The language will be tentatively named Terrible Horrible Evil Haskell Compiler, or THE Haskell Compiler. The THE Haskell Compiler will be abbreviated as THC. The lexical specification is defined in grammar.txt

THC will be very similar to a subset of haskell, and will be compiled into C. Some modifications will have to be made since haskell has whitespace dependent blocks, which is not an option here. We will separate statements with semicolons, and delimit blocks with braces if necessary. The two main features at the start will be type definitions, and function definitions. Types will follow an algebraic type system. An algebraic type system is very easy to implement in C, as the product of types can be implemented as a struct, and the sum of types can be implemented as a tagged union. Type parameters can be treated as generics, and generated as needed at compile time.

For functions, we want to try lazy evaluation and currying. Curried functions can be stored as a struct containing a pointer to a variable argument function, its arity, and a list of currently passed parameters. For lazy evaluation, an evaluated flag and the value can be added to the struct. The struct will store all the params the function needs to be called with until it is called.

For the grammar, we will start with just the type definitions, and the function definitions, and then add more haskell features if time allows.
The initial types will be Bool and Int, and the user may define their own algebraic data types. Bool and Int will both be represented by an int in the compiled C code.
On a side note, since we have algebraic data types, we don’t actually need to define any data types, and the user can just define their own implementation of the natural numbers and derive everything else from there.

The only syntax other than function definition, data definitions and expressions that will be implemented at the beginning will be case statements, that allow the user to pattern match data types. This can easily translate into a few if statements in C.

Testing the output of the compiler can easily be done by compiling the output with gcc and running as an executable.

Haskell defines ten infix operator precedence levels, with the option of left associative, right associative, or non-associative for each level. We may want to implement this behaviour at a later stage in the project, so we will define thirty lexical classes for infix operators at the start, although most will be unused. Statements defining infix operator precedences could be read by a preprocessor, which instructs the lexer on which lexical class each operator should go in. Operators will default to precedence 9, non associative.

Other lexical classes will include lowercase and uppercase identifiers, since they should be treated differently, “case” and “of” for case statements, semicolons for delimiting lines, braces for delimiting the block of a case statement, “->” for statements in a case statement, “data” and

“|” for defining data types, and “=” for assignment. All infix operators such as “==” or “>” which behave like functions will go in one of the thirty lexical classes for infix operators.

Lexer classes are defined in grammar.txt under “#Lexer classes” and are given as regexes. Grammar production classes are given above that in a format similar to the example provided.

Example program:

```
data List a = Empty | Node a (List a);
data Maybe a = Just a | Nothing;

(:) :: a -> List a -> List a
(:) = Node -- oh god i hope we can make this work please

len :: List a -> Int;
len l = case l of {
    Empty -> 0;
    Node _ ls -> add 1 (len ls);
};

head :: List a -> Maybe a;
head a = case a of {
    Empty -> Nothing;
    Node a _ -> Just a;
};

square :: Int -> Int;
square x = mult x x;

main :: Int;
main = add (square (head (2 : Empty))) (len (2 : Empty));
```

In the beginning we’ll probably just evaluate the value of main and print that to test our compiler, since monadic IO and RealWorld are a bit hard. The above program should print 5.

Update: lexer is functional with classes generated at runtime.