```
%final writeup for algoirithms
\documentclass[12pt]{article}
\usepackage{graphicx, listings, fancyvrb}

\begin{document}

\title{Final Project: Make A Lisp}
\author{Nick Creel}
\date{May 5, 2019}

\maketitle

\section[Introduction]{What is Make-A-Lisp?}

Make-A-Lisp, also known as MAL, is an implementation of Lisp
created by Joel Martin as a learning tool for writing
programming languages. Anyone who is capable of following along
with Martin's detailed step-by-step guide can create their very
own lisp interpreter, a helpful thing for those of us trying to
learn how to create our own programming languages. Jim and I
stumbled upon MAL while searching for a suitable final project
for my Programming Languages tutorial.

\section{My Implementation}
\subsection{Picking a Language}

I chose to write my implementation of MAL in Javascript. I
didn't really decide based on any particular feature of
Javascript, but felt that I should probably write something in
Javascript to build proficiency in the language. In the end, a
lot of the common motifs of Javascript programming didn't come
in handy, so it felt a lot like I was writing python with more
curly brackets...and more confusing errors.

\subsection{step 0}

MAL is unusual in that it does not provide the programmer with
a EBNF describing the language. In a roundabout way, Martin
avoids the topic of an EBNF by describing the functions that
```

make up the lexer and the parser and in what order they should be called. For example, a function that parses lists must also analyze the atomic contents of the list. The final function called can be read as the terminal/atomic expression seen in an EBNF, which indicated when the left recursion of a context-free grammar should stop.

Step 0 of the implementation of MAL involves setting up a skeleton for the MAL REPL (read evaluate print loop). The basic structure is like so:

```
\begin{enumerate}
    \item Take user input from console as string
    \item pass user input to reader function (lexing and
parsing)
    \item once parsed, evaluate the code that the user provided
    \item print the result of the evaluation, or any error
messages
    \item repeat
\end{enumerate}
```

Setting up the skeleton only involved setting up a series of functions that looks something like this:

```
\begin{code}
    PRINT(EVAL(READ(user\_input)))
\end{code}
```

At this point, the functions did nothing but immediately return the input that the user provided, so the REPL initially behaved like an echo function.

```
\subsection{step 1}

\begin{wrapfigure}
  \includegraphics[width=\linewidth]{$HOME/Pictures/
harvestmoon/drawthefuckingowl.png}
```

\end{wrapfigure}

Traditionally, when creating a compiler, the first two essential pieces that a programmer codes are the lexer and the parser. A lexer is a function/group of functions that reads user input, matches to a regular expression, and identifies different tokens in the input that are defined for the language. The lexer may also do a bit of pre-parsing by determining the type of each atomic token.

Once the lexer generates tokens, it passes them on to the parser, which performs a process known as syntactic analysis, using the tokens as input and producing a syntax tree. A syntax tree forms based on the specification provided in the grammar of a language, recursing down through the user input until finding an atomic value.

In the Make-a-Lisp guide, Martin combines the role of lexer and parser into a "reader," presumably representing the R in REPL. The user input is matched to a provided Regular Expression that contains all of the tokens possible in MAL (tokenize()), then the matched tokens are immediately sent to another function (read_form()). read_form runs until all of the tokens have been seen, parsing lists and atoms (variables and numbers). When parsing a list, the read_list function repeatedly calls read_atom on the contents of the list, and read_atom assigns a type to each token based on whether or not the item matches as a number or a variable. Because all lisp functions are wrapped in parenthesis, they are recognized as lists at this step.

It is simple to combine the lexer and the parser with Lisp, because the way that Lisp is written immediately reflects its syntax. For example, to add two numbers, a lisp programmer would write something like

\begin{code}
(+ 1 2)
\end{code}

and the syntax tree would look like

```
\begin{lstlisting}
          +
         /  \
        /    \
       /      \
  Int:1        Int:2
\end{lstlisting}
```

And, while evaluation typically follows parsing, Martin advises the programmer to create a print function that "pretty prints" the result of evaluation based on the type of thing being returned. I originally decided to use built-in Javascript types for this test, but realized that Javascript's types might not mean the same thing as the intended types for the MAL language. I have not defined the MAL types yet, but doing so would allow for greater flexibility with type definitions and correct evaluation of the user's input.

\subsubsection[Synchronous vs Asynchronous]{My Javascript Woes}

Javascript is a programming language designed for the implementation of websites in a browser. Because of this, some of the required elements of a REPL that would be simple to code in another language required special consideration. For example, the built-in library readline sounds like it would be just the solution for accepting user input. readline even has a method ".question" which provides a prompt and accepts a response from the user. This assumes that the function is synchronous, or that the execution of the code depends on the sequence of the source, but this isn't the case with Javascript.

For browser development, asynchronicity makes a lot of sense: often, a web page is expected to provide multiple functions to the user at once. Facebook allows users to scroll through their timeline while they message their friends, for example. If

Facebook made every user wait for their timeline to load in order to send a message, most users would be frustrated by slow load times.

Although Node allows users to run Javascript in their terminal, this is not the natural environment that Javascript's builtins depend on. Most visitors to websites do not interact with the website via the console, but through DOM elements that have particular event listeners attached to them. However, when Javascript is run using a CLI, there are no DOM elements that the event listener can attach to, and therefore no way to listen for events. Because of this, readline stalls when used to solicit input via the command line.

Jim and I butt our heads against this problem for about an hour and a half before we realized that the asynchronous nature of readline was preventing the REPL from prompting the user. Thankfully, someone else out there tried to do something as silly as writing a REPL using javascript and rectified the problem by writing a readline-sync library. Even though this was a source of frustration that made me feel like I was "yak-shaving," I did learn something new about Javascript and its applications.

\section{Next Steps}

Logically, the next step to take would be to write the EVAL functions so that the REPL can actually interpret user input rather than simply echoing input back to the user. Having this happen properly would require defining MAL types that are distinct from Javascript types so that evaluation is coded correctly. I don't have much experience beyond writing a lexer and parser, and I wish I had gotten farther with this project, but fumbling around with Javascript took up more time than expexted. In the future, I might not use a language like Javascript to implement command line interactivity; it's awkward and feels a bit like forcing a square peg through a round hole.

```
\end{document}
```