1 Introduction

My programming language, Arpeggify, solves the problem of automatically arpeggiating chord progressions (ie. successively playing the notes in a piece of music's chords). Having this language first can assist musicians with composition and allows them to experiment with and immediately hear different chord progressions. Rather than having to write out sheet music, potential composers could specify chord names and immediately hear what they would sound like together, without the need for any instruments. Second, this language could help musicians practice. After encoding a piece of music as a program, musicians could listen to and observe the piece's harmony and practice along to it. While there are many existing digital audio workstations (DAW's) these are often very expensive and require special equipment like MIDI controllers to function properly. Further, while there is sheet music notation software like Finale and Sibelius, these pieces of software again are expensive and require knowledge of music notation in order to hear audio playback. Arpeggify is built to be simple, cross-platform, and to provide immediate audio output without without any special equipment or additional hardware or software.

This problem needs its own programming language because there are infinitely many pieces of music, each defined by a grammar. Pieces of music can be broken down into phrases, chords, and rhythms, and the individual notes within these chords, and I think there is natural parallel between the structure of music and the structure of a program. Solving this problem with a programming language allows programmers to follow a set of rules to generate and infinite number of programs, all while exploring the idea of a piece of music being used as a computer program.

2 Design Principles

The guiding principles are the parallels between music compositions and programs. I've focused particularly on lead sheets for jazz tunes, which provide visual representations of the harmonic and rhythmic structures of the tune. Encoded in this sheet music is information about which chords go where, which types of chords are played, in addition to a feel for the overall structure of the piece. Many jazz compositions are in the form of an ”A” section repeated twice, followed by a ”B” section then by an ”A” section again, and Arpeggify allows us to specify chords and rhythms within phrases and then combine then to form larger pieces of music. Further, in my implementation of arpeggiation, I have made the language be easily extendable and for more chord types and features to easily be added. I've created a general interpretation for arpeggiating chords, which involves representing each note as a numeric value and then calculating the notes in a chord and their corresponding frequencies using formulas. This keeps large parts of my code generic, and allows for new types of chords to be defined and to work right away.

3 Example Programs

1) A simple 12-bar C blues progression. From within the 'lang' project directory, to run type:
'dotnet run ../examples/example-1.arp example-1.wav'
and an audio file 'example-1.wav' will be generated in the current directory.

```
Chords aChords = (C7, C7, C7, C7)
Chords bChords = (F7, F7, C7, C7)
Chords cChords = (G7, F7, C7, C7)

Rhythms rA = (4,4,4,4)
Rhythms rB = (4,4,4,4)
Rhythms rC = (4,4,4,4)

Phrase pA = (aChords,rA)
Phrase pB = (bChords,rB)
Phrase pC = (cChords,rC)

Tune blues = (pA,pB,pC,END)
```

2) The jazz standard ’Giant Steps.’ From within the ’lang’ project directory, to run type:
‘dotnet run ../examples/example-2.arp example-2.wav’
and a file ‘example-2.wav’ will be generated in the current directory.

```
Chords cOne = (BMa7, D7,  GMa7, Bb7, EbMa7, A-7, D7)
Rhythms rOne = (2,2,2,2,4,2,2)

Chords cTwo = (GMa7, Bb7, EbMa7, F#7, BMa7, F-7, Bb7)
Rhythms rTwo = (2,2,2,2,4,2,2)

Chords cThree = (EbMa7,  A-7, D7, GMa7, C#-7, F#7)
Rhythms rThree = (4,2,2,4,2,2)

Chords cFour = (BMa7, F-7, Bb7, EbMa7, C#-7, F#7)
Rhythms rFour = (4,2,2,4,2,2)

Tune giantSteps = ((cOne,rOne), (cTwo,rTwo),(cThree,rThree),(cFour, rFour), END)
```

3) The jazz standard ”I’ve Never Been in Love Before.” From within the ’lang’ project directory, to run type:
‘dotnet run ../examples/example-3.arp example-3.wav’
and a file ‘example-3.wav’ will be generated in the current directory.

```
Chords aChords = (BbMa7, G-7, C-7, F7, BbMa7, Eb7, D-7, G7, C-7, F7)
Rhythms rA = (2,2,2,2,2,2,2,2,4,4)

Chords bChords = (EbMa7, C-7, F7, BbMa7, A-7, D7, G-7,E7, A7, DMa7, C-7, F7)
Rhythms rB = (4,2,2,4,2,2,4,2,2,4,2,2)

Chords eOneChords = (BbMa7, C-7, F7)
Rhythms reOne = (4,2,2)

Chords eTwoChords = (BbMa7, F-7, Bb7)
Rhythms reTwo = (4,2,2)

Phrase A = (aChords,rA)
Phrase B = (bChords,rB)
Phrase eOne = (eOneChords,reOne)
Phrase eTwo = (eTwoChords,reTwo)

Tune neverBeen = (A, eOne, A, eTwo, B, A, eOne, END)
```

4 Language Concepts

To write Arpeggify programs a user needs to understand nomenclature for jazz chords and an understanding of how chords can be combined to build phrases, and, in turn, tunes. A programmer would need knowledge of note names and common chord extensions. Experimentation would of course be encouraged, but the they would also need a basic understanding of harmony in order for the program output to sound good. Programmers must know that every chord is built up of a root and extension, rhythms (or chord lengths) are build up simply of number, and phrases are built by combining a list of chords with an equal-length list of rhythms. Finally, they must understand how tunes are build by combining together phrases, allowing for repeats and easy encoding of tune’s with repeated sections.

5 Syntax

Formal grammar of Arpeggify:

```
<Expression>     ::= <Assignment>
                   | <Sequence>
<Sequence>       ::= <Expression> /newline <Expression>
<Assignment>     ::= <TypeName> <variable> <TuneBuilder>
<TypeName>       ::= "Tune"
                   | "Phrase"
                   | "Rhythms"
                   | "Chords"
<Variable>       ::= α ∈ {A, a, B, b, C, c, ...}+
<TuneBuilder>    ::= Tune
                   | Phrase
                   | Chords
                   | Rhythms
<Tune>           ::= (<TuneLiteral>)
                   | (<TuneVariable>)
<TuneLiteral>    ::= <Phrase>+ END
<TuneVariable>   ::= <variable>+ END
<Phrase>         ::= (<PhraseLiteral>)
                   | (<PhraseVariable>)
<PhraseLiteral> ::= <Chords> <Rhythms>
<PhraseVariable>::= <Variable> <Variable>
<Chords>         ::= (<Root><Extension> +)
<Root>           ::= <Note>
                   | <Accidental>
<Note>           ::= α in (A, B, C, D, E, F, G)
<Accidental>     ::= <Root> <Symbol>
<Symbol>         ::= #
                   | b
<Extension>      ::= Ma7
                   | -7
                   | 7
<Rhythms>        ::= (n ∈ {positive ints} +)
```

The language is built of tunes which are built of phrases which are build of combinations of chords and rhythms. Rhythms are just positive integers and chords and combinations of roots—a note or a note and an accidental—and extensions specifying the type of chord to be arpeggiated.

6 Semantics

1. **Sequence**

   Syntax:

   ```
   Phrase A = (aChords,rA)
   Phrase B = (bChords,rB)
   ```

   Abstract Syntax: Expr * Expr

   Meaning: Sequence evaluates two newline-separated expression, evaluating the first expression first and then the second expression next. The sequence operator returns the return value of evaluation of the second expression.

2. **Assignment** At a high level, our assignment operator binds the value on the right of the '=' operator with a variable name on the left of the '=' operator. Because assignment differs based on the type of data we are assigning to a variables, we break this down into four cases, some of which can be broken down even further

   (a) **Tune Assignment** Tune Assignment can be broken up into two cases, assigning phrase literals to a tune variable, or assigning phrase variables to a a tune variable

      i. TuneVariable Assignment
      Syntax:

      ```
      Tune neverBeen = (A, eOne, A, eTwo, B, A, eOne, END)
      ```

      Abstract Syntax:

      ```
      TuneBuilder * Map<String,TuneBuilder>
      ```

      Meaning: Each comma-separated variable corresponds to a phrase. The data structures associated with each of those phrases are looked up and combined and the result is constructed as a TuneBuilder of type Tune, bound to 'neverBeen' and returned.

      ii. TuneLiteral Assignment
      Syntax:

      ```
      Tune t = (((BMa7, D7,  GMa7, Bb7, EbMa7, A-7, D7), (2,2,2,2,4,2,2)), END)
      ```

      or

      ```
      Tune giantSteps = ((cOne,rOne), (cTwo,rTwo)), END)
      ```

      Abstract Syntax:

      ```
      TuneBuilder * Map<String,TuneBuilder>
      ```

      Meaning: Here, phrases are still combined into a tune, but the phrases are entered either as lists of chord and rhythm literals or as combinations of chords and rhythms variables. Here, first the paired chord and rhythm lists are combined into phrases and then the resulting phrases are combined to form a tune. Again, the result is bound to the variable on the left side of the '=' and the resulting TuneBuilder constructed as a Tune is returned.

   (b) **Phrase Assignment** Like Tune Assignment, Phrase assignment can also be broken up into two cases, depending on the right side of the assignment expression is made up of variable names or of literals.

      i. Phrase Variable Assignment
      Syntax:

      ```
          Phrase pA = (aChords,rA)
      ```

      Abstract Syntax:

      ```
      TuneBuilder * Map<String,TuneBuilder>
      ```

      Meaning: Here, aChords corresponds to a Chords variable and rA a Rhythms variable. The results of each of these variables are retrieved, and if they are both the right types and of the same length, then they are combined to form a phrase which is bound to the variable name and returned as a TuneBuilder.

      ii. Phrase Literal Assignment
      Syntax:

      ```
      Phrase p = ((BMa7, D7,  GMa7, Bb7, EbMa7, A-7, D7), (2,2,2,2,4,2,2))
      ```

      Abstract Syntax:

```
TuneBuilder * Map<String,TuneBuilder>
```

Meaning: Much like as in Tune Variable assignment, Chords and Rhythms are combined to form a phrase. However, here, rather than retrieving values or data structures corresponding to variables, the chords and rhythms listed as literals are simply combined as is to form a phrase. The result is bound to the variable name and returned.

(c) **Chords Assignment**
Syntax:

```
Chords cChords = (G7, F7, C7, C7)
```

Abstract Syntax:

```
TuneBuilder * Map<String,TuneBuilder>
```

Meaning: Here, the comma-separated chords are assigned to the variable indicated on the left of the '=' and returned as a TuneBuilder.

(d) **Rhythms Assignment**
Syntax:

```
Rhythms rA = (4,4,4,4)
```

Abstract Syntax:

```
TuneBuilder * Map<String,TuneBuilder>
```

Meaning: Here, the comma-separated rhythms are assigned to the variable indicated on the left of the '=' and returned as a TuneBuilder.

3. **Data Types**

(a) TuneBuilder
A TuneBuilder is a general framework for constructing, Tunes, Phrases, Rhythms, and Chords,

i. Tune
Syntax:

```
(A, eOne, A, eTwo, B, A, eOne, END)
```

Abstract Syntax:

```
Phrase list or string list
```

Meaning: A combination of phrases which can be arpeggiated. Stored as a list of phrases or as a list of phrase variables.

ii. Phrase
Syntax:

```
(aChords,rA)
```

Abstract Syntax:

```
Chord list * Rhythm list or string * string
```

Meaning: A phrase made up of Chords and corresponding Rhythms. Stored as a tuple of a Chord list and Rhythm list or a tuple of strings representing Chords and Rhythms variables respectively.

iii. Rhythms

```
(2,2,2,2)
```

Abstract Syntax:

```
        Rhythm list
```

Meaaning: A list of integers, each one corresponding the length of a chord in beats

iv. Chords

```
(CMa7,D-,F7,G7)
```

Abstract Syntax:

```
        Chord list
```

Meaaning: A list of chords, each one indicating the root of a chord and the type of chord

(b) Primitives

i. Chord
Syntax: CMa7
Abstract Syntax:
Root * Extension
Meaning: A single chord composed of a root and an extension indicating which other notes are in the chord

ii. Extension
Syntax: Ma7, -7, or 7
Meaning: Which other notes to add to a chord when combined with a chord's root

iii. Root
Syntax: C or C#
Abstract Syntax: Note or Note * Accidental
Meaning: The root of a chord, a natural note or a note with a flat or sharp.

iv. Note
Syntax: A
Abstract Syntax:
One of the letters A-G
Meaning, the note represented by the given letter

v. Accidental
Syntax: C#
Abstract Syntax:
Root * Symbol
Meaning: A note modified by a half step in either the flat or sharp direction

vi. Symbol
Syntax: or b
Abstract Syntax:
Char '#' or Char 'b'
Meaning: a symbol that modifies a note's pitch by one half step

4. **Arpeggiation**

Arpeggiation is done entirely behind the scenes. The user simply must ensure that a tune assignment is the last statement in a program, and Arpeggify will automatically arpeggiate the tune. Arpeggiation takes a tune and evaluates it, resulting in a .wav file audio output.

7 Remaining Work

My next step is to implement support for different tempos. My goal is to have a standard tempo which can be overridden by an additional command line argument. This should be easy to implement, since when I arpeggiate chords I play one note per beat and we can convert beats per minute to quarter note length in second by dividing 6000 by our BPM value.