Challenges & Answers

4. Scanner

1. The lexical grammars of Python and Haskell are not regular. What does that mean, and why aren't they?

A regular lexical grammar is a grammar that generates regular languages, which are the languages that can be recognized by finite automata. It consists of:

1. Terminals: Symbols that form strings
2. Non-terminals: Variables used to derive strings
3. Production rules: Rules for transforming non-terminals into terminals or other non-terminals
4. Start symbol: The non-terminal form which derivation begin

Python and Haskell are not *regular* because to parse a variable in these languages, we need additional information from its surrounding context. Take Python for example, since the meaning of each symbol depends on its indentation, the parser has to consider the surrounding layout to parse that symbol. The indentation, however, is not preserved in the token stream after lexical analysis since whitespace is stripped by the lexer.

2. Aside from separating tokens—distinguishing `print foo` from `printfoo`-spaces aren't used for much in most languages. However, in a couple of dark corners, a space does affect how code is parsed in CoffeeScript, Ruby, and the C preprocessor. Where and what effect does it have in each of those languages?

In C preprocessor, a space can effect how macro definitions are parsed. For example, using the symbol # with a macro parameter means a stringitization of the parameter. However, if there's a space between # and the parameter name, the C preprocessor will treat it as a the start of an empty preprocessor directive. Another example is with macro function definitions. The macro defined as `#define SUM (a, b) (a + b)` contains a whitespace between `SUM` and `(a, b)`, which when used, will not add `a` and `b` together but instead replaces the whole text with the literal string `(a, b) (a + b)`.

3. Our scanner here, like most, discards comments and whitespace since those aren't needed by the parser. Why might you want to write a scanner that does not discard those? What would it be useful for?

Some scanner needs extra information like indentation information, or converting comments into docstrings / type conversions.

7. Evaluating Expressions

1. Allowing comparisons on types other than numbers could be useful. The operators might have a reasonable interpretation for strings. Even comparisons among mixed types, like 3 < "pancake" could be handy to enable things like ordered collections of heterogeneous types. Or it could simply lead to bugs and confusion.

In lower level langauges, particularly those with direct memory control and reinterpretation, the ability compare different types together is likely to lead to bugs and confusion. Since the users of these languages often needs to manipulate the bytes of a value itself, mixed types comparison causes confusion because it doesn't inherently know which bytes in the variable have special meanings that the user intended and which bit does not. One prominent case is using int as a flag in C: If C allows comparison between int and float, then the user decided to compare the flags set on int A with the flags set on a (mistakenly typed) double B, then they are not comparing if the flags are similar; instead, they are comparing if the value approximation of the double is the same as that in C, which goes against their original intention without any warnings. Apart from that, mixed types comparison between collections of heterogeneous types can be useful if the criteria for comparison is **clear**, let's say only comparing between lengths, then I assume that could be beneficial. Anything else more than that would likely cause confusion.

In high level languages however, mixed types comparison work because they don't have the necessary tool to reinterpret / interact with the underlying representation of the type in an unpredictable way. For homogenous collections comparison, I say it's also a good thing to keep them clear and straightforward, mainly because you want to keep the performance of the language well optimized. That said, since higher level languages have a higher degree of freedom in terms of performance tradeoffs, more useful comparison functions, like comparing element wise could be useful if there's no way to override the default comparison (through == or ===) behavior of the language.

After examining Python, I saw that it allows you to compare different collections together, even if they are not related, for example comparing between a list and a dictionary. The comparison of course always evaluates to `false` since there are no valid criterias to compare them, but it doesn't raise any syntax / runtime error (might raise a warning if you're using an LSP). Like mentioned above, I think these obscure comparisons shouldn't exist in a language, regardless of it's support for bit manipulation, since it makes bugs hard to find without an lsp. That said, it also makes writing Python a lot easier for beginners?

2. Would you extend Lox to support comparing other types? If so, which pairs of types do you allow and how do you define their ordering? Justify your choices and compare them to other languages.
If possible, I would include an option to override the == comparison operator for more user flexibility. Apart from that, for native out of the box support, I think stopping at adding comparison support for homogenous collections based on their length should be appropriate.

3. Many languages define + such that if either operand is a string, the other is converted to a string and the results are then concatenated. For example, "scone" + 4 would yield scone4. Extend the code in visitBinaryExpr() to support that.

4. What happens right now if you divide a number by zero? What do you think should happen? Justify your choice. How do other languages you know handle division by zero, and why do they make the choices they do?
For floating point numbers, IEEE 754 floating point number standard defined dividing a floating point number by zero to result in either infinity or negative infinity if the numerator is a non-zero number. The operation should return NaN (not a number) if both the numerator and denominator are floating point numbers. Since I wrote the Lox interpreter in C++ and represented numbers using the `double` type, the behavior when dividing by zero should follow IEEE 754.

For integers, there isn't a defined way to handle divide by zero. Following popular languages like C, C++ and Rust, dividing by zero on an integer type raises a runtime error / abort the program immediately.

For Lox, I think that it is good to have support for infinity, negative infinity and NaN type, just to give users the flexibility to use them and since it adds no complexitiy to our implementation for these values are supported natively by C++.

5. Change the implementation in visitBinaryExpr() to detect and report a runtime error for this case.

8. Statements and State

3. What does the following program do?

The following program prints out `3` because the `a` inside the scope is overriding the global `a`, but the assignment expression was evaluated first before `a` was inserted into the scoped environment. This behavior is supported through the shadowing mechanism in Rust, but leads to undefined behavior in languages like C++, C, and straigt up error in Java, Javascript and Python. Even though C++ and C does allow variable shadowing for variables of different scopes, since the variable declaration is processed before the assignment expression is evaluated in these languages, the `a` in `a = a + 1` refers to the new variable being declared, and the new `a` has not yet been assigned a value, which leads to undefined behaviors. In Java and Javascript, this leads to an error since they don't support variable shadowing. Python is a special case, since there's no clean way to achieve this lexical scoping without additional statements, but if we were to put a similar code into a function

```
global a = 1
def func():
  global a
  a = a + 1
func()
```

then Python will report that a is accessed before it's initialized, for reasons similar to C and C++.

9. Control Flow

1. A few chapters from now, when Lox supports first-class functions and dynamic dispatch, we technically won't need branching statements built into the language. Show how conditional execution can be implemented in terms of those. Name a language that uses this technique for its control flow.

First clss functions and dynamic dispatch can be used to replace control flow by offloading the correct function call to the dispatcher at runtime through common function pointers, first class functions or interfaces. For example, a piece of code like

```
if (condition) {
  op1();
} else {
  op2();
}
```

can be rewritten to remove any branching statements

```
statements = {op1, op2};
op = statements[condition];
op();
```

provided that `op1` and `op2` share similarities that would enable dynamic dispatch, i.e sharing the same base class, sharing the same signature etc… Another way to avoid branching logic is to do everything through message sending, similar to Smalltalk.