Lab instructions

Creating a project

Open a terminal, and type:

```
dune init project halloween
```

This will create a new folder called 'halloween' that can be compiled using dune.

Compiling it

Let's try compiling our project, type:

```
cd halloween
dune build
```

The first command gets us into the halloween directory, the second compiles it. You'll be calling 'dune build' a lot: every time you solve type errors you'll want to re-run this to get the next error message.

Running it

After you've run the build command, there is an executable somewhere in the _build directory. It'll be different depending on your architecture. I'm on mac os X and for me the file is found in the same place as on a CSE machine I tested:

```
./_build/default/bin/main.exe
```

This will print 'hello world'.

Inspecting the generated files

Open the file 'halloween/dune-project' (in this document, I'll refer to files by their path, a slash, and then the file name). It should tell you the dune version used, the name of your project, a bunch of settings, and it has a placeholder for the author of the project: that's you! Go ahead and make some changes: put your name there for instance. If you're not planning to publish the source anywhere or license your code, you can remove the source and license parts. After you're done, check that the project still compiles by running dune build.

Adding a lexer and a parser

We'll make our changes to the lib folder: We'll start by creating a lexer and a parser. Go ahead and create a file halloween/lib/lexer.mll (so the filename is lexer.mll, but it should be in halloween/lib, per the convention I mentioned earlier) with this content:

```
{
 open Parser
 exception SyntaxError of string
}

let newline = '\r' | '\n' | "\r\n"
```

```
rule token = parse
 | [' ' '\t'] { token lexbuf }
 | newline { token lexbuf }
 | ['a'-'z' 'A'-'Z' '0'-'9']+ as word { WORD(word) }
 | _ { raise (SyntaxError ("Unexpected char: " ^ Lexing.lexeme lexbuf)) }
 | eof { EOF }
```

The first four lines are ocaml code that will be included in the generated lexer. The next lines define the types of tokens that our lexer parses. The incantation 'token lexbuf' skips a character (the word 'token' refers to the rule we are writing). We'll use the token WORD to parse any word. The code in our parser file will define WORD as a datastructure shared between the parser and the lexer, which is why we had to 'open Parser' at the top.

Let's write that parser too. Go ahead and create a file called halloween/lib/parser.mly:

```
%token <string> WORD
%token EOF
%start main
%type <string list> main
%%
main:
| line EOF { $1 }
line:
| { [] }
| WORD line { $1 :: $2 }
```

The first two lines define the tokens that are shared between the lexer and the parser. The next two lines define the starting token and the type it returns. The definitions of main and line are the grammar. In between the curly brackets is ocaml code again: it's the code used to build expression trees. Variables that start with a $ refer to the grammar.

Finally, we'll need to tell dune about our new files. We do so by editing the file 'halloween/lib/dune'. Its content should read (by adding the menhir and ocamllex lines):

```
(menhir (modules parser))

(ocamllex lexer)

(library
 (name halloween))
```

Let's check that everything still compiles by running 'dune build' (from the 'halloween' folder). I get this error:

Error: 'menhir' is available only when menhir is enabled in the dune-project file. You must enable it using (using menhir 2.1) in your dune-project file.

On my machine, I had to add it to halloween/dune-project, by putting this as the second line:

```
(using menhir 2.1)
```

On a CSE machine, this was done for me. Another error message I got (this time only on the CSE machine) was:

Error: Program menhir not found in the tree or in PATH

(context: default)

Hint: opam install menhir

You can follow this path for a while. If the machine tells you to run opam init, select 'y' for both options and then close and reopen the terminal (you don't have to reboot, but if you're using ssh you'll need to log back in). After a while, we should see no error messages when running 'dune build'. This completes adding a parser to our project. We've just not used it in our code yet.

Using our parser

Let's create a library of functions, create a file lib/halloween.ml with these contents:

```
let parse (s : string) : string list =
  let lexbuf = Lexing.from_string s in
  let ast = Parser.main Lexer.token lexbuf in
     ast
```

The .main refers to the main rule (you typically point to the start symbol when writing a parser), the Lexer.token refers to the rule called token we made in our lexer. Now we have a function that parses a string into a list of words (each of which is a string again).

Finally, we modify our executable, change the contents of bin/main.ml to the following:

```
open Halloween
let contents = parse "Hello world"
let () = print_int (List.length contents)
let () = print_endline ""
```

This allows us to use the parser (in the Halloween module), parse the string "Hello world" and print the number of words (and then a newline). Go ahead and build (dune build) and run (./_build/ and look for main), do you get '2'? If you add an exclamation mark to the string, do you get a parse error?

This way of running our code is a bit annoying: for every input we'd have to change our executable. That's hardly a usable program. Perhaps we can have our program take input from the command-line instead.

Change the contents of bin/main.ml to the following:

```
open Halloween

let rec keep_getting_input () =
  try
    let line = input_line stdin in
    let contents = parse line in
    let () = print_int (List.length contents) in
    let () = print_newline () in
    keep_getting_input ()
  with
    End_of_file -> ()
let _ = keep_getting_input ()
```

Now if you run the program, you can type a sentence, get the number of words, and keep going until you press ctrl-d (a way to signal 'end-of-file'). You can also cause a parse error by typing a symbol like ! or ?.

Hope you enjoyed!

Note to TAs: please sign off students if they've written a program that prints the number of words every time after you enter a line of words.

If you're getting confused, my file structure looks like this by the end:

```
halloween
  _build
    (contents depend on your platform)
  bin
    dune
    main.ml (contents modified in the last part of this tutorial)
  dune-project (contents modified by adding 'menhir')
  halloween.opam
  lib
    dune (contents modified to look at lexer and parser)
    halloween.ml (contents created to have the 'parse' function)
    lexer.mll (contents created)
    parser.mly (contents created)
  test
    dune
    halloween.ml
```

If you're looking for a challenge: have the parser just return a number, indicating the number of words, such that the program you write ends up doing the same thing, but without using the List.length function.

You'll need to change the function used in the parser (was the use of ::, should be a +1 instead) and the type returned (was a list, should be an int).