Chapter 1

Interpreters and Compilers

Parsing helps us figure out what all these squiggles on the page mean

Cliff

1.1 Introduction

We know how a grammar describes a language. Let us consider the following language which I will call Math-ew:

$$\begin{aligned} E \longrightarrow & \quad +NE \mid -NE \mid *NE \mid /NE \mid N \\ N \longrightarrow & \quad 0|1|2|3|4|5|6|7|8|9 \end{aligned}$$

Math-ew describes simple mathematical expressions. Some sentences in this language include "`+ 3 4`" and "`- 3 * 4 + 5 2`". We will use Math-ew throughout this chapter so feel free to refer back to it.

Now that we have a language to work with, the next question is how do we go from something like "`+ 3 4`" to what we can assume to be the correct value, 7? That is, in `utop` or `irb`, why is it that when we enter something like "`3 + 4`" we get back 7?

This is all the work of an interpreter. I personally call this a compiler, but the connotation is important.

1.2 Compilers/Interpreters

Consider what happens when we take a file like "`funs.ml`" and then run `ocamlc funs.ml`. It is ultimately the same thing that happens if we ran something like `gcc prog.c` or `javac Program.java`. We take a text file [1] and compile it down to machine code and then we get some program we can run. Practically though we take the text file and convert it to an assembly file (technically another text file) which an assembler then converts to the appropriate machine code.

All of this is to say we think of compilers as something that takes our code and creates a program, but this is practically incorrect. A compiler is a language translator. So you could theoretically create a Java to C compiler, or a Ruby to Java compiler [2].

An interpreter on the other hand takes a text file and returns a value. `irb` is an interpreter since it does not make assembly or machine code, but instead gives back a value. I would argue this is also a compiler because I could just define the target language as something like

$$S \longrightarrow value$$

. However maybe what I should say is that both interpreters and compilers are translators, but they translate in 2 different ways. A compiler translates by making machine code and converting an entire program to an analogous program in a different language. An interpreter translates by converting one statement at a time and evaluating these statements to a value.

[1] The only difference between "`funs.ml`" and "`funs.txt`" is the file name (which is arbitrary).

[2] See `https://pandoc.org/`

Regardless of if we are talking about a interpreter or a compiler, typically there are three things needed to make a translator:

- lexer: converts text file to a list of tokens
- parser: takes list of tokens and creates an intermediate representation (Typically a tree)
- evaluator/generator: takes the intermediate representations and either evaluates to a value, or generates analogous code in a different language.

We will talk about all of these things with an example for Math-ew written in C.

1.3 Lexing

When you are reading this, you are looking at squiggly lines made up of ink or pixels and being literate, you are able to make meaning of these squiggly lines. Like what a superpower: you can look at squiggly lines and then gain knowledge from said squiggly lines.

Lexing is analogous to figuring out what the words are. Consider the following sentence:

Your tongue does not fit comfortably in your mouth.

You see some squiggly lines and then get upset that you are now thinking how you should position your tongue. Magic. But lexing is just the process of figuring out what words are in the sentence (and maybe what type they are). That is, you split up the sentence into nine words: `"Your"`, `"tongue"`, `"does"`, `"not"`, `"fit"`, `"comfortably"`, `"in"`, `"your"`, and `"mouth"`. You may even tag certain words as a noun or verb or determiner. However, notice that we said it's just figuring out what the words are. Given the string

"green the truck"

You still recognize there are three words, `"green"`, `"the"`, and `"truck"`. Despite this being grammatically incorrect, you still were able to figure out these words. That is exactly what a lexer does.

When creating a new language, you would typically have a list of words which should be allowed in the language. In Math-ew we have operator words ("+", "-", "*", and "/") and digit words ("0", "1", "2", ... "8", "9"). So we should have something like

```
type token = Plus|Sub|Mult|Div|Num of int
(*
If we wanted, we could do something like
type token = Operation of string|Num of int
*)
```

We then want a function that takes a string and returns a list of tokens. We could do this by doing something like the following:

```
let rec lex str =
    if str = "" then [] else
    if String.sub str 0 1 = "+" then
        Plus::(lex (String.sub str 1 ((String.length str) - 1)))
    else if String.sub str 0 1 = "-" then
        Sub::(lex (String.sub str 1 ((String.length str) - 1)))
    else if String.sub str 0 1 = "*" then
        Mult::(lex (String.sub str 1 ((String.length str) - 1)))
    else if String.sub str 0 1 = "/" then
        Div::(lex (String.sub str 1 ((String.length str) - 1)))
    else if String.sub str 0 1 = "0" then
        Num(0)::(lex (String.sub str 1 ((String.length str) - 1)))
    ...
    else if String.sub str 0 1 = "9" then
        Num(9)::(lex (String.sub str 1 ((String.length str) - 1)))
    else lex (String.sub str 1 ((String.length str) - 1))
```

This is not the most efficient way to do this. This also assumes you will be fed valid input (a non-malicious user). I would recommend looking at the `Str` library which lets you use regular expressions in Ocaml.

However, this does work as a lexer.

```
let rec lex str =
lex "1 + 2" = [Num(1); Plus; Num(2)]
lex "1 2 +" = [Num(1); Num(2); Plus]
lex "+ 1 2" = [Plus; Num(1); Num(2)]
```

Again, notice that our lexer does not check if the string matches the grammar. All it does it check if all the words in the string are valid. In this particular lexer, we skip over any unwanted words but may not always be the desired behavior.

1.4 Parsing

Now that we have a list of tokens and know that we have a list of valid words (tokens), we need to make sure our sentence is grammatically correct. That is, if we have the string

green the truck

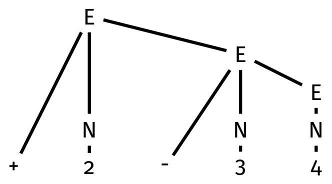
You lex this correctly and recognize all three words (`"green"`, `"the"`, `"truck"`) are valid words in the English language, but you notice this is grammatically incorrect. This is what the parser does. Check if things are grammatically correct or not, while also storing the sentence in an intermediate form, whether it be a tree, code, or something else.

This is typically done to make the evaluator or generating step easier, but from a linguistic point, we gain information just by having something be grammatically correct. Consider the sentence `"I like purple"` and `"I like purple shirts"`. By being grammatically correct, we know that in the first sentence `"purple"` is a a noun, whereas in the second sentence `"purple"` is an adjective. It would be harder to identify this if things were not grammatically correct.

Additionally there are many types of parsers that exist so depending on the grammar or what you want to prioritize using a different parser may be useful . I will be using a Left-to-right Left-derivation parser, that looks ahead by 1 token at a time or a LL(1) parser. A LL(1) parser is a type of a Recursive Decent Parser (RDP). This means that we will be using a left most derivation when checking the grammar, reading from left to right. The lookahead by 1 just means we will be looking one token at a time (to decide what production to use if needed).

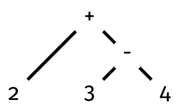
Hence why we may need to change our grammar to match what our parser is capable of. Regardless of which parser you use, you probably want to have a different data structure of representing the sentence. When talking about CFGs, we used a tree which is what we will use here. We will be using a Abstract Syntax Tree (AST) which will just make a tree based off the content of the language. We could make a parse tree instead which makes a tree based off the symbols of our language, but parse trees focus more on non-terminals (internal nodes) vs terminals (leaves) whereas ASTs focus more on content of the string.

Let us consider the following Math-ew sentence `"+ 2 - 3 4"`. The parse tree would look like:



Notice this is more closely aligned with the grammar. Where E is either N or some operator followed by two other E values.

For an AST, we care more about what is occurring. Here is an AST for the same sentence:



Either option is a valid output for a parser, but for more complex grammars, we sometimes don't care about all tokens (like white space or {}) or we can represent how some of the tokens are used via the structure of the tree. For example If we had parenthesis to show order of operations, the above AST would not need to include them since the structure of the tree shows that we want to subtract 4 from 3 before we add 2. Hence a parse tree could be replaced with something more useful.

So if I wanted to create an AST, then I have to consider how to define and design my tree. The nodes in my tree will either be a number (leaf) or a operation with the subtrees being children of the node. Translating this to code looks like:

```
type ast = Int of int
          |Add of ast * ast
          |Minus of ast * ast
          |Times of ast * ast
          |Divide of ast * ast
```

Now we need to make the parser. For Math-ew (and typically other recursive languages), we need to consider the fact that for a sentence like "- 2 * 2 3", that the character "2" is a stand alone sentence. However we cannot recursively call here despite the fact that $E \longrightarrow N$. That is we run into a small issue:

```
let rec parser tok_list = match tok_list with
[] -> failwith "Math-ew does not support empty strings"
|Plus::t -> let next_expr = parse t in Add(next_expr, ?)
...
```

We have no idea what should be in the spot of the question mark. Since `parse t` would parse the rest of the list and still have something left over (namely `[Mult; Num(2);Num(3)]`).

We can solve this problem in 2 ways, either we find some way to figuring out our remaining tokens or we can consider how our language is structured. Let's do the latter since it's harder.

In our language, we know that any operator will be followed by a number or an expression. Hence, we can mimic our grammar with our parser like so:

```
let rec parse tok_list = match tok_list with
Num(x) -> Int(x)
|Plus::Num(x)::t -> Add(Int(x), parse_E t) (* + N E *)
...
```

Here, we can break down each branch as an operator, followed by a number, followed by the remainder. Technically though we are looking ahead by 2 tokens so this is a LL(2) parser. We can easily fix this in the following way:

```
let rec parse tok_list = match tok_list with
Num(x) -> Int(x)
|Plus::t -> let h::t = t in Add(Int(h),parse t)
...
```

A slight difference, but syntactically shows that we are only looking 1 token ahead at a time instead of 2. We can expand on this more in appendix D //TODO.

Notice that this also will only match on gramatically correct sentences. So if our sentence was something like "+ - 2 3 4" we would not match and we would probably throw an error. Which means that now all we have left is to take meaning from the sentence.

1.5 Evaluating/Generating

Now that we have a parser that generates a tree of some sort, now we need a way to traverse through the tree to compute a final value (at least for an interpreter). We need to create meaning from our AST. So suppose that we have a sentence like:

Colorless green ideas sleep furiously

This contains all valid words in the English lexicon (our lexer says ok), and it is grammatically correct (our parser would make a tree) but in English this means nothing. The evaluater's job is to make sure this can be done. We helped this process by designing our AST in a way that we can easily traverse where each node has meaning.

That is if we consider the above AST, notice that we just need to perform a post order traversal to compute a value. That is, at the root, we need to process our left subtree and then our right subtree, and then we can add our values together. This can very easily be modeled with the following code:

```
let rec eval ast = match ast with
Int(x) -> x
|Add(x,y) -> let left = eval x in
              let right = eval y in
              left + right
...
```

TA-DA! We now have a way to get a single value from a string. Namely:

```
eval (parse (lex "+ 2 3")) = 5
```

Typically an interpreter will take a program statement by statement and do this exact thing. A compiler will look at a list of strings (read: text file) and compute an analogous text file.

Notice that technically, the meaning from this evaluator is created purely from what we decided Add would do. Additionally this language is very simple. Once we start adding more to our language, it may be possible that we have something that follows our grammar but makes no sense. Typically we can fix this by changing our parser, or rewriting the grammar (there are multiple different grammars that express the same language). In some cases, type checking is done during evaluation (like in ruby).

Suppose that I had a statement that looked like `"x+ 1"`. In most programming languages this is grammatically correct, variable added to a constant. However if $x = true$ then trying to do this operation would fail and it would be meaningless. Some languages get around this by casting, or by creating a new data type or behavior (see javascript and C).

This idea of defining behavior of a language is a branch of semantics. One particular type called operational semantics is the next topic.