Will Grawthwal, Louis Sobel, Chase Lambert

ABCPlayer Design Proposal

There are three main components to our design of the ABCPlayer, a *lexer*, a *parser*, and a *scheduler*. Below are descriptions of the three components, followed by an explanation of what will be necessary to bring them all together.

Lexer

The lexer takes a file with multiple lines of text as its input and returns an ordered list of tokens. A token will be represented by the following datatype:
ABCToken:
- enum tokenType
- Some set of values, as described below
- Line number (for debugging purposes)

Here is a list of the different token types and their corresponding values that will be returned:

Lexeme	**value**
HEADER	<key,value>
VOICE	<voice name>
NOTE	<name,octave,duration>
ACCIDENTAL	<note,modifier>
REST	<duration>
STARTCHORD	Null
ENDCHORD	Null
STARTTUPLET	<notecount>
ENDTUPLET	Null
STARTSECTION	Null
STARTBAR	Null
STARTREPEAT	Null
MULTIENDING	<ending number>
ENDSECTION	Null
ENDBAR	Null
ENDREPEAT	Null
COMMENT	<comment>

Parser:

The parser takes a list of tokens as its input. The output it a tree-like data structure representing the piece.

The parser assumes that the input is in the correct format, which is enforced by the Lexer. For example, the X header is the first header. The Parser starts of reading the header tokens and puts them into the environment. A few important environment parameters are also calculated, like default notes per quarter and ticks per default note.

After the headers are parsed, the more complex tree-like structure is created. In going through tokens, the state of the parser is held on a stack. Whenever an END token is found, an object with the corresponding token is made and the top element of the stack is removed. This recursively happens until the entire tree is made.

Below is the class structure of the parser output. It also returns metainfo such as headers and essential timing info. Explanations of how the scheduler treats each type is described in the next section.

```
interface Schedulable:
        - accept(Visitor v)
        - add(Schedulable s)
class TuneSequence implements Schedulable:
        - contents: a list of Schedulables
class TuneParallel  implements Schedulable:
        - contents: an ArrayList of Schedulables
class TuneRepeatable extends TuneSequence:
        null
class Tuple extends TuneSequence:
        - multiplier: how much much to multiply the contents of each element in the tuple
class Chord extends TuneParallel:
        - contents: overrides TuneParallel because Chord can only contain primitives
class TunePrimitive implements Schedulable:
        - duration: duration of the primitive as a fraction of the default note
class Rest extends TunePrimitive:
        null
class Note extends TunePrimitive:
        - note: the MIDI ptich represented by this Note
```

Scheduler

The scheduler walks the syntax tree returned by the parser. It will be implemented using the visitor pattern. It maintains state variables:

- clock: the tick count that the scheduler is at
- multiplier - ticks per default duration (L defined in header)

And environment variables:

- the SequencePlayer for which it is scheduling (which never changes)

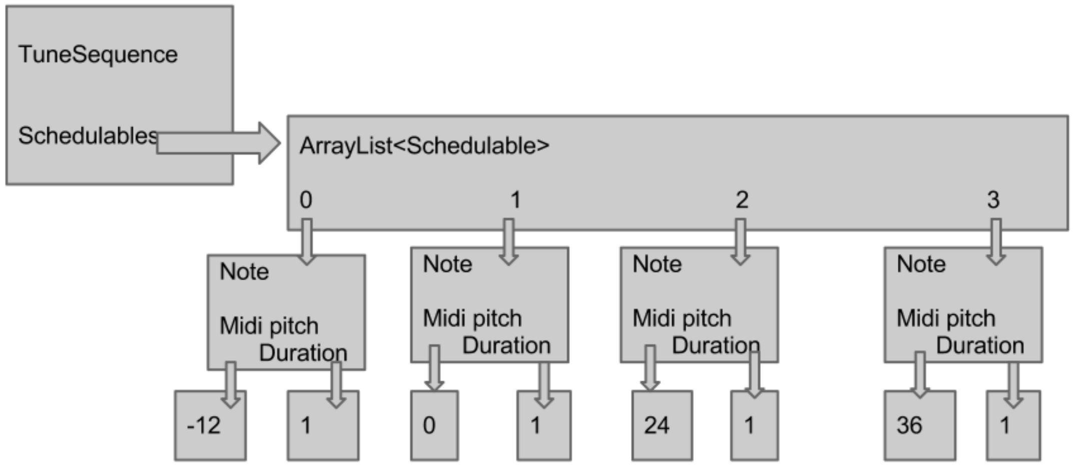
The Scheduler schedules the types as follows:

```
- TuneSequence t:
        for each Schedulable s in t.contents:
                s.accept(this)
```

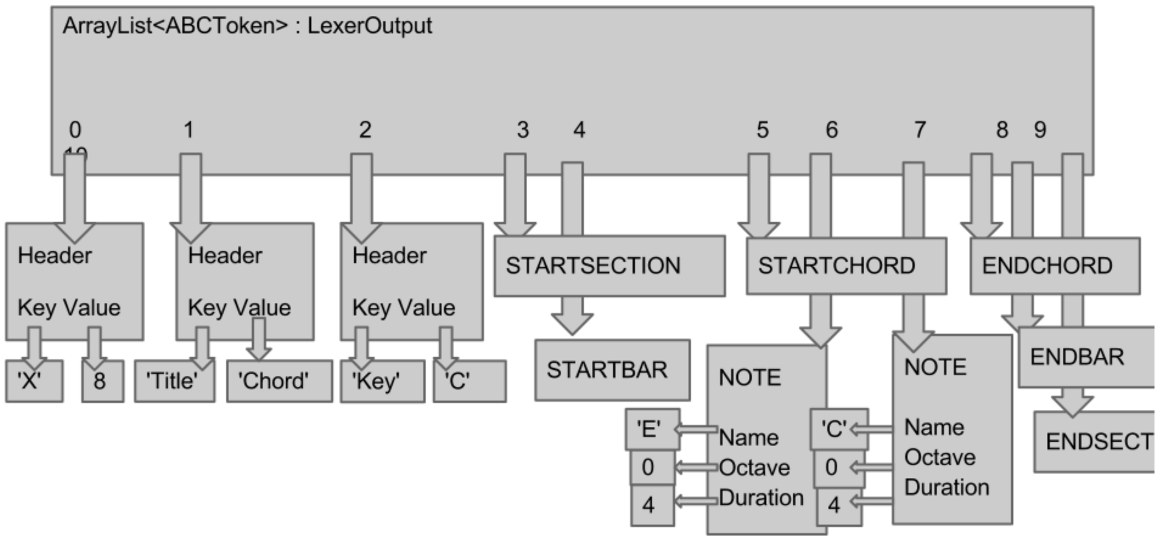
```
- TuneParrallel t:
    oldClock = this.clock
    for each Schedulable s in t.conents:
        this.clock = oldClock
        s.accept(this)
- TuneRepeatable t:
    if t.contents.size() == 1:
        t.contents[0].accept(this)
        t.contents[0].accept(this)
    else:
        for i = 1...t.contents.size()-1:
            t.contents[0].accept(this)
            t.contents[i].accept(this)
- Tuple t:
    oldMultiplier = this.multiplier
    this.multiplier = this.multiplier * t.multiplier
    for Schedulable s in t.contents:
        s.accept(this)
    this.multiplier = oldMultiplier

- Rest r:
    tick_duration = r.duration * this.multiplier
    this.SequencePlayer.schedule(REST,this.clock,this.clock + tick_duration)
    this.clock = this.clock + tick_duration
- Note n:
    tick_duration = n.duration * this.multiplier
    this.SequencePlayer.schedule(n.pitch,this.clock,tick_duration)
    this.clock = this.clock + tic
```

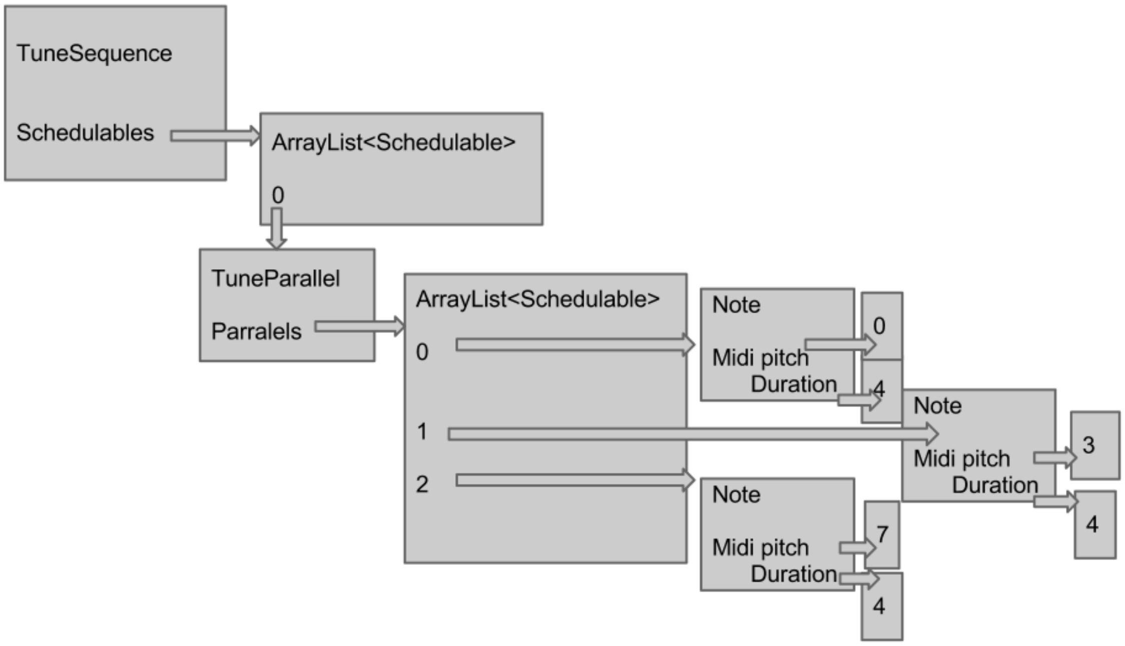
1). Sample1 - Parser Output



Sample2 - Lexer output



Sample3 - Parser Output


TuneSequence
Schedulables
ArrayList<Schedulable>
0
TuneParallel
Parralels
ArrayList<Schedulable>
0
1
2
Note
Midi pitch
Duration
0
4
Note
Midi pitch
Duration
3
4
Note
Midi pitch
Duration
7
4