1. Speech signal annotation. Presented a file containing data from continuous speech signal, this program will perform a Fast Fourier Transform on segments of the data and write the spectral coefficients to a new file in frame file format with a target value assigned to each frame. The user is queried at each frame for the appropriate tag to assign. An ordered list of tags is taken from a command line so that the user does not need to select tags. The tags represent spellings of phonemes.

The input file consists of nothing but raw **short int**'s, the output of the command

`audioconvert -f format=raw` *filename*`.au > `*filename*`.raw`

where the file *filename*`.au` was generated by Sun's `audiotool`. The Fourier and spectral power routines are adapted from *Numerical Recipes*, pages 381–429. Here's the basic program layout.

⟨ Include Files 2 ⟩
⟨ Global Definitions 4 ⟩
⟨ Function Definitions 6 ⟩
⟨ Main Program 7 ⟩

2. Do the standard sort of includes. I'm using the math library, and I want to be sure to use the definitions of π in the header file.

⟨ Include Files 2 ⟩ ≡
```
#include <stdio.h>
#include <math.h>
```
See also sections 26 and 31.

This code is used in section 1.

3. Here defined are some constants used in analyzing the speech signal. The signal is divided into equally sized segments called frames, and each frame is analyzed as a unit. There are tradeoffs between the size of a frame (in samples) and the spectral resolution. To increase the resolution of the analysis (increasing the number of frequency bins), the frame needs to be larger. However, since a speech signal changes over time (i.e., the frequency distribution is not static), too large a speech frame will blend too many different sounds together in one analysis. To avoid this, one could then sample the speech signal at a higher rate to provide more samples in a shorter stretch of time, but at the cost of higher data storage and processing requirements.

A typical frame in speech recognition research spans a 10 ms time slice, but it is also thought that a single utterance may occupy up to 30 ms. For the phonemic spelling that I am using, a characterizing utterance may occupy as many as 5 or 6 frames.

With all of this in mind, I have settled on a frame resolution of 128 bins, which requires a FFT of length of 256 samples. With a sampling rate of 16 000 Hz, this yields a time slice of 16 ms per frame. At one point, I thought it might be useful to analyze the signal with frames which overlapped by some amount, but I am not currently studying this. `PRECISION` represents the number of bits per sample.

```
#define FFT_LENGTH 256 /* Number of samples for FFT */
#define FRAMESIZE 128 /* Number of spectral coefficients to write per frame */
#define OVERLAP 0 /* Overlap in samples from one frame to the next */
#define SAMPLE_RATE 16000 /* Sampling rate (samples/second) (set by hardware) */
#define PRECISION 16 /* Bits per sample */
```

4. Because these arrays can be large, I need to allocate them in global storage rather than on the stack. The array *allframes* will contain the spectral parameters for all the frames, each of which has `FRAMESIZE` elements. Array *w1* is a work array for the *spctrm*( ) routine, and needs to be twice as long as `FFT_LENGTH`+1. Finally, the data array *arr* (creative, ain't it?) needs to contain the whole input file. Space is allocated for this array by the *read_input_file*( ) routine.

```
⟨ Global Definitions 4 ⟩ ≡
  typedef double oneframe[FRAMESIZE];    /* Spectral information */
  oneframe *allframes;    /* Storage is allocated later. */
  double channel_min[FRAMESIZE];
  double channel_max[FRAMESIZE];
  double channel_ave[FRAMESIZE];
  double w1[2 * (FFT_LENGTH + 1)];    /* Work array for FFT */
  short *arr;    /* The sample array */
  char *the_phrase;    /* Pointer to the command-line phrase */
```

See also sections 5, 8, 10, and 27.

This code is used in section 1.

5. Now let's consider tag-handling. Since the string order needs to be consistent across training sets, I'm going to hard code the tag order here. For n tags, there will be $n+1$ strings, the extra string defining what value to assign when no output node is sufficiently active (a null case). I have arbitrarily grouped the tags as vowels, dipthongs, consonants, and complex consonants. There's really no difference between complex consonants and regular consonants, except that some of the complex ones are spelled with more than one letter. As far as I'm concerned, they're all still just one utterance. It is important that the multi-character strings be loaded first.

```
⟨ Global Definitions 4 ⟩ +≡
  char *tag_strings[] ← {"␣",    /* Null Case */
  "aI","aU","eI","oI","oU","3R",    /* Dipthongs */
  "tS","dZ","D","N","S","T","Z",    /* Complex Consonants */
  "&","A","e","i","I","O","u","U","V",    /* Vowels */
  "b","d","f","g","h","j","k","l","m",    /* Consonants */
  "n","p","r","s","t","v","w","z",    /* Consonants */
  (char *) Λ    /* Null ptr signals end */
  };
```

6. Return the value of the phoneme pointed to by string s.

```
⟨ Function Definitions 6 ⟩ ≡
  int phoneme_value(char *s)
  {
    char **tptr ← tag_strings;
    int count ← 0;
    while (*tptr ≠ Λ) {
      if (strncmp(*tptr, s, strlen(*tptr)) ≡ 0) return count;    /* found the string */
      count++;
      tptr++;
    }
    return −1;    /* String not found */
  }
```

See also sections 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 28, 29, 30, 32, 33, 34, 35, 36, and 37.

This code is used in section 1.

7. And now, here's the main program. Get options and input file from the command line. Determine the size of the structures and read the input file. Perform the spectral analysis, do normalization, and then tag the frames.

⟨ Main Program 7 ⟩ ≡

```
main(int argc, char *argv[])
{
  int nsamples, nframes;
  int i, fc;
  handle_arguments(argc, argv);
  arr ← read_input_file(infilename, &nsamples);
  nframes ← nsamples/(FFT_LENGTH − OVERLAP);    /* Calculate number of frames */
  setup_taglist(nframes);
  allframes ← (oneframe *) malloc(sizeof(oneframe) * nframes);
  if (parm_norm ≡ READNORM)    /* Read normalization params from file */
    input_norm(normfilename, channel_min, channel_max, channel_ave, FRAMESIZE);
  else init_norm(channel_min, channel_max, channel_ave, FRAMESIZE);
  for (i ← fc ← 0; i ≤ nsamples − FFT_LENGTH; i += FFT_LENGTH − OVERLAP, fc++) {
    spctrm(arr, i, allframes[fc], w1, FRAMESIZE);    /* Get power */
    if (parm_norm ≤ 0)    /* Gather data for normalization? */
      get_ave(allframes[fc], channel_ave, FRAMESIZE);
  }
  if (parm_norm ≤ 0)
    calc_norm(allframes, channel_min, channel_max, channel_ave, FRAMESIZE, nframes);
  if (parm_norm ≡ WRITENORM)    /* Write normalization params to file */
    output_norm(normfilename, channel_min, channel_max, channel_ave, FRAMESIZE);
  if (parm_phrase) {
    init_phrase(taglist, nframes);
    audio_init();    /* Open and initialize the audio device */
    editing_loop(nframes);
  }
  else write_frame_file(nframes);
}
```

This code is used in section 1.

8. I'll need a place to store file names. These variables will be filled in by the *handle_arguments*( ) function. Also, put the command-line flags here.

⟨ Global Definitions 4 ⟩ +≡

```
char *infilename;    /* Input file name from command line */
char *outfilename;    /* Output file name derived from input file name */
char *normfilename;    /* File to store normalization values in */
char *annotation_file;    /* File from which external annotations may read */
int parm_phrase ← 0, parm_annofile ← 0, parm_norm ← 0;
```

9. Deal with the command line arguments. Recognized command-line parameters are "`-norm`", "`-afrom`", "`-phrase`", and a file name. Only one of "`-afrom`" or "`-phrase`" may be specified. If "`-afrom`," then the next argument is taken to be the frame file from which annotations will be copied. Otherwise, the argument following "`-phrase`" is understood to be the phonetic spelling with which the speech signal will be annotated. If "`-norm`" is specified, then the next argument specifies a file from which normalization values will be read (if the file exists), or to which they will be written (if the file does not exist). All training datasets used in one session should be normalized with the same values to reduce the variability between samples.

The file name specifies the input file and the basename for framefile output. The output filename is produced by stripping any extension from the input filename, and the string "`.frame`" is tacked onto the end. The resulting filename must not already exist, or the program exits.

```
#define READNORM 1
#define WRITENORM −1

⟨ Function Definitions 6 ⟩ +≡
  handle_arguments(int argc, char **argv)
  {
    char *s, *name;
    FILE *tmp;

    name ← *argv;
    infilename ← Λ;
    argc−−;
    argv++;
    while (argc−−) {
      if (strcmp(*argv, "-phrase") ≡ 0) {
        argc−−;
        argv++;
        parm_phrase++;
        the_phrase ← *argv;
      }
      else if (strcmp(*argv, "-afrom") ≡ 0) {
        argc−−;
        argv++;
        parm_annofile++;
        annotation_file ← *argv;
      }
      else if (strcmp(*argv, "-norm") ≡ 0) {
        argc−−;
        argv++;
        parm_norm++;
        normfilename ← *argv;
      }
      else {
        infilename ← (char *) malloc(strlen(*argv) + 1);
        strcpy(infilename, *argv);
      }
      argv++;
    }
    if (parm_phrase ∧ parm_annofile) {
      fprintf(stderr, "Only␣one␣of␣'-phrase'␣or␣'-afrom'␣is␣allowed.\n");
      exit(1);
    }
    if (infilename ≡ Λ ∨ strlen(infilename) ≡ 0) {
```

```
    fprintf(stderr, "Missing␣an␣input␣file.\n");
    exit(1);
  }
  else {
    outfilename ← (char *) malloc(strlen(infilename) + 7);
    strcpy(outfilename, infilename);
  }    /* Now format the output filename based on the input file name. First, find the last dot which
          indicates a possible extension. */
  s ← outfilename + strlen(outfilename) − 1;
  while (*s ≠ '.' ∧ s > outfilename) s−−;
  if (s ≡ outfilename) strcat(outfilename, ".frame");    /* No extension, concatenate */
  else strcpy(s, ".frame");    /* old extension overwritten */
  if ((tmp ← fopen(infilename, "r")) ≡ Λ) {
    fprintf(stderr, "Input␣file␣\"%s\"␣cannot␣be␣opened.\n", infilename);
    exit(1);
  }
  else fclose(tmp);
  if ((tmp ← fopen(outfilename, "r")) ≠ Λ) {
    fprintf(stderr, "Output␣file␣\"%s\"␣already␣exists.\n", outfilename);
    exit(1);
  }
  if (parm_norm) {
    if ((tmp ← fopen(normfilename, "r")) ≡ Λ) parm_norm ← WRITENORM;
        /* doesn't exist, so we'll write it */
    else {
      parm_norm ← READNORM;    /* Exists, so we'll read it */
      fclose(tmp);
    }
  }
  if (parm_annofile) {
    if ((tmp ← fopen(annotation_file, "r")) ≡ Λ) {
      fprintf(stderr, "Input␣annotation␣file␣\"%s\"␣could␣not␣be␣opened.\n", annotation_file);
      exit(1);
    } else fclose(tmp);
  }
}
```

10. A phrase will be parsed into a doubly-linked list. The phrase elements are converted into values which represent an element's ordering in the *tag_strings* list. Along with the value is stored a starting and ending frame number for the tag.

```
⟨ Global Definitions 4 ⟩ +≡
  struct tag {
    int tag_value;
    int sframe, eframe;
    struct tag *next;
    struct tag *prev;
  };
  struct tag *taglist;
```

11. Convert the command-line phrase into a doubly-linked list of phoneme values stored in *taglist*. The values are derived from the order of strings in *tag_strings*. A phrase containing a character not located in *tag_strings* constitutes a fatal error. All whitespace in the phrase is deleted, but the null case (no identifiable speech) is added to both ends of the phrase.

```
⟨ Function Definitions 6 ⟩ +≡
  parse_phrase(char *s)
  {
    int ph_val;
    struct tag *p;
    p ← taglist ← (struct tag *) malloc(sizeof(struct tag));    /* create header */
    taglist→next ← taglist→prev ← Λ;
    taglist→tag_value ← −1;
    taglist→sframe ← taglist→eframe ← 0;
    insert_node(p, 0);    /* insert null case after header node */
    p ← p→next;    /* Advance to newly created node */
    while (*s ≠ '\0') {
      while (*s ≡ '␣' ∨ *s ≡ '\t' ∨ *s ≡ '\n') s++;    /* skip whitespace */
      ph_val ← phoneme_value(s);
      if (ph_val < 0) {
        fprintf(stderr, "Unrecognized␣character(s)␣in␣phrase␣at:␣|%c|%12-s\n", *s, s + 1);
        exit(2);
      }
      insert_node(p, ph_val);    /* insert new node after p */
      p ← p→next;    /* Advance pointer to newly added node */
      s += strlen(tag_strings[ph_val]);
    }
    insert_node(p, 0);    /* Insert null case after last sound */
  }
```

12. There are three cases for establishing how *taglist* is initialized. If a phrase has been specified, then setup *taglist* from that command-line argument. If an annotation file has been specified, then read that file to find tags. Otherwise, set up the null case where no tags have been specified.

```
⟨ Function Definitions 6 ⟩ +≡
  setup_taglist(int nframes)
  {
    if (parm_phrase) parse_phrase(the_phrase);
    else if (parm_annofile) parse_from_file(annotation_file);
    else {
      taglist ← (struct tag *) malloc(sizeof(struct tag));    /* create header */
      taglist→prev ← Λ;
      taglist→tag_value ← −1;
      taglist→sframe ← taglist→eframe ← 0;
      insert_node(taglist, 0);    /* insert null case after header node */
      taglist→next→sframe ← 0;
      taglist→next→eframe ← nframes − 1;
    }
  }
```

13. Function *init_phrase*( ) traverses the taglist, echoing the phrase elements back to the user, and establishing an initial distribution of frames through the list.

```
⟨ Function Definitions 6 ⟩ +≡
  init_phrase(struct tag *list, int frames)
  {
    int n_els, cur_el, elc;
    float frame_spread;
    struct tag *p ← list→next;    /* Skip header node */
    n_els ← 0;
    while (p ≠ Λ) {    /* Count list elements */
      n_els++;
      p ← p→next;
    }
    p ← list→next;    /* Back to the list beginning */
    frame_spread ← (float) frames/(float) n_els;
    cur_el ← 0;
    elc ← 0;
    printf("%d␣Frames,␣%d␣Elements,␣Phrase:\n␣␣", frames, n_els);
    while (p ≠ Λ) {
      p→sframe ← (int)(cur_el * frame_spread + 0.5);
      p→eframe ← (int)(++cur_el * frame_spread + 0.5) − 1;
      if (p→tag_value ≥ 0) printf("'%s'(%d,%d)␣", tag_strings[p→tag_value], p→sframe, p→eframe);
      p ← p→next;
      elc++;
      if (elc ≥ 6) {
        printf("\n␣␣");
        elc ← 0;
      }
    }
    if (¬elc) printf("\n");
  }
```

14. This function inserts a new node into the list with *tag_value* ← *value* after the node pointed to by *p*. One frame is stolen from each of the left and right sibling if either exists and has a frame to share.

```
⟨ Function Definitions 6 ⟩ +≡
  insert_node(struct tag *p, int value)
  {
    struct tag *tmp;
    tmp ← (struct tag *) malloc(sizeof(struct tag));
    tmp→tag_value ← value;
    tmp→sframe ← tmp→eframe ← −1;
    tmp→next ← p→next;    /* Set new node's next from left node's next */
    tmp→prev ← p;    /* Set the new node's prev to the left node */
    p→next ← tmp;    /* Set left node's next to the new node */
    if (tmp→next ≠ Λ) tmp→next→prev ← tmp;    /* Set right node's prev to new node */
    if (tmp→next ≠ Λ ∧ tmp→next→sframe < tmp→next→eframe) {
      tmp→eframe ← tmp→next→sframe;    /* Steal frame from right node */
      tmp→next→sframe++;
    }
    if (tmp→prev→tag_value ≠ −1 ∧ tmp→prev→eframe > tmp→prev→sframe) {
      tmp→sframe ← tmp→prev→eframe;    /* Steal frame from left node */
      tmp→prev→eframe−−;
    }
    if (tmp→sframe ≡ −1 ∧ tmp→eframe ≠ −1) tmp→sframe ← tmp→eframe;
    else if (tmp→eframe ≡ −1 ∧ tmp→sframe ≠ −1) tmp→eframe ← tmp→sframe;
  }
```

15. Delete the node pointed to by *p*. In addition to updating the pointers to the left and right of the deleted node *p*, any frames under the control of *p* must be split between the left and right nodes. Handle the special case of when left or right siblings don't exist. There should always be a *prev* value since the header node may not be deleted, so to check whether or not a valid *prev* node exists, check for *tag_value* ≠ −1.

```
⟨ Function Definitions 6 ⟩ +≡
  struct tag *delete_node(struct tag *p)
  {
    struct tag *tmp;
    if (p→next ≠ Λ) tmp ← p→next;    /* return the next node on the list of available */
    else tmp ← p→prev;    /* otherwise return the prev node */
    p→prev→next ← p→next;    /* re-route left's next to p's right */
    if (p→prev→tag_value ≠ −1) {
      if (p→next ≠ Λ)    /* Is there a node to the right? */
        p→prev→eframe ← (p→sframe + p→eframe)/2;    /* Split p's frames */
      else p→prev→eframe ← p→eframe;    /* left node gets all p's frames */
    }
    if (p→next ≠ Λ) {    /* Update right side */
      p→next→prev ← p→prev;    /* re-route right's prev to p's left */
      if (p→prev→tag_value ≠ −1)    /* Is there a node to the left? */
        p→next→sframe ← (p→sframe + p→eframe)/2 + 1;    /* Split p's frames */
      else p→next→sframe ← p→sframe;    /* right node gets all p's frames */
    }
    free(p);    /* De-allocate the node */
    return tmp;
  }
```

16. Function *get_ave*( ) accumulates values for an average from each bin of p.

⟨ Function Definitions 6 ⟩ +≡

```
get_ave(double *f, double *avearr, int length)
{
  int i;
  for (i ← 0; i < length; i++) avearr[i] += f[i];
}
```

17. Calculate the average and standard deviation values.

⟨ Function Definitions 6 ⟩ +≡

```
calc_norm(oneframe *f, double *min, double *max, double *ave, int length, int frames)
{
  int i, j;
  for (j ← 0; j < length; j++) ave[j] /= (float) frames;
  for (i ← 0; i < frames; i++)
    for (j ← 0; j < length; j++) min[j] += (f[i][j] − ave[j]) * (f[i][j] − ave[j]);
  for (j ← 0; j < length; j++) {
    max[j] ← ave[j] + 3.0 * sqrt(min[j]/(double)(frames − 1));
    min[j] ← ave[j] − sqrt(min[j]/(double)(frames − 1));
  }
}
```

18. There's not a whole lot to reading the input file. It consists of 2-byte signed integer values which will be converted to floating point at reading. The required storage space is allocated by this function and returned to the calling routine. Also, the number of samples is returned in *samples*.

⟨ Function Definitions 6 ⟩ +≡

```
short *read_input_file(char *filename, int *samples)
{
  FILE *inf;
  short *darr;
  int ret;
  if ((inf ← fopen(filename, "r")) ≡ Λ) {
    fprintf(stderr, "Yikes!␣␣Can't␣open␣input␣file␣\"%s\".␣␣What␣gives?\n", filename);
    exit(1);
  }
  fseek(inf, 0, SEEK_END);      /* determine file size */
  *samples ← ftell(inf)/(sizeof(short));
  fseek(inf, 0, SEEK_SET);
  darr ← (short *) malloc(sizeof(short) * *samples);
  ret ← fread(darr, sizeof(short), *samples, inf);
  if (ret < *samples) {
    fprintf(stderr, "Oops:␣Input␣file␣ran␣short␣of␣expected␣samples␣%d␣at␣%d\n", *samples, ret);
    *samples ← ret;
  }
  fclose(inf);
  return darr;
}
```

19. Read the tags from an existing annotation file.

```
⟨ Function Definitions 6 ⟩ +≡
  parse_from_file(char *filename)
  {
    FILE *annof;
    int ph_val, frames, framesize, i, j;
    struct tag *p;
    char the_line[128];
    p ← taglist ← (struct tag *) malloc(sizeof(struct tag));    /* create header */
    taglist→next ← taglist→prev ← Λ;
    taglist→tag_value ← −1;
    taglist→sframe ← taglist→eframe ← 0;
    if ((annof ← fopen(filename, "r")) ≡ Λ) {
      fprintf(stderr, "Could␣not␣open␣annotation␣file␣\"%s\"\n", filename);
      exit(1);
    }
    fgets(the_line, sizeof (the_line), annof);
    if (strncmp(the_line, "framesize␣=", 11) ≠ 0) {
      fprintf(stderr, "Annotation␣file␣missing␣framesize\n");
      exit(1);
    }
    framesize ← atoi(the_line + 11);
    fgets(the_line, sizeof (the_line), annof);
    if (strncmp(the_line, "nframes␣=", 9) ≠ 0) {
      fprintf(stderr, "Annotation␣file␣missing␣nframes\n");
      exit(1);
    }
    frames ← atoi(the_line + 9);
    for (i ← 0; i < frames; i++) {
      fgets(the_line, sizeof (the_line), annof);
      ph_val ← atoi(the_line);
      insert_node(p, ph_val);    /* insert new node after p */
      p ← p→next;    /* Advance pointer to newly added node */
      p→sframe ← p→eframe ← i;
      for (j ← 0; j < framesize; j++) fgets(the_line, sizeof (the_line), annof);
    }
    fclose(annof);
  }
```

20. Read a normalization file. No fancy format, just one value each of *min*, *max*, and *ave* per line.

⟨ Function Definitions 6 ⟩ +≡

```
input_norm(char *filename, double *min, double *max, double *ave, int n)
{
  FILE *norm;
  char the_line[128];
  int i;
  if ((norm ← fopen(filename, "r")) ≡ Λ) {
    fprintf(stderr, "Couldn't␣get␣normalization␣file␣\"%s\"\n", filename);
    exit(1);
  }
  for (i ← 0; i < n; i++) {
    fgets(the_line, sizeof (the_line), norm);
    if (feof(norm)) {
      fprintf(stderr, "Normalization␣file␣too␣short\n");
      exit(2);
    }
    sscanf(the_line, "%lg␣%lg␣%lg", &(min[i]), &(max[i]), &(ave[i]));
  }
  fclose(norm);
}
```

21. Write a normalization file. One value each of *min*, *max*, and *ave* per line.

⟨ Function Definitions 6 ⟩ +≡

```
output_norm(char *filename, double *min, double *max, double *ave, int n)
{
  FILE *norm;
  char the_line[128];
  int i;
  if ((norm ← fopen(filename, "w")) ≡ Λ) {
    fprintf(stderr, "Couldn't␣write␣normalization␣file␣\"%s\"\n", filename);
    exit(1);
  }
  for (i ← 0; i < n; i++) fprintf(norm, "%lg␣%lg␣%lg\n", min[i], max[i], ave[i]);
  fclose(norm);
}
```

22. Setup the normalization arrays. This need only be done once, and only if these values are not being read from a file.

⟨ Function Definitions 6 ⟩ +≡

```
init_norm(double *minarr, double *maxarr, double *avearr, int length)
{
  int i;
  for (i ← 0; i < length; i++) {
    minarr[i] ← 0.0;
    maxarr[i] ← −999999.0;
    avearr[i] ← 0.0;
  }
}
```

23. Perform spectral analysis. The `WINDOW` definition describes the Welch data-windowing function. Variable m is the number of frequency bins, requiring $2m$ samples for an FFT. The final power p for each frequency bin is squashed by $\log(1+p)$ by means of the *log1p*( ) function.

```
#define WINDOW(j) (1.0 − ((((j − 1) − facm) ∗ facp) ∗ (((j − 1) − facm) ∗ facp)))

⟨ Function Definitions 6 ⟩ +≡
  spctrm(short ∗arr, int nextval, double ∗f, double ∗w1, int m)
  {
    int j, j2, mm, m4, m44, m43;
    float facm, facp, den;
    float tempw, sumw, w;

    w1 −−;
    f −−;    /∗ This algorithm was written for FORTRAN, so rather than subtract 1 from each array
        index, I'll just bias the pointers. ∗/
    mm ← m + m;    /∗ 2m ∗/
    m4 ← mm + mm;    /∗ 4m ∗/
    m44 ← m4 + 4;    /∗ reference for folding negative frequencies ∗/
    m43 ← m4 + 3;
    for (j ← 1; j ≤ m; j++) f[j] ← 0.0;    /∗ Initialize the output array ∗/
    den ← 0.0;
    facm ← m − 0.5;
    facp ← 1.0/(m + 0.5);
    sumw ← 0.0;    /∗ Sum the weights for the selected windowing function. ∗/
    for (j ← 1; j ≤ mm; j++) {
      tempw ← WINDOW(j);
      sumw ← sumw + tempw ∗ tempw;
    }
    for (j ← 1; j ≤ mm; j++) {
        /∗ Fill the work array with the real-valued function, zeroing the imaginary part. ∗/
      j2 ← j + j;
      w1[j2 − 1] ← (float) arr[nextval++] ∗ WINDOW(j);    /∗ apply the windowing function ∗/
      w1[j2] ← 0.0;
    }
    w1++;    /∗ Do the FFT on the data, but first reset the pointer to its non-biased value. ∗/
    four1(w1, mm);
    w1 −−;
    f[1] ← (w1[1] ∗ w1[1]) + (w1[2] ∗ w1[2]);    /∗ Determine the power for this segment ∗/
    for (j ← 2; j ≤ m; j++) {
      j2 ← j + j;
      f[j] ← (w1[j2] ∗ w1[j2]) + (w1[j2 − 1] ∗ w1[j2 − 1]) + (w1[m44 − j2] ∗ w1[m44 − j2]) + (w1[m43 −
          j2] ∗ w1[m43 − j2]);
    }
    den ← m4 ∗ sumw;
    for (j ← 1; j ≤ m; j++)    /∗ Scale the power distribution ∗/
      f[j] ← (float) log1p((double) f[j]/den);
  }
```

24. The *four1* ( ) procedure is taken from *Numerical Recipes*. *four1* ( ) returns in place the complex Fourier transform of the *nn* complex-valued data points contained in the array *data*.

```
⟨ Function Definitions 6 ⟩ +≡
  four1(double *data, int nn)
  {
    double wr, wi, wpr, wpi, wtemp, theta, tempr, tempi, ts;
    int m, n, i, j, mmax, istep;
    data--;    /* Offset to compensate for FORTRAN */
    n ← 2 * nn;
    j ← 1;
    for (i ← 1; i ≤ n; i += 2) {
      if (j > i) {
        tempr ← data[j];
        tempi ← data[j + 1];
        data[j] ← data[i];
        data[j + 1] ← data[i + 1];
        data[i] ← tempr;
        data[i + 1] ← tempi;
      }
      m ← n/2;
      while ((m ≥ 2) ∧ (j > m)) {
        j ← j − m;
        m ← m/2;
      }
      j ← j + m;
    }
    mmax ← 2;
    while (n > mmax) {
      istep ← 2 * mmax;
      theta ← (double)(2.0 * M_PI)/(double) mmax;
      ts ← sin((double) 0.5 * theta);
      wpr ← −2.0 * ts * ts;
      wpi ← sin(theta);
      wr ← 1.0;
      wi ← 0.0;
      for (m ← 1; m ≤ mmax; m += 2) {
        for (i ← m; i ≤ n; i += istep) {
          j ← i + mmax;
          tempr ← wr * data[j] − wi * data[j + 1];
          tempi ← wr * data[j + 1] + wi * data[j];
          data[j] ← data[i] − tempr;
          data[j + 1] ← data[i + 1] − tempi;
          data[i] ← data[i] + tempr;
          data[i + 1] ← data[i + 1] + tempi;
        }
        wtemp ← wr;
        wr ← wr * wpr − wi * wpi + wr;
        wi ← wi * wpr + wtemp * wpi + wi;
      }
      mmax ← istep;
    }
  }
```

25. User Interaction. Here I will start building the user interface for selecting the phonemes to go with frames. This task is inherently auditory, and so I must be able to play the samples through a sound device.

Each frame represents `FFT_LENGTH` samples of a speech signal. If a number of frames are to represent a phoneme, then I must be able to stretch or shrink the frames selected. I may also need to add or delete phonemes from time to time, or change the phoneme represented. Here are the keystroke functions.

KEY	TASK
`-`	Shorten frame selection from the right side
`+`	Lengthen frame selection on the right side
`[SPC]`	Mark current selection and move right
`[BKSPC]`	Mark current selection and move left
`A`	Insert a phoneme to the right of the current
`D`	Delete the current phoneme
`C`	Change the current phoneme
`p`	Play the current selection
`W`	Write the annotated frame file (no exit)
`E`	Exit, but query to save changes

26. The first step is to open and initialize the device `/dev/audio`. I will need access to some more definitions in system files.

```
⟨ Include Files 2 ⟩ +≡
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/audioio.h>
```

27. I will also need a file descriptor for the audio device. I'll also declare the audio-specific I/O structure for initializing the device.

```
⟨ Global Definitions 4 ⟩ +≡
  int audio_dev;
  audio_info_t a_info;
```

28. Open and initialize the audio device. The `AUDIO_INITINFO` macro initializes an *audio_info_t* structure so that no configuration changes are made to the device on a subsequent *ioctl*( ). Once initialized in this way, small changes to the device initialization may be made easily.

```
#define AUDIO_DEVICE "/dev/audio"
⟨ Function Definitions 6 ⟩ +≡
  audio_init( )
  {
    int err;
    audio_dev ← open(AUDIO_DEVICE, O_RDWR);
    if (audio_dev < 0) {
      fprintf(stderr, "Couldn't␣open␣audio␣device␣%s\n", AUDIO_DEVICE);
      exit(3);
    }
    AUDIO_INITINFO(&a_info);
    a_info.play.sample_rate ← SAMPLE_RATE;
    a_info.play.channels ← 1;    /* Mono mode */
    a_info.play.precision ← PRECISION;
    a_info.play.encoding ← AUDIO_ENCODING_LINEAR;
    err ← ioctl(audio_dev, AUDIO_SETINFO, &a_info);
    if (err < 0) {
      fprintf(stderr, "Error␣in␣initializing␣audio␣device\n");
      exit(3);
    }
  }
```

29. Play the samples represented by the frames selected in the phoneme structure pointed to by p. Each frame represents `FFT_LENGTH` samples. An entire buffer is written at once to the audio device. Ya gotta *love* UNIX!

```
⟨ Function Definitions 6 ⟩ +≡
  play_frames(struct tag *p)
  {
    int nsamples, first;
    if (p ≡ Λ) {
      fprintf(stderr, "Illegal␣phoneme␣structure␣pointer.\n");
      return;
    }
    if (p→sframe ≡ −1 ∨ p→eframe ≡ −1) {
      fprintf(stderr, "␣␣Phoneme␣%s,␣Start␣frame␣=␣%d,␣End␣frame␣=␣%d\n", tag_strings[p→tag_value],
          p→sframe, p→eframe);
      return;
    }
    nsamples ← (p→eframe − p→sframe + 1) * FFT_LENGTH;    /* number of samples */
    first ← p→sframe * FFT_LENGTH;    /* index of first sample */
    write(audio_dev, &(arr[first]), nsamples * sizeof(short));
  }
```

30. And now, here's the editing loop. The editing process is driven by key presses. As such, I will need uninhibited access to the input stream. The loop is simple: the current position within the phoneme chain is displayed and then the process waits for the next keystroke. The action specified by the keystroke is taken, and the current position in the phoneme chain is re-displayed. All this ends when the exit key is pressed.

```
⟨Function Definitions 6⟩ +≡
  editing_loop(int nframes)
  {
    struct tag *p ← taglist→next;
    char the_key;
    int last_was_write;
    setup_screen_io();
  ReLoop: do {
      display_position(p);
      the_key ← getch();
      switch (the_key) {
      case 'E': break;     /* Exit activity */
      case '-': reduce_right(p);
        last_was_write ← 0;
        play_frames(p);
        break;
      case '+': extend_right(p);
        last_was_write ← 0;
        play_frames(p);
        break;
      case 'p': play_frames(p);
        break;
      case '␣':
        if (p→next ≠ Λ) {
          p ← p→next;
          play_frames(p);
        }
        else beep();
        break;
      case '\b':
        if (p→prev→tag_value ≠ −1) {
          p ← p→prev;
          play_frames(p);
        }
        else beep();
        break;
      case 'A': insert_node(p, 0);
        last_was_write ← 0;
        p ← p→next;
        break;
      case 'D': p ← delete_node(p);
        last_was_write ← 0;
        break;
      case 'C': modify_phoneme(p);
        last_was_write ← 0;
        break;
      case '?': case 'h': key_usage();
        break;
```

```
      case 'W': last_was_write ← 1;
        write_frame_file(nframes);
        break;
      default: beep();
      }
    } while (the_key ≠ 'E');
    move(15, 4);
    addstr("Sure␣(y/n)?␣");
    refresh();
    the_key ← getch();
    if ((the_key ≠ 'y') ∧ (the_key ≠ 'Y')) {
      move(15, 4);
      clrtoeol();
      goto ReLoop;
    }
    if (¬last_was_write) {
      move(15, 4);
      clrtoeol();
      move(15, 4);
      addstr("Last␣command␣was␣not␣(W)rite.␣␣Save␣(y/n)?␣");
      beep();
      refresh();
      the_key ← getch();
      if (the_key ≡ 'y' ∨ the_key ≡ 'Y') write_frame_file(nframes);
    }
    restore_screen_io();
  }
```

31. I'll be using the `curses` library to do screen I/O. More header files comming up.

⟨ Include Files 2 ⟩ +≡

```
#include <curses.h>
```

32. Define functions for `curses` initialization and terminal state restoration.

⟨ Function Definitions 6 ⟩ +≡

```
  setup_screen_io()
  {
    initscr();      /* Initialize curses */
    cbreak();       /* Enable unbuffered character retrieval */
    noecho();       /* Individual characters are not echoed */
  }
  restore_screen_io()
  {
    clear();
    refresh();
    endwin();       /* End curses */
  }
  key_usage()
  { }
```

33. Write the annotated frames to the output file.

```
⟨ Function Definitions 6 ⟩ +≡
  write_frame_file(int n)
  {
    FILE *outf;
    struct tag *p ← taglist→next;
    int i, j;
    double val;
    if (parm_phrase) {
      move(15, 4); addstr("Writing␣frame␣file..."); refresh();
    }
    if ((outf ← fopen(outfilename, "w")) ≡ Λ) {
      if (parm_phrase) {
        move(15, 4); clrtoeol();
        addstr("Couldn't␣open␣output␣file␣\"");
        addstr(outfilename); addstr("\""); refresh();
      }
      else {
        fprintf(stderr, "Couldn't␣open␣output␣file␣\"%s\"\n", outfilename);
        exit(2);
      }
      return;
    }
    fprintf(outf, "framesize␣=␣%d\nnframes␣=␣%d\n", FRAMESIZE, n);
    while (p ≠ Λ) {
      for (i ← p→sframe; i ≤ p→eframe; i++)
        if (i > −1) {
          fprintf(outf, "%d\n", p→tag_value);
          for (j ← 0; j < FRAMESIZE; j++) {
            val ← (allframes[i][j] − channel_min[j])/(channel_max[j] − channel_min[j]);
            if (val < 0.0) val ← 0.0;
            else if (val > 1.0) val ← 1.0;
            fprintf(outf, "%lg\n", val);
          }
        }
      p ← p→next;
    }
    fclose(outf);
    if (parm_phrase) {
      move(15, 4); addstr("Writing␣frame␣file...done!"); refresh();
      sleep(1);
      move(15, 4); clrtoeol(); refresh();
    }
  }
```

34. To change a phoneme in the current pointer, just ask the user for the new value and search for it in the *tag_strings* list.

```
⟨ Function Definitions 6 ⟩ +≡
  modify_phoneme(struct tag *p)
  {
    char new_phoneme[128];
    int new_ph;
    move(12, 2); addstr("Enter␣new␣phoneme␣string:␣"); refresh();
    echo(); getstr(new_phoneme); noecho();
    new_ph ← phoneme_value(new_phoneme);
    if (new_ph ≥ 0) p→tag_value ← new_ph;
    else {
      move(14, 4); addstr("Phoneme␣unrecognized!"); beep(); refresh();
      sleep(1);
    }
    move(12, 0); clrtobot(); refresh();
  }
```

35. Extend the current selection of frames to the right. Recursively steal frames from other nodes further right if none are available to the immediate right.

```
⟨ Function Definitions 6 ⟩ +≡
  extend_right(struct tag *p)
  {
    if (p ≡ Λ ∨ p→next ≡ Λ) {
      beep();
      return;
    }
    if (p→next→sframe ≡ p→next→eframe) extend_right(p→next);
    if (p→next→sframe < p→next→eframe) {
      p→eframe++;
      p→next→sframe++;
    }
  }
```

36. Reduce the frame selection from the right, putting frames into the next node. Don't remove the last frame.

```
⟨ Function Definitions 6 ⟩ +≡
  reduce_right(struct tag *p)
  {
    if (p ≡ Λ ∨ p→next ≡ Λ) {
      beep();
      return;
    }
    if (p→sframe ≡ p→eframe) {
      beep();
      return;
    }
    p→eframe−−;
    p→next→sframe−−;
  }
```

37. This function displays the current position in the phoneme chain given by p with surrounding context, including the number of frames and samples.

```
#define CONTEXT 5

⟨ Function Definitions 6 ⟩ +≡
  display_position(struct tag *p)
  {
    struct tag *np ← p;
    int colpos[CONTEXT], i, plines, curcol;
    char conv[128], *s1, *s2;
    for (i ← 0; i < CONTEXT; i++) colpos[i] ← (int)(((float)(COLS − 8)/(float) CONTEXT) * i) + 4;
    move(2, 2); addstr("Phrase:␣");
    plines ← 3; curcol ← 4;
    move(plines, curcol);
    s2 ← the_phrase;
    while (*s2 ≠ '\0') {
      s1 ← conv;
      while (*s2 ≠ '␣' ∧ *s2 ≠ '\t' ∧ *s2 ≠ '\n' ∧ *s2 ≠ '\0') *s1++ ← *s2++;
      *s1 ← '\0';
      if (curcol + strlen(conv) ≥ COLS − 4) {
        move(++plines, 4);
        curcol ← 4;
      }
      addstr(conv); addch('␣');
      while (*s2 ≡ '␣' ∨ *s2 ≡ '\t' ∨ *s2 ≡ '\n') *s2++;
      curcol += strlen(conv) + 1;
    }
    plines += 3;
    for (i ← plines; i ≤ plines + 3; i++) {
      move(i, 0); clrtoeol();
    }
    i ← (int)(CONTEXT/2);
    while (i−− ∧ np→prev→tag_value ≠ −1) np ← np→prev;
    if (np ≠ Λ) {
      for (i ← 0; i < CONTEXT; i++) {
        move(plines, colpos[i]); addch('"'); addstr(tag_strings[np→tag_value]); addch('"');
        sprintf(conv, "%d", np→sframe);
        move(plines + 1, colpos[i]); addstr("Start:␣"); addstr(conv);
        sprintf(conv, "%d", np→eframe);
        move(plines + 2, colpos[i]); addstr("End:␣"); addstr(conv);
        if (np ≡ p) {
          move(plines + 3, colpos[i]); addstr("=======");
        }
        np ← np→next;
        if (np ≡ Λ) break;
      }
    }
    refresh();
  }
```

38. Index. Underlined section numbers indicate definitions.

⟨ Function Definitions 6, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 28, 29, 30, 32, 33, 34, 35, 36, 37 ⟩ Used in section 1.

⟨ Global Definitions 4, 5, 8, 10, 27 ⟩ Used in section 1.

⟨ Include Files 2, 26, 31 ⟩ Used in section 1.

⟨ Main Program 7 ⟩ Used in section 1.

ANNOTATE