Design **and** developmenT

My design and development sections are going to be integrated into one, as instead of planning the entire project, and then coding the entire project I feel like the better approach is to tackle it one module at a time, as this allows a lot of room for my program to evolve. Therefore, I will design each module before starting development, then develop it, making changes from the original plan, and testing each newly added feature for all of the use cases.

Doing this allows any changes I make in development to not affect the later stages of the program, because a lot of code in an interpreter builds on previous code, so any changes could leave the rest of my plan redundant which would be a large time loss.

## References

This is a list of some of the sources I have used for the project. This was only really relevant in the BODMAS stage as afterwards I used completely my own concepts; however I never directly copied any code: only seeing how the overall structure of the Interpreter had to be built.

🡺 [https://youtu.be/SToUyjAsaFk](https://youtu.be/SToUyjAsaFk?list=LL)  
🡺 [https://youtu.be/JO\_0e9mPofY](https://youtu.be/JO_0e9mPofY?list=LL)  
🡺 <https://craftinginterpreters.com/a-map-of-the-territory.html>

# design: An overview of interpreting

This first section is going to be an overall demonstration of how my interpreter is going to function. We are going to take a relatively basic arithmetic expression, in plaintext, and go through the logical steps the interpreter will take to produce a result. This includes the three main stages: lexical analysis, parsing and then execution. I will later go into detail about the plans to code each stage, but this is a general abstracted explanation of the process which removes all the complicated details.

## Lexical Analysis

The interpreter will take a plaintext input, for example being the one below, which we will be using throughout this demonstration to eventually produce a result.Currently, to the computer, this string of characters holds absolutely no meaning or value, so in lexical analysis, the meaningless string of characters is converted into a list of tokens. This is done by a program called a lexer.

On a simplified level, the lexer takes a line of the input text, and goes character by character, looking at the type of character or arrangement of characters, and converts it into a list of objects, which are called tokens. Some characters are very easy to convert into tokens, such as parenthesis, however, when tokens which are longer than a character long become involves, it requires much more difficult code to distinguish multi-digit integers, strings, floats and variable names.

The lexer must also be trained on what to ignore, including white-space characters and comments, as what is produced at the end of the lexing process must be a list of meaningful tokens which can be given to the parser so the next stage can start.

A hexagon with black text

Description automatically generatedIn terms of the tokens that are produced themselves, they are all objects, as I am using an object-oriented approach to this project. Every single token has a token type, and every single token type has its own class, with different properties. Throughout the process, we will define and create different types of tokens, but for now we have four different groups of tokens.

🡺 Literal tokens hold values, with different types being integers, floats, Booleans etc. They will have a property called value which, obviously, holds their value.

🡺 Identifiers, at this stage, are variables, and will later include function and procedure names. For now, they would have a property which points to the location of their value in the program (to be expanded on later)

🡺 Binary operators are the arithmetic operations, including adding, subtraction, multiplication and so on.

🡺 Finally, there are other kinds of characters, such as parenthesis and dots, which may be included.

At this point, the objects still hold no logic or meaning behind them but is merely the plaintext described in a way that a computer could begin to understand it: the equivalent of us reading a sentence, but not yet understanding the grammar and meaning behind it.

A number and text on a black background

Description automatically generated

Using our example, at the end of the lexical analysis stage, we have been left with this array of tokens which is equivalent to the plaintext, ready to be passed onto the next stage of the process.

## The basic concept of parsing

A screenshot of a computer

Description automatically generatedParsing is the most complex stage out of the three and is where the tokens from the lexer are given a meaning, a grammar, that means something to the computer. This is where our simple list of objects is built into an abstract syntax tree, which is a type of data structure. Where each of our objects is linked in some way to another object, as its children. The job of the parser is to correctly link all these objects together in the correct structure based off the rules that we give it.

These set of rules become very complex, but in our example, the computer needs to arrange the objects based on the mathematical principle, order of operations. This has two key parts:

🡺 Precedence: the order itself. This is how certain features have priority, with items in parenthesis being calculated first, then any exponents, then multiplication and division, then addition and subtraction, according to the BODMAS structure of mathematics that we use.

🡺 Associativity: this is the order in which operations with the same precedence are executed. For example, a chain of subtractions will result in a different value based off whether they are calculated from left to right or from right to left. In BODMAS, this is from left-to-right for arithmetic expressions.

Based on these rules, the tree is created, with the calculations that need to be performed first at the bottom of the tree, and this is the overall premise of parsing before the result is produced.

Although this may seem simple with our BODMAS example, the premises of precedence and associativity are still present for all other aspects of the language. When keywords and different data structures and functions and all the specific features of the language become involved, it makes this parsing step extremely complex to create.

### What are the set of rules?

In my brief description, I described how a set of rules determined how the abstract syntax tree would be created. This sounds extremely vague, but it needs to be extremely precise in the actual code. This is where a special kind of notation is used, called Backus-Naur form - or BNF for short - is a context-free grammar used to describe the how the logic behind a grammar is defined.

#### A basic example of bnf

For this we will be defining what an integer is using BNF. There are a few key aspects of BNF. The definitions by themselves may not make much sense but will be clear with context afterwards.

🡺 name = a terminal (aka lexemes), this means it is in its lowest state, and cannot be expanded upon, for example the digit 1 is a terminal as it is clearly in its defined form

🡺 <name> = a non-terminal, which can be expanded and defined by a series of terminals or non-terminals

🡺 | = or, used when defining a non-terminal which can have several different ways it can be defined

🡺 ::= = the equals of BNF, where the non-terminal on the left-hand side of the ::= is defined by what has been written on the right-hand side of it.

This may sound extremely confusing for now but let’s begin the process of defining an integer. We must first decompose the integer into the smallest item that it is comprised of: a digit, which we can easily define with the following statement:

<digit> ::= 0|1|2|3|4|5|6|7|8|9

This statement shows how a non-terminal called a digit, is defined as a 0 or a 1 or a 2 or a 3 and so on. This may seem redundant and obvious information to a human, but every little thing must be explained to a computer. This is why this example is a good introduction to BNF, as it shows how every little detail must be considered so the grammar is built from the ground up. Furthermore, we can then expand on this definition.

<integer> ::= <digit>|<digit><integer>

This statement says that the non-terminal integer is defined as a digit OR a digit combined with an integer. This recursive self-defining forms the basis of BNF, as therefore this means that an integer is defined as 1 or more digits, with no limits on the number of digits available, hence defining the integer as every single possible integer ever. It is the structure like this that allows BNF to represent the grammar of entire programming language by building up from the most basic steps.

However, our definition of an integer is still wrong, because an integer cannot begin with a 0 as the first integer, or two zeros as the first two, as these integers have already been defined. Therefore, the actual definition for an integer would be as follows:

<non-zero digit> ::= 1|2|3|4|5|6|7|8|9  
<digit> ::= 0|<non-zero digit>  
<digits> ::= <digit>|<digit><digits>  
<integer> ::= <digit>|<non-zero digit><digit>

By ensuring that it does not begin with a zero, this has already significantly increased the complexity of the BNF by at least double, but it does show how important being precise is, and the attention to detail that is required in order to produce an interpreter that does not contain lots of errors.

#### Extended backus-naur form

Shortened as EBNF, Extended Backus-Naur Form is a more concise way of writing in BNF. Although everything in EBNF can also be written in BNF, it just makes the process easier to read whilst avoiding lots of repetition and reduces the amount of self-recursion. Taking our previous example, the new definition of an integer would be.

<non-zero digit> ::= 1|2|3|4|5|6|7|8|9  
<digit> ::= 0|<non-zero digit>  
<integer> ::= <non-zero digit>{<digit>}

The introduction of the curly brackets { } mean that whatever is contained within them can be repeated zero or more times. This eliminates the need for something to be defined by itself, and makes it overall clearer to read. Other additions to EBNF include:

🡺 (a|b) = an option that needs to be made, for example  
<x> ::= (a|b)<y> 🡺 <x> ::= a<y>|b<y>

🡺 [a|b] = an option that doesn’t have to be made, for example  
<x> ::= [x|b]<y> 🡺 <x> ::= <y>| a<y>|b<y>

Again, these shortenings just reduce the amount of repetition that needs to be made whilst writing BNF. There are different variations of the syntax that use different variations, but the only one I will really be using are the curly brackets, which for clarification has the example:

<x> ::= a{<y>} 🡺 <z> ::= <y>|<y><z>  
 <x> ::= a|a<z>

As this is very useful for shortening and occurs very commonly, but the main point to take away is that any notation that I write in EBNF is always able to be written in the simplest form of BNF, showing that the entire grammar of the language stems from these small definitions.

#### Order of operations in bnf

Just as anything in a language can be expressed in BNF, our order of operations problem can also be written in BNF. Below is the basic version which we can use to solve our solution. We will expand upon this later to include exponents and prefix operators, but for now we will just include the necessary addition, subtraction, multiplication, division and parenthesis needed for our example. The required EBNF is:

1) <expr> ::= <term>{(+<term>|-<term>}  
2) <term> ::= <factor>{\*<factor>|/<factor>}  
3) <factor> ::= (<expr>)|identifier|integer

A number and symbols on a black background

Description automatically generatedThis simplified solution also defines identifiers and literals as terminals rather than non-terminals. It can be argued that they would be non-terminals, but for the sake of explanation I have defined them as terminals, with identifiers being variables and integers holding a value.

Take this small example for an explanation of the process, starting with this set of tokens:

A number and symbols on a black background

Description automatically generatedLooking at the initial tokens, we have 3 literals, which are integers, and 2 binary operators, a multiplication and an addition. Just for clarity, we can label our integers, to distinguish them from the binary operators, which hold no value and are only there for logic.

A screenshot of a computer

Description automatically generatedLooking at our integers, our third line of EBNF states that a <factor> can be defined as an integer, so therefore each of our terms can now be labelled as a <factor>.

A computer screen shot of a number

Description automatically generated with medium confidenceAccording to our second line, one of the definitions of a <term> is two <factors> multiplied by each other, so we can therefore group our <factor>\*<factor> and label it is a <term>. The remaining 4 is also labelled a <term>, as a <term> can still be defined by a <factor> by itself, so long as there are no \* or / binary operators next to it, which leaves us with two <terms> separated by a + binary operator.

Now, our first line of BNF defines an <expr> as a <term>, followed by a + or – binary operator, followed by a <term> (just once in this case but can be repeated). Therefore, we can now define our entire expression as a single <expr> non-terminal as shown below.

A diagram of numbers and symbols

Description automatically generatedA diagram of a mathematical function

Description automatically generatedThis expression is now completely defined, as an entire tree has been created, starting from the single <expr> non-terminal, which allows us to then progress to the next stage after parsing. This more detailed perspective of how the different steps of logic work lead to the following abstract syntax tree:

This holds the exact same meaning as the previous diagram, just presented in a different, less confusing way which is closer to how it will be treated for the final stage. In reality, each of the different boxes is an object, with each of the arrows being a property within the object that points to the other object as its children, so our diagram is just a visualisation. However, the key takeaway is that our parser turns our set of tokens from the lexer into this abstract syntax tree, stemming from a single initial object at the start, which descends into sets of different pairs until the whole expression is now defined aligning to the rules that we have chosen, so that our precedence (order of operations) will be correct.

A computer screen shot of a graph

Description automatically generatedTo go back to our example, our original string of tokens will produce a BNF tree like this:

This clearly shows how quickly the diagrams get messy using this approach, and how complex the process of parsing is, so it is best to use abstraction, and remove the details of how the parser works, leaving us with our final parsed tree which can be passed onto the next stage.

A screenshot of a computer

Description automatically generatedThis process of parsing is called recursive decent parsing, and the process itself is very detailed, and can cause lots of errors which I will not go into detail about now. In the later stages of design, I will explain the actual coded steps in order to perform this process, but for now it is just the logic behind it that is of importance. The key points from this are that:

🡺 The rules of the parses must be very carefully defined  
🡺 All possible inputs must be thought out  
🡺 The process itself is very long and complex  
🡺 The result that must be produced must start from a single object.  
🡺 The order in which the tree descends must match the precedence of the operations so that the execution stage is correct.

Our original other point of associativity has not been mentioned but it was defined in the DNF stage. If we were to reverse the way that we wrote our DNF rules, then the operations would have been performed right to left but ensuring that associativity is correct in the parsers process is something extremely important that I have not mentioned.

## Execution

A diagram of a mathematical function

Description automatically generated with medium confidenceThe final stage is taking our abstract syntax tree and producing a single result. From now on, it is more useful to focus on the fact that the tree is a set of connected objects, rather than the very over-simplified diagram. Continuing from our small example earlier, we can represent the set of diagrams as shown:

This approach highlights a better way to visualise the tree, not just numbers connected by operations, but only integer objects, with their values being properties, rather than being the main part of the object itself/ Additionally, rather than them being magically “connected” to each other, the binary operator objects also have two properties: a left child property and a right child, which then each point to another different object. This focus on this object-oriented perspective will make the overall execution stage a lot easier.

There would be lots of different properties for the different types of objects that we could have. For example, any variable objects would have a property that points to the location where their value is stored. Or, the negate operator [for example –(3\*2)] would only have a single child, rather than a left and right, as it is not a binary operator.

Despite all of the objects having different properties, one of the things they would all share is an evaluation method, called eval(). This use of polymorphism – one of the pillarstones of object-oriented programming.

A diagram of a mathematical system

Description automatically generated with medium confidenceThe evaluation function of each object returns it’s “value”. The most obvious instance of this is for an integer object, which returns the value of its value property as the evaluation method. Similarly, a variable would return the value stored in the memory location that its property points to. These are all very straightforward.

Where the complexity arises is with the binary operators. The result of the evaluation method of an addition object, is the result of the evaluation method of its left child, plus the result of the evaluation method of its right child.

This is relatively straightforward with just one binary operator, but when one of its children is another binary operation, its evaluation method has to call upon a further two evaluation methods in order to get it’s value, which it then returns to the previous.

A screenshot of a computer

Description automatically generatedThis means, that when you want to find the result of an expression, you call the evaluation method of the object at the top of the tree. In turn, this will then descend down the tree, calling the evaluation method of every “branch” object in the tree until the “leaf” objects are reached. The results are then returned back up the tree, so each method can obtain a value, back to the final starting object which returns the final answer

This is how the final evaluation stage of the interpreter. I will not fully explain our example, as it would merely be a lot of repetition, but the key focuses are the same. Descending down the tree, calling the evaluation method of each of its children, down to the bottom of the tree. Then, all the returns making their way back up to their parents, up to the top addition object, where the value of the equation has been determined.

Again, like lexing, this process is relatively straightforwards when you understand it, and it definitionly presents parsing as the most complicated step in the process, as it’s finding out the order of what to do that’s the most computationally complex.

Note that, in our simple example, the evaluation method only returns a value, but when we begin to impliment more complex features, such as seting variable values, iteration and function calls, this evaluation method will contain a lot more steps than merely returning a value, in order to actually change something in the code. But, for now, all we need to do is return values.