too short, we will also make our language more verbose: By pre-empting valuable short-word space, we force other frequently-used affixes to use longer representations than they deserve, thus indirectly lengthening the average utterance. The correct length may be derived from a little elementary information theory.

A bitchannel carries the most information when it looks completely random. To the extent that you can predict incoming bits, they are carrying less information to you than they could have. If you can predict them all exactly, the channel is not telling you anything at all. Any language feature which makes the speechstream look less random is introducing redundancy, and hence ultimately adding to the average verbosity of utterances in the language. (One may wish to deliberately add redundant information to a message in order to allow

the listener to detect and correct errors. This is
a separate issue.)

Applying the above observation to our word-boundary
problem, we should select our word-boundary marker
so that it would chop a truly random bitstream into
"words" with an average length of eight to sixteen bits.

Imagine we divide the bitstream into one-bit codons.
On the average, every other codon will match our
marker, and our words will average 2*1==2 bits.

If we divide the bitstream into two-bit codons,
one codon out of four will match our boundary
marker, and words will average 4*2==8 bits.

If we divide the bitstream into three-bit codons,
one codon out of eight will match our boundary
marker, and words will average 8*3==24 bits.

Thus, we may conclude that whatever syntactic mechanism
we use to clump affixes together to form words should
ideally add about two bits per word to the bitstream.

This is a problem: Since humans are not terribly
good at shift-and-mask operations, we have set things
up so all affixes are of length 4*N bits. We don't
have any two-bit affixes! Luckily, our other problem
provides the solution: We know we need to attach
about two bits of tree-structure information to
each word to record the tree structure. Two bits
of end-of-word-indicator affix plus two bits of
tree structure yields a four-bit field -- just
the length of our letters. Thus our tentative
solution to the word-resolution problem: A
word is a string of affixes ending with one of the
reserved affixes 'l', 'n' 's' 'v'. (Any four of the
eight single-letter affixes would do just as well.)
Note that this does _not_ mean that a word ends with
the first 'l', 'n', 's', or 'v' letter! These letters
will frequently occur inside of multi-letter affixes.
A word is ended only when one of these letters appears
as a single-letter affix. Example words:

```
 l      A one-char   word made of the single affix   l
 bl     A two-char   word made of the two    affixes b l
 cln    A three-char word made of the two    affixes cl n
 bdv    A three-char word made of the three  affixes b d v
 fvlczs A five-char  word made of the three  affixes fvl cz s
```

Assembling words into parsetrees: The simplest approach
is a precedence grammar. But instead of assigning each
word a fixed precedence, we will use the word ending to
explicitly specify the precedence for each word:

```
 Words ending in the affix 'l' are leafs.
 Words ending in the affix 'n' bind most tightly.
 Words ending in the affix 's' bind next most tightly.
 Words ending in the affix 'v' bind next most tightly.
```

This allows us to express any desired sentence structure without recourse to parentheses. (Humans are not very good at dealing with parentheses, as any Lisp programmer can attest!) Thus, using non-leaf words as infix binary operators for the moment,

```
    bl cln dl bdv gl  ==  (bl cln dl) bdv gl
```

but

```
    bl clv dl bdl gl  ==  bl clv (dl bdl gl)
```

just as

```
    a  *   b   +  c   ==  (a  *   b)  +   c
```

but

```
    a  +   b   *  c   ==  a   +  (b   *   c)
```

except that we have divorced the meaning of a word from its precedence and can specify them separately, whereas conventional mathematical welds the two together.

In practice, the above four precedence levels provide enough machinery to resolve most sentence structures. In principle, however, there is no limit to the number of precedence levels we might need. Thus, we amend our definition of a 'word' to provide an infinite number of end-of-word affixes: The alphabetically last four affixes in each affix length group are end-of-word affixes, and each such affix binds less tightly than the previous one:

End-of-word-affix	Precedence
l	0
n	1
s	2
v	3
ts	4
tt	5
tv	6
tz	7
pzs	8
pzt	9
pzv	10
pzz	12
kzzs	13
kzzt	14
kzzv	15
kzzz	16
zvzzs	17
zvzzt	18

```
          zvzzv               19
          zvzzz               20

           ...               ...
```

This provides us with a pedantically complete system,
but only the first four, or at the very most eight,
affixes are ever likely to be needed in practice.
In the limit of large expressions, this system uses
an average of about 2.6 bits of tree-structure
information per word, compared to the optimal 2.0
bits, incurring a verbosity penalty of about 3-6%
in order to keep things simple for humans.

To distinguish between unary and binary operators, we
borrow a trick from C. C manages to use '*' as both
a unary and binary operator without incurring ambiguity.
How? A little thought shows that this works because:
1) C syntax forbids the appearance of two leafs
   without an intervening binary operator.
2) '*' is used only as a prefix unary operator, never
   as a postfix unary operator. (Mutual exclusion
   is the key -- it would work fine if the reverse
   was true.)
Thus, we can avoid the overhead of explicitly
specifying the arity of a word by adopting a tree
syntax like:

```
%token PRIORITY_3_WORD
%token PRIORITY_2_WORD
%token PRIORITY_1_WORD
%token PRIORITY_0_WORD

%%

  . . .

binary_priority_3_expr
:                                        unary_priority_3_expr
| binary_priority_3_expr PRIORITY_3_WORD  unary_priority_3_expr
;
unary_priority_3_expr
:                                       binary_priority_2_expr
|                        PRIORITY_3_WORD binary_priority_2_expr
;

binary_priority_2_expr
:                                        unary_priority_2_expr
| binary_priority_2_expr PRIORITY_2_WORD  unary_priority_2_expr
;
unary_priority_2_expr
:                                       binary_priority_1_expr
|                        PRIORITY_2_WORD binary_priority_1_expr
;

binary_priority_1_expr
:                                        unary_priority_1_expr
| binary_priority_1_expr PRIORITY_1_WORD  unary_priority_1_expr
;
unary_priority_1_expr
```

```
:                                                PRIORITY_0_WORD
|                       PRIORITY_1_WORD          PRIORITY_0_WORD
;
```

Thus, using the above grammar

```
    bl cln bdn gl  ==  bl cln (bdn gl)
```

just as

```
    a  *   *   b   ==  a  *   (*   b)
```

in C.

Example

Let's build a toy language using this syntax. We'll take common English words and assign them affixes, then make up some sentences using this nano-vocabulary. You can run these sentences through the supplied parser if you wish.

English word	Toy affix
a	b
and	cd
be	hk
but	mt
can	cn
car	cc
do	hd
drive	mg
for	tf
get	ct
go	mj
have	hv
how	hj
I	g
if	hf
in	md
is	cz
it	td
like	tk
may	mc
not	d
of	cv
on	hn
or	cj
she	ck
shoe	ch
shy	cg
so	hs
that	hm
the	hb
they	hc
this	hg
to	th
was	hz
what	ht

```
   will            ml
   with            hp
   you              j

Sample sentences:

  Gl tkn jl.
  I  like you.

  Ckl tkn gl.
  She likes me.

  Gl mgn hbn ccl.
  I drive the car.

  Gl mgn ckn ccl.
  I drive her car.

  Gl cnn mgn bn ccl.
  I can drive a car.

  Gl tks ckl mgn gn ccl.
  I like her driving my car.

  Gl mln mgn gn ccl thn jl.
  I  will drive my car to you.
```

Summary

We have shown that a simple, near-optimal surface syntax for a loglan may be obtained by:

1) Adopting an alphabet such as bcdf ghjk lmnp stvz.
2) Using a Huffman-style expanding-opcode technique to define a set of affixes of varying lengths.
3) Selecting a subset of these affixes to simultaneously mark the end of a word and indicate the binding precedence of that word.
4) Using a simple precedence grammar based on unary and binary operators.

Compared to existing loglans, such a scheme:

* Is much simpler. The informal English word-resolution algorithm for existing loglan run for a dozen pages or more, and the grammars for hundreds of rules.
* Potentially allows for mechanical recognition of continuous speech. Existing loglans require pauses between about 30% of the words in a sentence.
* Is suited to laboratory studies of the Sapir-Whorf Hypothesis. Any such studies will require the construction of a test language which exhibits some property of interest, together with a control language which differs from the test language only in this property. A test group will learn the test language and a control group the control language. Existing loglans are too complex to be quickly built and learned.

```
* Possesses a certain elegance.
-------------planb-------------------
# Recompile planb under Sun Unix (on a 3/160):
cc planb.c -o planb
-------------planb.c-----------------
/************************************************************************/
/*                              planb.c                                 */
/************************************************************************/

/************************************************************************/
/*  This program parses a simple human language syntax.  It may be      */
/*  compiled with simply "cc planb.c -o planb".                         */
/************************************************************************/

/************************************************************************/
/*                            contents                                  */
/*                                                                      */
/*      main                                                            */
/*      findAffixes                                                     */
/*      findParse                                                       */
/*      findWords                                                       */
/*      affix_ends_word TRUE iff affix marks end of word                */
/*      hexVal          Convert bcdf ghjk lmnp stvz to binary value     */
/*      printIndent     Indent a line of pretty-print output            */
/*      printNode       Recursive part of parsetree prettyprint         */
/*      printTree       Parsetree prettyprint                           */
/*      toLower         Convert uppercase letters to lowercase          */
/*                                                                      */
/************************************************************************/

/************************************************************************/
/*                            history                                   */
/*                                                                      */
/* 90Jan17 CrT  Created.                                                */
/************************************************************************/

/* Width of display, used for prettyprinting: */
#define SCREENwIDTH 80



#define MAIN    1

#include

#define VERSION "1.00 90Jan17"

#define loop     while (1)

#ifndef TRUE
#define TRUE     1
#endif

#ifndef FALSE
#define FALSE    0
#endif


/*********************************/
/* Array to hold pointers to the */
/* affixes in the input buffer:  */
```

```
/*********************************/
#define MAXaFFIXES  100
char *   Affix[ MAXaFFIXES ];
int      Affix_count;


/*********************************/
/* Array to hold pointers to the */
/* affixes in the input buffer:  */
/*********************************/
#define MAXwORDS  100
char *   Word[            MAXaFFIXES ];
int      Word_precedence[ MAXaFFIXES ];
int      Max_precedence_found;
int      Word_count;
int      This_word;


/*************************************/
/* Buffer to read the planb input into: */
/*************************************/
#define MAXiN    10000
char     Inbuf[      MAXiN ];
char*    Inbuf_lim;

/*************************************/
/* Buffer to hold the initial parsetree, */
/* as a parenthesized string of ints. We */
/* also prettyprint the tree into this   */
/* buffer, later on in printTree():      */
/*************************************/
#define MAXoUT   5000
int      outbuf[    MAXoUT ];



/*********************************************************************/
/* When we've successfully parsed the input, we're left with a giant */
/* parenthesized expression in outbuf[].  We then construct a proper */
/* nodes-and-pointers parsetree using the following node structure:  */
/*********************************************************************/
#define MAXnODE      100
struct node {
    int             word;    /* Index into Word[] array  */
    struct node *   kidL;
    struct node *   kidR;
} NodeList[ MAXnODE ], *NextNode;

int Verbose;


/*********************************************************************/
/*      main                                                         */
/*********************************************************************/
main() {
    struct node *   findParse();
    struct node *   parseTree;
    Verbose = TRUE;

    /* Sign-on message: */
    printf("\n              -- planb --"             );
```

```
    printf("\n          CurrenT Software's"          );
    printf("\n      Public-domain LANguage Basher"   );
    printf("\n         Version %s\n", VERSION        );

    printf("\nSample sentences you can enter:\n");
    printf("\nGl tkn jl.");
    printf("   (\"I  like you.\")");

    printf("\nCkl tkn gl.");
    printf("   (\"She likes me.\")");

    printf("\nGl mgn hbn ccl.");
    printf("   (\"I drive the car.\")");

    printf("\nGl mgn ckn ccl.");
    printf("   (\"I drive her car.\")");

    printf("\nGl cnn mgn bn ccl.");
    printf("   (\"I can drive a car.\")");

    printf("\nGl tks ckl mgn gn ccl.");
    printf("   (\"I like her driving my car.\")");

    printf("\nGl mln mgn gn ccl thn jl.");
    printf("   (\"I will drive my car to you.\")\n");

    /* Toplevel read-eval-print loop: */
    loop {

        printf("\n\n\nEnter sentence:     ");
        gets(Inbuf);
        if (!*Inbuf) exit(0);

        /* Break the input textstring into affixes: */
        if (!findAffixes())   continue;

        /* Group the affixes into words: */
        findWords();

        /* Group the words into a parsetree: */
        parseTree = findParse();
        if (Verbose)   printTree( parseTree );

    }
}

/*********************************************************************/
/*      findAffixes                                                  */
/*********************************************************************/
findAffixes() {
    char *   s;
    char *   d;

    s           = Inbuf;

    /* Kill whitespace, check for unwanted chars */
    /* in input, and fold rest to lower case:    */
    for (s = d = Inbuf;   *s = *d++;   ) {

        /* Convert to lower case: */
        *s = toLower( *s );
```

```
    /* Delete blanks, complain about anything else but bcdf ghjk lmnp stvz: */
    switch (*s) {

    case 'b':   case 'c':   case 'd':   case 'f':
    case 'g':   case 'h':   case 'j':   case 'k':
    case 'l':   case 'm':   case 'n':   case 'p':
    case 's':   case 't':   case 'v':   case 'z':
        ++s;
        break;

    case ' ':
    case '\t':
        break;

    default:
        printf("Invalid char '%c'\n", *s );
        return FALSE;
    }
}
if (Verbose)   printf("\nFolded   input: '%s'\n",Inbuf);

/* Insert a blank after every char in input string: */
d     = &Inbuf[ 2*(s-Inbuf) ];    /* Set d twice as far into Inbuf as s.   */
*d--  = *s--;                     /* Deposit new terminal nul.             */
while (d > Inbuf)     {   *d-- = ' ';   *d-- = *s--;   }

/* Break input into affixes, following rule that number of       */
/* trailing chars in affix is given by number of leading 1 bits: */
s = d = Inbuf;    /* Actually, this is already true. */
Affix_count = 0;
while (*s) {

    int hex = hexVal( *s );
    int leading_1s = 0;

    /* Remember start of affix, and keep count of affixes found: */
    Affix[ Affix_count++ ] = d;

    /* Count number of leading 1s: */
    while (hex == 0xF) {
        leading_1s += 4;
        *++d = *(s+=2);
        if (!*s) { printf("Input ends with incomplete affix"); return FALSE; }
        hex = hexVal( *s );
    }
    while (hex & 1)     {   hex >>= 1;   ++leading_1s;   }

    /* Eat that number of trailing chars: */
    while (leading_1s--) {
        *++d = *(s+=2);
        if (!*s) { printf("Input ends with incomplete affix"); return FALSE; }
    }

    /* Lay down terminal nul for affix: */
    *++d = '\0';

    /* Bump s to start of next affix: */
    *++d = *(s+=2);
}
```

```
    if (Verbose) {
        printf("\nAffixes  found:");
        {
            int i;
            for (i = 0;   i < Affix_count;   ++i)     printf(" %s",Affix[i]);
            printf("\n");
        }
    }

    /* Remember first free location in Inbuf[]: */
    Inbuf_lim = d;
}

/*********************************************************************/
/*      findParse                                                    */
/*********************************************************************/
struct node *   findParse_unary(  precedence )
int                              precedence;
{
    struct node *   findParse_binary();
    struct node *   nod;


    /* If we're out of words or don't have a word of     */
    /* correct precedence, just return right expression: */
    if (This_word == Word_count                          ||
        Word_precedence[ This_word ]   !=   precedence
    ) {
        return   findParse_binary( precedence-1 );
    }

    /* Build and return a parsetree node for our word: */
    nod       = NextNode++;
    nod->word = This_word++;
    nod->kidL = (struct node *) FALSE;
    nod->kidR = findParse_unary( precedence );
    return nod;
}
struct node *   findParse_binary( precedence )
int                              precedence;
{
    struct node *   left;
    struct node *   nod ;

    /* Test for limiting case of recursion: */
    if (precedence < 0)            return (struct node *) FALSE;

    /* Parse left subexpression: */
    left = findParse_unary( precedence );

    /* If we're out of words or don't have a word of    */
    /* correct precedence, just return left expression: */
    while (This_word < Word_count                        &&
        Word_precedence[ This_word ]   ==   precedence
    ) {
        /* Build and return a parsetree node for our word: */
        nod       = NextNode++;
        nod->word = This_word++;
        nod->kidL = left;
        nod->kidR = findParse_unary( precedence );
        left      = nod;
```

```
    }
    return left;

}
struct node *   findParse() {
    /*****************************************************************/
    /* Here we group the words into a parsetree.  The parse is based on  */
    /* the idea that priority-0 words are leafs, and the other words are */
    /* unary or binary operators. Whether such words are unary or binary */
    /* is determined from context, using a grammar much like that for    */
    /* the '*', '-' and '&' unary/binary operators of C.  The grammar is */
    /* technically infinite since we have an infinite number of possible */
    /* precedences.  Here it is in YACC syntax:                          */
    /*                                                                   */
    /*   . . .                                                           */
    /* %token PRIORITY_3_WORD                                            */
    /* %token PRIORITY_2_WORD                                            */
    /* %token PRIORITY_1_WORD                                            */
    /* %token PRIORITY_0_WORD                                            */
    /*                                                                   */
    /* %%                                                                */
    /*                                                                   */
    /*   . . .                                                           */
    /*                                                                   */
    /* binary_priority_3_expr                                            */
    /* : unary_priority_3_expr                                           */
    /* | unary_priority_3_expr PRIORITY_3_WORD  binary_priority_3_expr   */
    /* ;                                                                 */
    /* unary_priority_3_expr                                             */
    /* :                                       binary_priority_2_expr    */
    /* |                     PRIORITY_3_WORD   unary_priority_3_expr     */
    /* ;                                                                 */
    /*                                                                   */
    /* binary_priority_2_expr                                            */
    /* : unary_priority_2_expr                                           */
    /* | unary_priority_2_expr PRIORITY_2_WORD  binary_priority_2_expr   */
    /* ;                                                                 */
    /* unary_priority_2_expr                                             */
    /* :                                       binary_priority_1_expr    */
    /* |                     PRIORITY_2_WORD   unary_priority_2_expr     */
    /* ;                                                                 */
    /*                                                                   */
    /* binary_priority_1_expr                                            */
    /* : unary_priority_1_expr                                           */
    /* | unary_priority_1_expr PRIORITY_1_WORD  binary_priority_1_expr   */
    /* ;                                                                 */
    /* unary_priority_1_expr                                             */
    /* :                                             PRIORITY_0_WORD     */
    /* |                     PRIORITY_1_WORD   unary_priority_1_expr     */
    /* ;                                                                 */
    /*****************************************************************/

    /* Move all the parsetree nodes to the freelist: */
    NextNode = NodeList;

    /* Reset global current-word pointer to start of Word[]: */
    This_word = 0;

    /* Now a trivial recursive-descent parse does the trick: */
    return   findParse_binary( Max_precedence_found );
}
```

```
/*********************************************************************/
/*      findWords                                                    */
/*********************************************************************/
findWords() {

    /* Here we fill Words[] and Word_precedence[] with the text string */
    /* and precedence respectively of each word found in the input:    */
    char *   s;
    char *   d;
    int      a = 0;

    Word_count              = 0;
    Max_precedence_found    = 0;

    /* We will write the words we find into the free space at the end of Inbuf: */
    d          = Inbuf_lim;

    /* While unprocessed affixes remain: */
    while (a < Affix_count) {

        int precedence  = 0;
        int end_of_word = affix_ends_word( Affix[a], &precedence );

        /* Remember start of word, and keep count of words found: */
        Word[ Word_count ] = d;

        /* Copy affix to where we're constructing our word: */
        for (s = Affix[a];   *d++ = *s++;   );
        ++a;

        /* Keep appending affixes until we read end of word or input: */
        while (!end_of_word   &&   a < Affix_count) {

            d[-1]       = '-';
            for (s = Affix[a];   *d++ = *s++;   );
            end_of_word = affix_ends_word( Affix[a], &precedence );
            ++a;
        }

        /* Remember precedence of the word: */
        Word_precedence[ Word_count++ ] = precedence;

        /* Remember maximum precedence found: */
        if (Max_precedence_found < precedence)   Max_precedence_found = precedence;
    }

    if (Verbose) {
        int p;
        printf("\nWords found, by precedence:\n");
        for (p = Max_precedence_found;   p >= 0;   --p) {
            int w;
            printf(   "Precedence %3d:",   p   );
            for (w = 0;   w < Word_count;   ++w) {
                if (Word_precedence[w] == p)     printf(   " %s",   Word[ w ]   );
                else {
                    int i = strlen( Word[w] ) +1;
                    while (i--)   printf(" ");
                }
            }
            printf("\n");
```

```
        }
    }

    return TRUE;
}

/*************************************************************************/
/*      affix_ends_word      TRUE iff affix marks end of word            */
/*************************************************************************/
affix_ends_word( affix, precedence )
char *          affix;              /* input */
int  *                precedence;   /* valid only if fn returns TRUE */
{
    if (!strcmp( affix,     "l" )) { *precedence =  0;   return TRUE; }
    if (!strcmp( affix,     "n" )) { *precedence =  1;   return TRUE; }
    if (!strcmp( affix,     "s" )) { *precedence =  2;   return TRUE; }
    if (!strcmp( affix,     "v" )) { *precedence =  3;   return TRUE; }

    if (!strcmp( affix,    "ts" )) { *precedence =  4;   return TRUE; }
    if (!strcmp( affix,    "tt" )) { *precedence =  5;   return TRUE; }
    if (!strcmp( affix,    "tv" )) { *precedence =  6;   return TRUE; }
    if (!strcmp( affix,    "tz" )) { *precedence =  7;   return TRUE; }

    if (!strcmp( affix,   "pzs" )) { *precedence =  8;   return TRUE; }
    if (!strcmp( affix,   "pzt" )) { *precedence =  9;   return TRUE; }
    if (!strcmp( affix,   "pzv" )) { *precedence = 10;   return TRUE; }
    if (!strcmp( affix,   "pzz" )) { *precedence = 11;   return TRUE; }

    if (!strcmp( affix,  "kzzs" )) { *precedence = 12;   return TRUE; }
    if (!strcmp( affix,  "kzzt" )) { *precedence = 13;   return TRUE; }
    if (!strcmp( affix,  "kzzv" )) { *precedence = 14;   return TRUE; }
    if (!strcmp( affix,  "kzzz" )) { *precedence = 15;   return TRUE; }

    if (!strcmp( affix, "zvzzs" )) { *precedence = 16;   return TRUE; }
    if (!strcmp( affix, "zvzzt" )) { *precedence = 17;   return TRUE; }
    if (!strcmp( affix, "zvzzv" )) { *precedence = 18;   return TRUE; }
    if (!strcmp( affix, "zvzzz" )) { *precedence = 19;   return TRUE; }

    if (!strcmp( affix,"ztzzzs" )) { *precedence = 20;   return TRUE; }
    if (!strcmp( affix,"ztzzzt" )) { *precedence = 21;   return TRUE; }
    if (!strcmp( affix,"ztzzzv" )) { *precedence = 22;   return TRUE; }
    if (!strcmp( affix,"ztzzzz" )) { *precedence = 23;   return TRUE; }

    /* There are infinitely more (the last 4 numerically of each affix size) */
    /* but why be pedantic?  Only the first eight have any real chance of    */
    /* being used.                                                           */

    return FALSE;
}

/*************************************************************************/
/*      hexVal          Convert bcdf ghjk lmnp stvz to binary value      */
/*************************************************************************/
hexVal( c )
int     c;
{
    switch (c) {

    case 'b':    return 0x0;
    case 'c':    return 0x1;
    case 'd':    return 0x2;
```

```
    case 'f':    return 0x3;

    case 'g':    return 0x4;
    case 'h':    return 0x5;
    case 'j':    return 0x6;
    case 'k':    return 0x7;

    case 'l':    return 0x8;
    case 'm':    return 0x9;
    case 'n':    return 0xA;
    case 'p':    return 0xB;

    case 's':    return 0xC;
    case 't':    return 0xD;
    case 'v':    return 0xE;
    case 'z':    return 0xF;

    default:
        printf(   "hexVal: internal error '%c'",   c   );
        exit(1);
    }
}

/*********************************************************************/
/*      printIndent     Indent a line of pretty-print output         */
/*********************************************************************/
char *   printIndent( depth, out )
int                  depth;
char *                      out;
{
    int   col;

    *out++ = '\n';
    for (col = 0;   col < 2*depth;   ++col)   *out++ = ' ';

    return out;
}

/*********************************************************************/
/*      printNode       Recursive part of parsetree prettyprint      */
/*********************************************************************/
char* printNode( nod, depth, out0 )
struct node *    nod;
int                  depth;
char *                      out0;
{
    static char *   lParen = "<([{";
    static char *   rParen = ">)]}";
    int             i;
    char *          out = out0;

    if (!nod)    return out;

    /* Leaf node? */
    if (!nod->kidL   &&   !nod->kidR) {

        /* Yes: */
        sprintf( out, "%s ", Word[ nod->word ]  );
        out += strlen( out );
        return out;
    }
```

```
    /* Internal node.  Try fitting it all on one line: */

    /* Open clause: */
    sprintf( out,  "%c%s: ",   lParen[ depth & 3 ],   Word[ nod->word ]   );
    out += strlen( out );

    /* Do kids: */
    out = printNode( nod->kidL, depth+1, out );
    out = printNode( nod->kidR, depth+1, out );

    /* Close clause: */
    sprintf( out,  "%c ",      rParen[ depth & 3 ]   );
    out += strlen( out );

    /* Did it all fit on one line? */
    if ((out - out0) + 2*depth   >=   SCREENwIDTH) {

        /* No, break it up into multiple lines: */

        /* Erase previous one-line try: */
        out = out0;

        /* Open clause: */
        sprintf( out,  "%c%s: ",   lParen[ depth & 3 ],   Word[ nod->word ]   );
        out += strlen( out );

        /* Do kids: */
        ++depth;
        if (nod->kidL) {
            out = printIndent( depth, out );
            out = printNode( nod->kidL, depth, out );
        }
        if (nod->kidR) {
            out = printIndent( depth, out );
            out = printNode( nod->kidR, depth, out );
        }

        /* Close clause: */
        --depth;
        out = printIndent( depth, out );
        sprintf( out,  "%c ",      rParen[ depth & 3 ]   );
        out += strlen( out );
    }

    return out;
}

/*****************************************************************/
/*      printTree       Parsetree prettyprint                    */
/*****************************************************************/
printTree(      self )
struct node *   self;
{
    printf( "\nParse:    " );
    printNode( self,  5, (char*)outbuf );
    puts(                (char*)outbuf );
}

/*****************************************************************/
/*      toLower         Convert uppercase letters to lowercase   */
```

```
/*****************************************************************/
toLower( c )
int      c;
{
    if (c < 'A'   ||   c > 'Z')   return c;

    return   c  +  ('a' - 'A');
}
-----------------------------------------------------------------

--Submitted by Jeff Prothero, jsp@milton.u.washington.edu
  May 9, 1990

 -------------------- cut here --------------------
```