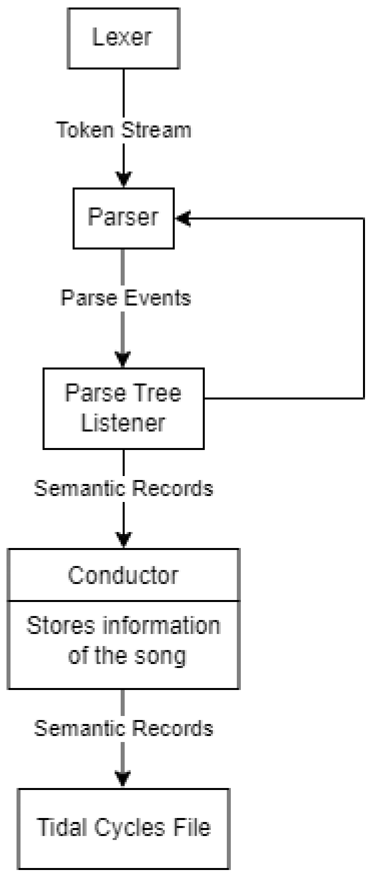
The Chordpiler

By Nate Romack and Jack Anderson

The translator is simply a tool that translates code from one language to another. Most of the time it is used in a compiler or an interpreter to translate high level programming languages to low level machine code for execution on a specific CPU architecture like Intel's x86 or Apple's ARM architectures. A compiler will output object code or assembly to a file to be linked and executed or translated to machine code by an assembler and then linked and executed. An Interpreter can execute high level code by recognizing and then immediately attempting to load, link, and execute instructions rather than compiling, linking, and then executing. There are many implementations that can be used to achieve this that will not be covered due to the scope of the paper.



Compilers are implemented using a couple components. The first component is the lexer, this component recognizes what is in the program. The lexer turns the input into a continuous stream of tokens to be fed into the parser. It does this by defining tokens with regular expressions and adding its token to the token stream when seen. After the input has been tokenized the compiler will pass that stream of tokens to the parser to be built into a data structure to represent the structure of the program. Most of the time it is built into a parse tree where leaves of the tree are tokens that represent data and the internal nodes are expression identifiers like function definitions, control statements, or mathematical operators. Once the tree is built the program is checked for semantic errors then traversed to generate an internal representation of the program. Finally the internal representation is passed to the code generator which builds the output file based on the internal representation. This component can sometimes include a code optimizer that will optimize the IR before returning the complete output. The flow of a compiler is reviewed by the graph. Now that the workings of a compiler have been explained we can talk about our Chordpiler.

We made this compiler because we wanted a way to translate the guitar tabs that we already know how to write music with, to code that will produce a chord loop with the program Tidal Cycles. Tidal Cycles is part of a system that interprets your code and sends the proper requests to an audio engine called Superdirt which synthesizes the requested sound.

To make our lexer and parser we used a tool called ANTLR which takes in a grammar file of lexer and parser rules that are defined using regular expressions and generates the components. Using ANTLRs parse tree walker we were able to send events to our SimpleSongBuilder which maintains information about the song and prints the code. The input contains the BPM, one or more chords, and the strums per chord above each chord specified. If the number of strums is left blank it will default to one. Because tidal works in cycles per second rather than beats per minute we had to do some math to convert the BPM by taking the bpm/60/4. First we divide by 60 to get the beats per second, then we divide by 4 to account for the beats per cycle. The output will be six audio channels and a list of notes on them, each channel represents one string of a guitar. The channels are an array of 6 linked lists and are iterated over using the string counter. Each time a new chord is entered the string counter is reset and the strums for that chord are written to a list. Each time that a new string is entered in the parser it sends an event, gets the note from the beginning of the string and gets its value in semitones. It also resets the fret counter and a boolean tracking if the string had a fingering. Every time a new position rule is entered it adds one to the fretCount. Whenever a position is entered it can either be barlines or a fingering. Fingerings are defined as ‘x’ to mark the fret being played or an ‘o’ on any fret to mark a rest for that string. Once a fingering is detected it sends an event and writes the current note plus the fretCount to the correct output channel if it is ‘x’, otherwise it prints a ‘~’ for rests. On string exit the stringCount is increased, If an exit string event is sent and isFound is false then it will write the current note value of the string, representing an open string being played. Currently the compiler only supports standard tuning where each string is tuned to E,A,D,G,B,e going from bottom to top starting at E2 and ending at e4. Once the parse tree is fully walked our main method calls our print function which takes the channels array and prints them by printing the note value the amount of times specified by the number of strums and slowing the track by the number of chords specified. This is to fit the loop into the proper time as tidal cycles divide notes into one cycle no matter how many notes are in the loop, in other words it does not add time for notes only divides it. Here is what the output looks like.

```
hush

setcps(120/60/4)

d1 $ slow 4 $ note "[4 ] [5 ] [7 ] [4 ] " # sound "superpiano"

d2 $ slow 4 $ note "[0 ] [0 ] [-1 ] [0 ] " # sound "superpiano"

d3 $ slow 4 $ note "[-5 ] [-3 ] [-5 ] [-5 ] " # sound "superpiano"

d4 $ slow 4 $ note "[-8 ] [-7 ] [-10 ] [-8 ] " # sound "superpiano"

d5 $ slow 4 $ note "[-12 ] [-12 ] [-13 ] [-12 ] " # sound
"superpiano"

d6 $ slow 4 $ note "[~ ] [-19 ] [-17 ] [~ ] " # sound "superpiano"
```
test1.tidal

Overall this project has been a huge learning experience for the both of us. We learned a lot about the use of event listeners and object oriented design. It has also expanded our view of what a compiler can do. If we had more time we would add more sounds that the user could specify rather than just using the “superpiano” sample. We would also allow them to change the time signature and tuning of the guitar.