

Labelling Guitar Chords

Jason Brazile

The Problem

A few years ago, I began to teach myself how to play the guitar. There were particular songs that I wanted to play for which I was not able to find accurate transcriptions. I began listening to recordings, attempting to figure out the notes and fingerings that were used in order to write them down. After being asked a few times for copies of my transcriptions, I came up with a way to represent them as ASCII text files. (See Appendix A - Representing Guitar Chords in ASCII).

Shortly after that, it became a very tedious and error prone process figuring out what the pitches were from my guitar fingerings, and from those, attempting to determine the chord names. I started referring to a book that cataloged thousands of guitar chords in my attempt to do this faster and more accurately. Eventually, I decided it was time to write a program.

The **Guitar Fingering Labeller** or **gfl** is a program that reads ASCII based guitar fingerings, determines the pitches and from that the chords represented by them. It produces a new file with the chords labelled and the fingerings formatted nicely into rows and columns.

Example

As an example, here are the hastily typed chords used in the introduction to a song I recently transcribed:

```
+-----+ +-----+   +-----+ +-----+   +-----+
| | | | | | | | | | |   | | o | o | 6 | | o o o | 5   | | o | o | 4
+-----+ +-----+   +-----+ +-----+   +-----+
| | o | | | | | | | | |   | | | | o | | | | | | | |   | | | o | | |
+-----+ +-----+   +-----+ +-----+   +-----+
o | | | o | | | | | o | |   | | | | | | | o | | | | | | |   | | | | | | |
+-----+ +-----+   +-----+ +-----+   +-----+
| | | o | | | | | | | | |   | | | | | | | | | | | | | | | |   | | | | | | o
+-----+ +-----+   +-----+ +-----+   +-----+
| | | | | | | | | | |   | | | | | | | | | | | | | | | |   | | | | | | o
+-----+ +-----+   +-----+ +-----+   +-----+
                                0 x                                0                                0
```

I ran this through **gfl**:

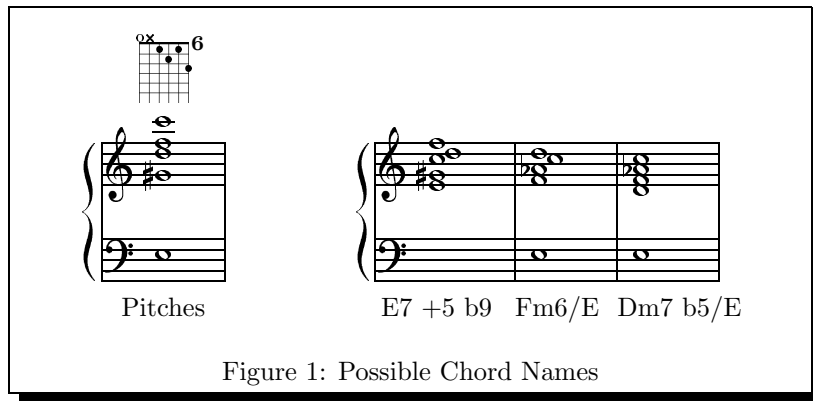
```
$ gfl meditacao.tab
```

which produced this output:

```
G6          Bm7          E7 +5 b9          Am9          B7 b9/A
+-----+ +-----+ +-----+ +-----+ +-----+
| | o | | | | 2 | | o | o | o 2 | | o | o | 6 | | o o o | 5 | | o | o | 4
+-----+ +-----+ +-----+ +-----+ +-----+
o | | | o | | | | | o | | | | | | | | | | | | | | | | | | | o | | |
+-----+ +-----+ +-----+ +-----+ +-----+
| | | o | | | | | | | | | | | | | | | | | | | | | | | | | | | |
+-----+ +-----+ +-----+ +-----+ +-----+
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
+-----+ +-----+ +-----+ +-----+ +-----+
x           0   x   x   0 x           x 0           x 0
```

The chords were labelled, the fingerings were shifted to the top of each grid adjusting the fret number as needed, and all unused strings were explicitly marked as not used.

Most of the time, **gfl** labels the chords correctly. But some chords can be labelled one of several different ways depending on the key the song is currently in and how the chord functions harmonically with respect to the chords that surround it. To give an example, let's look at the middle chord above and make use of the **gfl** "analyze" flag:



```
$ gfl -a meditacao.tab
```

The “analyze” flag prints out each chord in a single column and precedes it with a “comment block” that documents the pitches and octaves determined by each string in the chord followed by an ordered list of the guesses `gfl` has made for the chord name. The output is formatted this way so that a user could choose an alternate name by “uncommenting” that name and running `gfl` again with the “keep names” flag.

```
# pitches: [E(4),-1,G#(5),D(6),F(6),C(7)]
#
# E7 +5 b9
# Fm6/E
# Dm7 b5/E
#
# E7 +5 b9
+++++
| | o | | 6
+++++
| | | o | |
+++++
| | | | | o
+++++
| | | | | |
+++++
0 x
```

The information in the output above is illustrated in Figure 1. The pitches are the same even though they appear not to be. They are merely rearranged in different octaves and/or renamed inharmonically (e.g. $G\#$ is the same as $A\flat$) in order to support the naming.

In this case, `gfl` came up with 3 possible names for the given chord, and it determined that `E7 +5 b9` is the most likely. As it turns out, this is the correct guess, because the chord is harmonically serving as V/ii (“five of two”) to the immediately following `Am9` chord (see Appendix C - Basic Harmony).

As previously mentioned, `gfl` usually guesses correctly, but this is really due to luck in the heuristic it uses to rank its guesses, because in fact `gfl` isn’t currently intelligent enough to make use of harmonic context (i.e. it doesn’t “know” that this chord is serving as V/ii) while making these guesses.

Other Features

Re-Shaping

A secondary function of `gfl` is to re-format the guitar chords into rows and columns using the (possibly newly determined) names. For example, you could specify that you only wanted 3 chords per row instead of 5, and the program would reshape the example above like this:

```

      G6              Bm7              E7 +5 b9
+-----+          +-----+          +-----+
| | o | | | 2    | o | o | o 2    | | o | o | 6
+-----+          +-----+          +-----+
o | | | o |      | | | | o |      | | | o | |
+-----+          +-----+          +-----+
| | | o | |      | | | | | |      | | | | | o
+-----+          +-----+          +-----+
| | | | | |      | | | | | |      | | | | | |
+-----+          +-----+          +-----+
| | | | | |      | | | | | |      | | | | | |
+-----+          +-----+          +-----+
x      0          x  x              0 x

      Am9              B7 b9/A
+-----+          +-----+
| | o o o | 5    | | o | o | 4
+-----+          +-----+
| | | | | |      | | | o | |
+-----+          +-----+
| | | | | o      | | | | | |
+-----+          +-----+
| | | | | |      | | | | | o
+-----+          +-----+
x 0              x 0

```

This ability to reformat is very useful when you want to add or remove a chord. You can merely add the new chord to the next line or extend an existing column by one chord and `gfl` can reshape and readjust everything.

Other Instruments/Tunings

The most unusual feature of `gfl` is that you can specify the tuning and number of strings that are used when doing chord analysis. This is specified in a command line argument as a colon separated list of MIDI note numbers corresponding to the tunings of the strings you are interested in (see [DS89]). Labelling guitar transcriptions based on fingerings in a non-standard tuning should be no problem. And oddly enough, you should be able to label and format mandolin fingerings like this:

```
$ gfl -t 55:62:69:76 sonho_meu.tab
```

This corresponds to the standard mandolin tuning - G, D, A, E. For another example, I believe ukulele tuning is A, D, F#, B which you should be able to handle like this:

```
$ gfl -t 69:74:78:83 tiptoe.tab
```

What it Doesn't Do

The feature that I am most often asked about is being able to type in the chord names for a song and have `gfl` return guitar fingerings. If there were only one possible way to finger any particular chord, this would be easy to do. However, if there were 2 possible ways to finger each particular chord, then for a song with N chords, there would be 2^N different ways to finger it that `gfl` would have to consider.

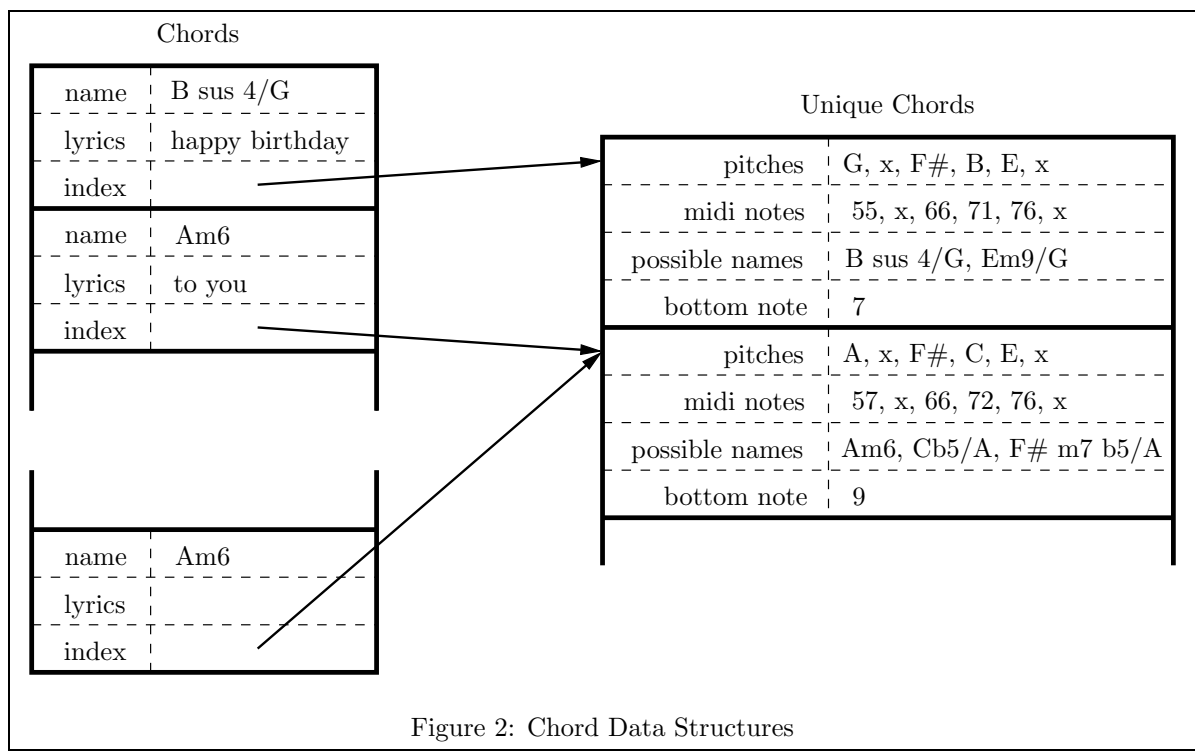
Obviously, this would be not be practical for most songs. But to compound the problem, instead of 2 different ways to finger any particular chord, there are probably an average of about 20. This puts the problem in the realm of Artificial Intelligence).

But again assuming that `gfl` could actually try each possible combination looking for an “optimal” list of fingerings, one would have to come up with an evaluation function that determined why one chord choice is better than another in a given progression. There are many factors that could be taken into consideration in such a decision involving both musical issues (e.g. voicings) and physical issues (e.g. your hand can only span so many frets). These choices would have to be decided subjectively.

How it Works

The processing of `gfl` can be broken down into the following stages:

Input Parsing The input is parsed into 4 data structures - the header text, the trailing text, a collection of the unique chords encountered, and a chronologically ordered list of all the chords (which references the



unique chords as well as containing chord specific information such as lyrics and “kept” names - i.e. for those times when the user disagrees with `gf1`’s best guess. See Figure 2 and Appendix B - Representing Complex Data Structures in Perl)

Determination of Pitches For each of the unique chords, each pitch name and octave is determined and stored in a human readable format. This is used when the “analyze” option has been selected or for debugging purposes.

Determination of Possible Chord Names Each of the unique chords is permuted and processed into a canonical form in order to be compared against a table of known chord forms in an attempt to make matches. Each match is given a “score” based on a simple heuristic. All matches for a given chord are sorted by that score and added to a list of possible names for that chord.

Output Formatting When it comes time to produce output, text is printed to a buffer in a tabular format which allows for the reshaping of the text into rows and columns.

This corresponds to a the main perl subroutine as shown in Figure 3.

Input Parsing

I began publishing my transcriptions on the web and my intent was to have `gf1` analyze the HTML versions of transcriptions without having to do manual editing or other pre/post processing. Because the guitar chord parts of my transcriptions are simple ASCII text displayed verbatim, I was able to consider an input file to have the following form:

1. Header text (optional)
2. Guitar Chords
3. Trailer text (optional)

In my case, the header text and trailer text contain HTML, but `gf1` neither knows nor cares.

The transition from the “header text” section to the “guitar chords” section is determined by the initial appearance of the ASCII fingering grids which may have been preceded by one line of chord names which is otherwise impossible to differentiate from header text. For this reason, a line of lookahead is needed (see [AU77]).

```

sub main
{
    ...

    $lines = &lexical_scan($filename);

    ($header, $trailer, $chords, $uniq_chords) = &parse($lines);

    if ($reshape_only){
        $keep_names = 1;
    }else{
        $pitch_tab = &init_pitches;
        $uniq_chords = &add_pitches($pitch_tab, $uniq_chords);
        $chord_tab = &init_chords;
        $uniq_chords = &add_chord_names($pitch_tab, $chord_tab, $uniq_chords);
    }

    &write_out($outfile, $header, $trailer, $chords, $uniq_chords);
}

```

Figure 3: Main Program Logic

The termination of the “guitar chords” section is determined by encountering at least 3 blank lines. Whatever comes after that is considered “trailer text”.

Parsing the “guitar chords” section is slightly more challenging because I was guided by the following principles in the representation of the chords:

- The grids should be able to represent any number of strings
- There can be any number of grids per line.
- The grids may optionally be preceded by a line containing chord names.
- The grids must be followed by open/dampened string usage indicators.
- A grid may optionally have lyrics underneath it that must not extend beyond the width of that grid, but should be associated with that chord.

These principles led to the design of a finite state machine (see [AU77]) that can parse guitar chord input based on these goals (see Figure 4).

Refer to Figure 5 to see how this state machine could be implemented in perl.

The result of the parsing stage is 4 data structures - one for the header text, 2 for the guitar chords, and one for the trailer text. The reason there are 2 data structures to maintain information for the guitar chords, is that some chords will be duplicated in the song and we would like to only analyze them once. However, even though the pitches for the chord may duplicated, there may be other information associated with the chord (e.g. lyrics) that require individual chord information to be kept. The individual chord information references the shared chord information so there is no duplication.

Determination of Pitches

The “determination of pitches” phase is done in order to print pitch information in human readable format. The actual pitches are stored internally as MIDI note numbers (for example, the E on the low string is represented by the number 52). Pitch names and octaves are easily determined by divide and modulo operations as there are 12 different pitches in an octave.

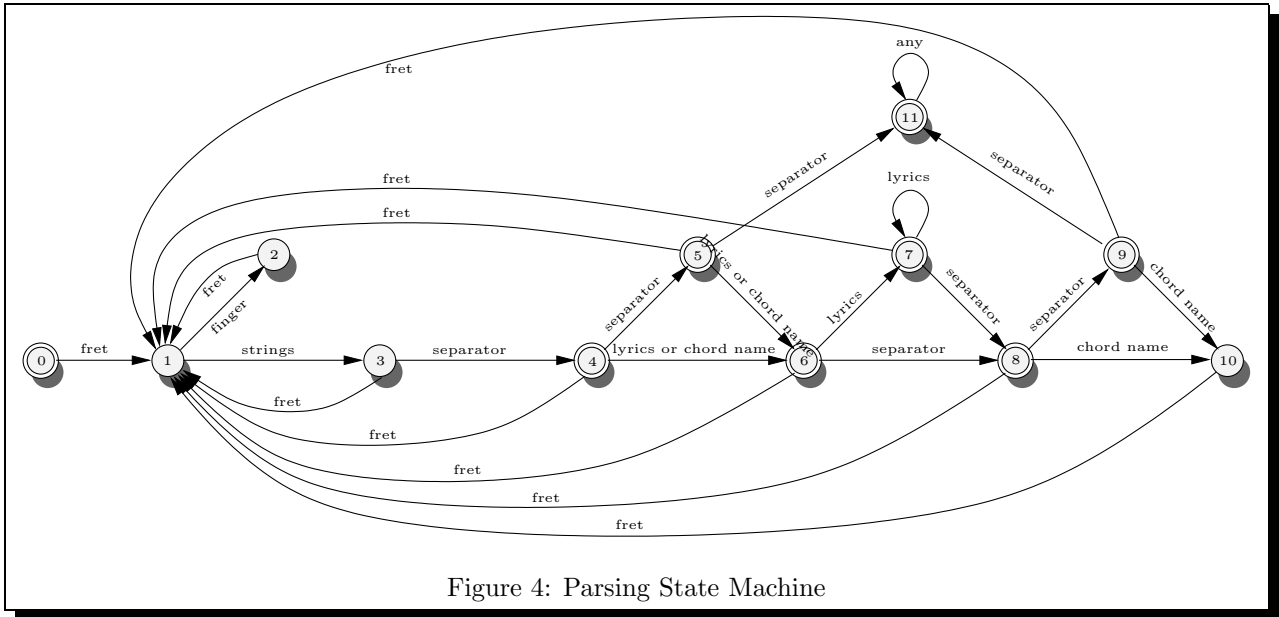


Figure 4: Parsing State Machine

```

$state = 0;

foreach $line (@$lines){
  if ($state == 0){
    if ($line->{'type'} eq 'fret'){
      ...
      $state = 1;
    }else{
      ...
    }
  }elseif ($state == 1){
    if ($line->{'type'} eq 'finger'){
      ...
      $state = 2;
    }elseif ($line->{'type'} eq 'strings'){
      ...
      $state = 3;
    }else{
      ERROR
    }
  }elseif ($state == 2){
    if ($line->{'type'} eq 'fret'){
      $state = 1;
    }else{
      ERROR
    }
  }
  .
  .
  .
}

```

Figure 5: Implementing the Parsing State Machine

```

sub chord_lookup
{
    ...

    ($bottom, $the_rest, $combined) = &notes_only($chord);

    $canon = &chord_canon($bottom % 12, $combined);
    foreach $c (@$chord_tab){
        if (&chord_compare($canon, $c->{'cnotes'})){
            RECORD THIS MATCH ALONG WITH A HIGH RANK
        }
    }

    foreach $note (@$the_rest){
        $canon = &chord_canon($note % 12, $the_rest);
        foreach $c (@$chord_tab){
            if (&chord_compare($canon, $c->{'cnotes'})){
                RECORD THIS MATCH ALONG WITH A LOWER RANK
            }
        }
    }

    foreach $note (@$the_rest){
        $canon = &chord_canon($note % 12, $combined);
        foreach $c (@$chord_tab){
            if (&chord_compare($canon, $c->{'cnotes'})){
                RECORD THIS MATCH ALONG WITH A LOWER RANK
            }
        }
    }

    SORT THE MATCHES BY RANK
    return($sorted_matches);
}

```

Figure 6: Determining Possible Names

Determination of Possible Chord Names

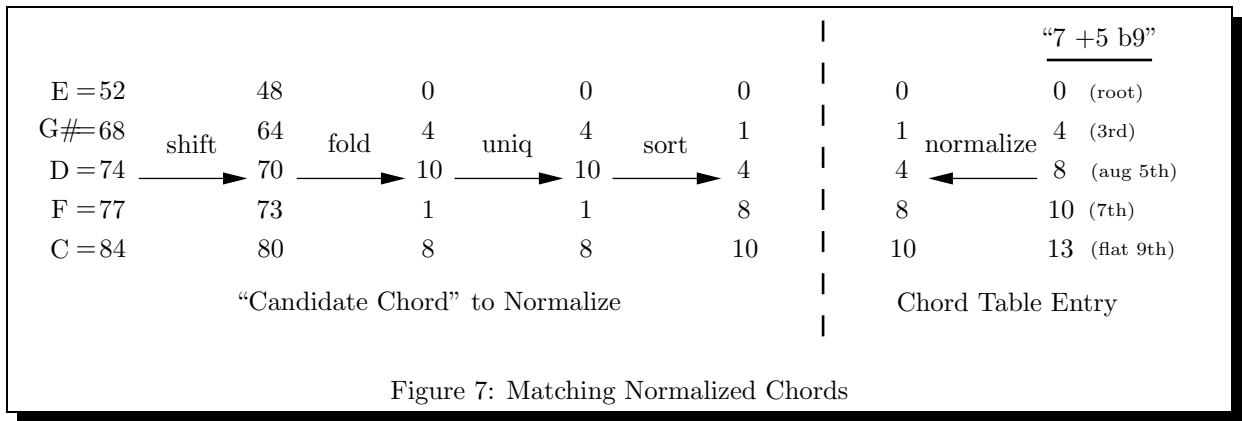
The determination of chord names is where all the difficult work of `gfl` is performed. Each unique chord in the song has to be permuted, transformed into a canonical form, and then compared against a table of known chord forms.

Chords are named as a combination of their root pitch and their quality. A “C major 7” chord (CM7) has “C” as its root, and “major 7” as its quality. Most of the time, the root of the chord appears on the bottom (i.e. it is the lowest note played) but there are times when a chord is in inversion (a chord tone other than the root is on the bottom), or there may be a note on the bottom that seems to not even be a chord tone. In these cases, you end up with what’s often referred to as a “slash chord”.

A “slash chord” is named as a combination of its root pitch, its chord quality, and a slash followed by the bass note. For example, a C major chord in first inversion (i.e. E, C, G, as opposed to root position: C, E, G) could be labelled as the slash chord C/E.

The possibility of slash chords adds slight complexity to the issue of determining the chord name.

The first thing that is done is to separate the pitches into 3 groups: the *bottom* (a group of one), the *rest* (all but the bottom), and a *combined* group which includes all pitches. Then, the determination of chord names is attempted in a 3 part permutation phase described below. For each of the parts, each generated permutation goes through the same process. First, the permutation or “candidate chord”, is put into a canonical form (which



is described below), then it is compared against every chord in a table of known chord forms. When a match is found, it is added to a list of matches and a score is assigned to that match roughly indicating how “good” this match is. All of the possible names for the permutations generated by a particular chord are then sorted by their scores and kept for later use.

Permutations

The permutation phase is coded as shown in Figure 6.

The first part of the permutation phase consists of attempting the single permutation generated by choosing the *bottom* pitch as root and using the *combined* group as the pool of remaining chord pitches. This covers the common case where the bass note is the root of the chord. Matches based on this permutation are given a high rank.

The second part of the permutation phase involves generating all permutations where each note in the *rest* group is tried as the root of the chord, and only the notes in the *rest* group are used as the pool of remaining chord pitches. This covers slash chords where the bass note is not a chord tone.

The third part of the permutation phase involves generating all permutations where each note in the *rest* group is tried as the root of the chord, but the notes in the *combined* group are used as the pool of remaining chord pitches. This covers slash chords where the bass note is a chord tone (i.e. the chord is in inversion).

The heuristic used in ranking the matches is based on ordering the known chord table from “simpler” (e.g. minor or *m*) to “more complex” (e.g. diminished 7, flat 13 or *dim 7 b13*). In addition, matches that are made during the second or third part of the permutation phase are artificially ranked lower than those made during the first part. In effect, this leads to *gfl* preferring labels where the bass note is the root of the chord and “simpler” chord namings over “more complex” chord namings.

A Canonical Form

A canonical form is generated from chord pitches as shown in Figure 7. The value of the proposed root (modulo 12) is subtracted from all the other chord pitches. This “shift” causes each pitch to represent its relative interval away from the root and assumes the root is now pitch 0. Then each pitch is octave folded so that there are no relative intervals greater than 11. Next, any duplicate pitches are removed. And finally, the remaining relative intervals are sorted. Note that there are other possible canonical forms, and even for this form these steps don’t have to be performed in this order.

The table of known chord forms represents pitches in a more user-friendly “relative interval” format which is normalized once upon startup. For chords that can appear with missing pitches (e.g. it is sometimes common to leave out the 5th), that chord form must be listed explicitly as its own entry.

Output Formatting

The output file is written in the following manner. First the header text is written as is. Then the chords are written out according to whichever mode of processing was specified, and finally the trailer text is written as is.

If the `analyze` option was selected, then the chords are written in a single column where each chord is preceded by a “comment” section that gives pitch names and all possible names for that chord as determined by `gf1`.

Otherwise, the chords are written out using either the name `gf1` determined to have the highest score, or if the `keep_names` option was invoked, the name that was read in during input. If the program couldn't determine a name and no name was given, then the name is left blank. When this happens, the user would probably want to rerun `gf1` using the `analyze` option to see which pitches are being specified and either determine there was a data entry error (which is most often the case), or that a new entry needs to be added to the table of known chord forms.

Future Work

There are several features I would like `gf1` to have. My first choice would be the ability to generate postscript or something similar for prettier and more compact printout. The important thing is to not allow page breaks to occur in the middle of fingering grids, which is what currently happens with the ASCII-based grids.

Guitar tablature enthusiasts would like it if `gf1` were made to understand, label, and analyze tablature rather than just block chords. This would certainly be useful for more melodic guitar pieces as opposed to the mostly harmonic and rhythmic songs that I play.

However, what I would really like to have is a program that helps analyze chord progressions at a higher level and can even suggest possible substitutions in order to help spruce up boring progressions (see [Lav91] and [Ber95]). Perhaps some descendant of `gf1` will have that ability some day.

Conclusion

I started work on this program a few years ago in C, before I knew Perl. After modifying data structure handling and memory management support routines yet again because of design changes in my data structures for the 3rd of 4th time, I stopped altogether due to the tediousness of it all. When I later decided to try again using Perl, I was very pleased to be able to make several interim changes to my data structures in a mostly painless way. Many languages like scheme or tcl could have probably fulfilled this need, but quick and dirty regular expressions and easily composable data structure primitives for lists and associative arrays helped me quickly get a working prototype. Seeing an early prototype actually do something useful, provided the motivation I needed to take the time to redo a few things and come up with a mostly complete tool. The quick modify/test turnaround that Perl provides by combining its compile and run phases greatly speeds up the development process.

For examples of real songs processed by `gf1` see:

<http://www.etc.ch/~jason/tabs.html>

Appendix A - Representing guitar chords in ASCII

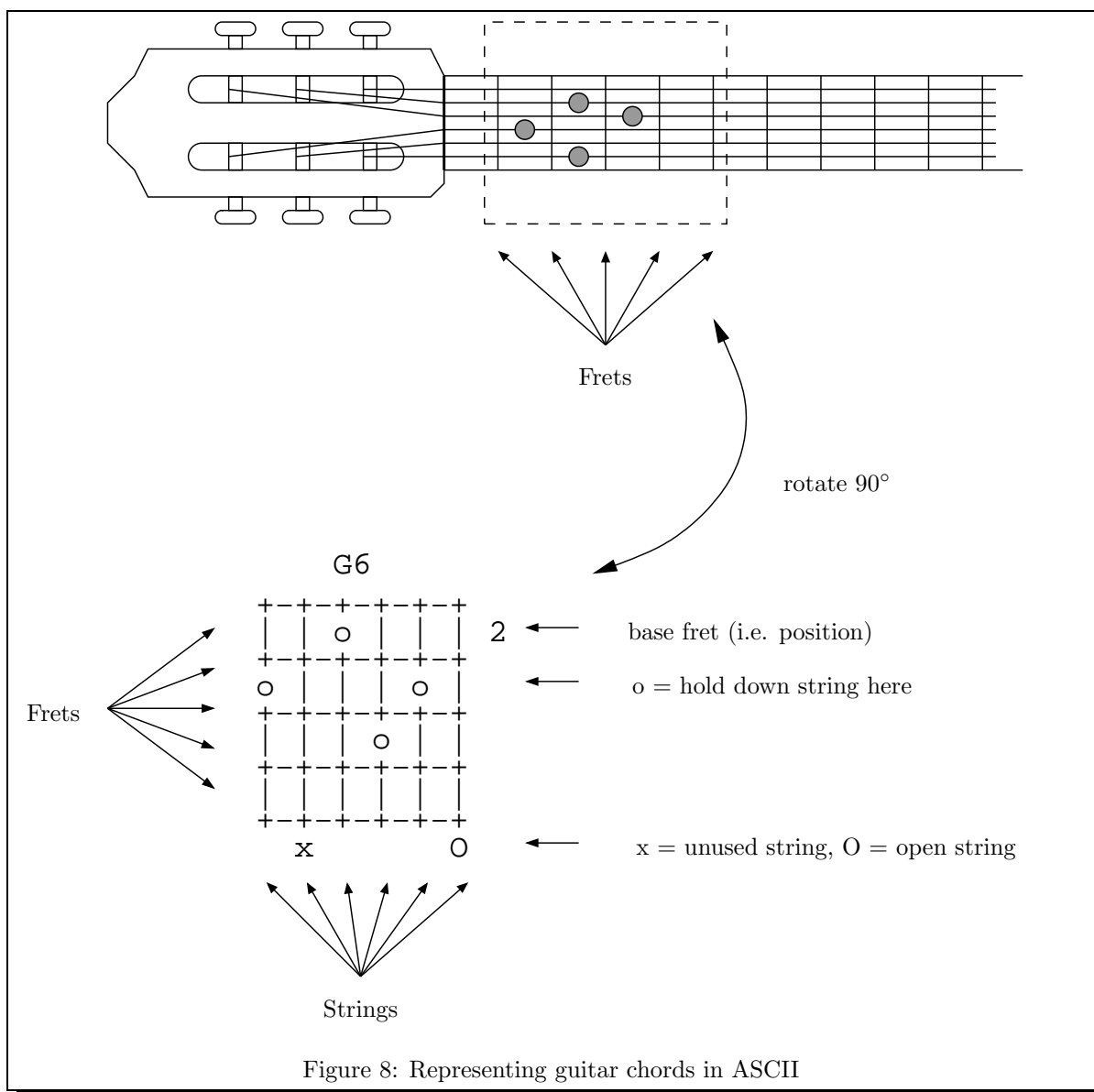


Figure 8: Representing guitar chords in ASCII

Appendix B - Representing Complex Data Structures in Perl

A multi-layer data structure was presented in Figure 2 which needed to be implemented in perl. There are many ways to implement such a data structure, but usually one of the better ways to implement arbitrarily complex data structures is to use perl references to compose the perl list and associative array primitives.

There are simple rules one can use to help decide which of the primitive types are suitable for which pieces of your data structure. If your collection needs to be ordered, you will want to use arrays. If you want fast keyword lookup of objects, you will probably want to use associative arrays, etc.

Another choice that has to be made in composing these data structures is whether or not to use references to objects or the objects themselves. Often references are better because they promote less copying, therefore being more efficient. Also, references are more easily passed to other subroutines.

For the chord data structures used in `gf1`, perl lists and associative arrays were composed using perl references. The chord list needed to be ordered chronologically, which suggested the outermost layer needed to be an array. However, each chord had attributes which we wanted to be able to access by name rather than implicitly by index, so each array element was a hash table of attributes (rather than being yet another array).

So in order to iterate through the chord list printing out all the interesting information there is for each chord, we would access our data structures as shown in Figure 9. You may want to refer back to Figure 2 to remind you how the data structures are logically shaped.

```
foreach $chord (@chords){
    $chord_info = $uniq_chords->[$chord->{'chord_index'}]

    print $chord->{'name'}, "\n";
    print join(",", @{$chord->{'lyrics'}}), "\n";
    print join(",", @{$chord_info->{'pitches'}}), "\n";
    print join(",", @{$chord_info->{'midi_notes'}}), "\n";
    print join(",", @{$chord_info->{'possible_names'}}), "\n";
    print $chord_info->{'bottom_note'}, "\n";
}
```

Figure 9: Accessing Chord Data Structures

Appendix C - Basic Harmony

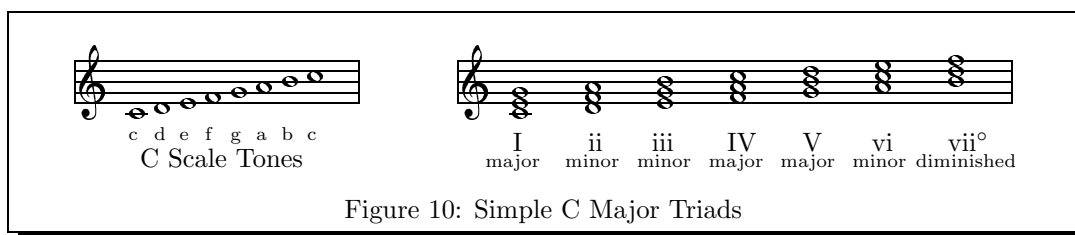


Figure 10: Simple C Major Triads

Given a key, say C Major, one can construct a chord based on each scale tone in that key (i.e. C, D, E, F, G, A, B), merely by starting with the scale tone, and then skipping every other scale tone. If you limit the number of notes in the chord to 3, you get triads (see Fig 10). While it appears that all the listed triads are symmetric, they in fact are not. This makes sense when you look at a keyboard and notice that the pattern of white and black keys is irregular “*wbwbwbwbwbw*” and that C major scale tones only refer to the white keys. For example, there are 3 chromatic pitches (i.e. either scale or non-scale pitches) in between the notes C and E in the first triad listed (i.e. C#, D, and D#). However, there are only 2 chromatic pitches in between the notes D and F in the second triad listed (i.e. D#, and E - because E# is the same pitch as F).

These asymmetries lead to the chords being labelled differently. If the number of pitches between the 3 notes in a triad are 4 followed by 3, this is labelled a *major* triad. If the number of pitches between the 3 notes in a triad are 3 followed by 4, this is labelled a *minor* triad. For completeness, (3, 3) is labelled a *diminished* triad, and (4, 4) is labelled an *augmented* triad.

It turns out that nearly all songs can be harmonized (albeit simply) using only the triads constructed from the scale tones. In fact, a great many songs can be harmonized using only 3 of the constructed triads - I, IV, and V (it is no coincidence that these 3 triads cover all the scale tones). For example, here is a simple harmonization for the Happy Birthday song:

```
I             V
Happy Birthday to You
V             I
Happy Birthday to You
I             IV
Happy Birthday dear Pathologically Eclectic Rubbish Lister
IV  I       V I
Happy Birthday to You
```

If instead of limiting your chords to triads (i.e. 3 notes each), you constructed chords of 4 notes using the same technique (skipping every other scale tone), you would end up with what are known as the diatonic 7th chords - which often give more interesting harmonies.

Just as we did with triads, you can also classify 7th chords based on the relative distances between each of the 4 notes (See Figure 11).

Even richer harmonies can be obtained by using 5 or more notes, especially if you don't limit yourself to using only scale tones. Learning how to construct these more complex chords in a way that makes them sound good when played in a series is what music theory is all about (see [Ott84] and [Hea80]).

distance from root	Δ	pitch	name
(0,3,6,9)	(3,3,3)	C, Eb, Gb, Bbb	C diminished 7 (dim7)
(0,3,6,10)	(3,3,4)	C, Eb, Gb, Bb	C half diminished 7 (m7 b5)
(0,3,7,10)	(3,4,3)	C, Eb, G, Bb	C minor 7 (m7)
(0,3,7,11)	(3,4,4)	C, Eb, G, B	C minor #7 (m #7)
(0,4,7,10)	(4,3,3)	C, E, G, Bb	C dominant 7 (7)
(0,4,7,11)	(4,3,4)	C, E, G, B	C major 7 (M7)
(0,4,8,10)	(4,4,3)	C, E, G#, B	C augmented 7 (7 +5)
(0,4,8,12)	(4,4,4)	C, E, G#, B#	C whole tone 7? (+7 +5)
Common Chords with distances including 5 or 2			
(0,5,7,10)	(5,2,3)	C, F, G, Bb	C 7 suspended 4 (7 sus 4)
(0,4,6,10)	(4,2,4)	C, E, Gb, Bb	C 7 flat 5 (7 b5)

Figure 11: Classification of 7th chords

Appendix D - A Complete Song

Perl Happy Birthday Song
(traditional)

arranged by
Jason Brazile

Bsus 4/G	Am6	D9	G6	GM7
+++++ o 3	+++++ o 4	+++++ o 4	+++++ o 2	+++++ o o 3
+++++ o o	+++++ o o o	+++++ o o o	+++++ o o	+++++ o o
+++++ o	+++++ 	+++++ 	+++++ o	+++++
+++++ 	+++++ 	+++++ 	+++++ 	+++++
+++++ x x	+++++ x x	+++++ x x	+++++ x x	+++++ x x
Happy Birthday to	you Happy	Birthday to	you Happy	Birthday Pathologic- -ally Eclectic
A# dim7 b13	A# dim7	Am7	Adim7	G69
+++++ o	+++++ o	+++++ o o o o 5	+++++ o o 4	+++++ o 3
+++++ o o	+++++ o	+++++ 	+++++ o o	+++++ o
+++++ 	+++++ 	+++++ 	+++++ 	+++++ o o o
+++++ 	+++++ 	+++++ 	+++++ 	+++++
+++++ x x 0	+++++ x x 0 0	+++++ x x	+++++ x x	+++++ x
Rubbish	Lister Happy	Birthday	to	you

This arrangement Copyright(c) 1998 by Jason Brazile

References

[AU77] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. AddisonWesley, Reading, Massachusetts, 1977. ISBN: 0-201-00022-9.

[Ber95] Arnie Berle. *Understanding Chord Progressions for Guitar*. Amsco Publications, New York, NY, 1995. ISBN: 0-8256-1488-0.

- [DS89] Steve DeFuria and Joe Scacciaferro. *MIDI Programmer's Handbook*. M&T Books, Redwood City, California, 1989. ISBN: 1-55851-068-0.
- [Hea80] Dan Hearle. *The Jazz Language - A Theory Text for Jazz Composition and Improvisation*. CPP/Belwin Inc., Miami, Florida, 1980.
- [Lav91] Andy Laverne. *Handbook of Chord Substitutions*. Ekay Music Inc, Katonah, NY, 1991. ISBN: 0-943-48-51-8.
- [Ott84] Robert W. Ottman. *Advanced Harmony*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984. ISBN: 0-13-011370-0.