fck Developer Book

v0.1.0

nxe

Contents

Appendices

Preface

This book has been written to accompany the fck compiler and CLI. It will go through design choices of the language and implementations for the compiler. This book is not intended to teach you how to write a programming language and assumes some basic knowledge in the subject.

Before you start, some notes. Firstly, the above contents are not exhaustive, they include only main headings. Each section will have a sub-contents that lists all the second level headings inside it, with each of the second level headings listing all sections inside it. Secondly, the terminology used in this book regarding languages is quite specific. The term *language* will refer to the fck programming language (or a more general programming language; whereas *spoken language* will refer to a non-programming language that is spoken and written, such as Spanish.

This book is for version 0.1.0 of fck and was completed in October 2023.

1 Introduction

This section will give a brief introduction into fck. Namely:

1. Justify the need/desire to have a multi-lingual language (Section 1.1)
2. Explain how the multilingual aspect functions from a user perspective (Section 1.2)

1.1 Justification

Programming languages are almost entirely written for English speaking programmers. There are very few languages that can be written, without the need for external files, in any other spoken language than English. For completeness, we will discuss one such language in Section 1.1.1.

The world is increasingly becoming more and more reliant on technology, and having programming be reliant on knowing English seems a bit wrong. As an interesting note related to this, approximately two billion people speak English, or around 25% of the world population.

This is why I made fck. To allow more people to write code, and to allow more collaboration through code.

1.1.1 Citrine

Citrine is the only real example I could find of a multilingual language. Personally, I find some aspects of Citrine to hinder it:

1. **Use of icons**
   The designers of Citrine cite Smalltalk-70 as one of their design inspirations, and have taken the usage of icons into their language. This, I believe, is one of the weakest points of the language. The usage of icons can be quite helpful in some situations. One notable example is APL which is famous, or infamous, for being completely incomprehensible to anyone who doesn't know what any of it means. For example, `(((≢×+.*∘2)-2*⍨+⌿)÷≢×1⌈¯1+≢)Nv` is valid APL to determine sample variance. To many, this is incomprehensible; but to those who know APL, this makes sense. Citrine does something similar, in that it uses a few icons for repeated things, such as `self` or to indicate that a value is owned by a class and is private. To me, this is different to APL. APL leans heavily on the use of icons as a feature of the language and has created something quite interesting; Citrine's use of icons seems more like a gimmick than a useful feature, and just places undue effort on the user to set up.
2. **Minimal language configurability**
   Citrine only has built-in languages. This means you cannot add in an unsupported language without building the language from source which can

often be quite time-consuming. Citrine also, so far as I can tell, will only allow you to write code in one language and as such requires downloading a language specific binary instead of a general purpose one.

3. **Interpreted not compiled**
   Citrine is an interpreted language. This does reduce the overhead for building the language from scratch, but also requires any computer to have the interpreter for the correct language if you want to run code on it.
4. **General ecosystem**
   The Citrine ecosystem is greatly lacking in any form of documentation of help. There is little documentation and very little to no description of the language beyond what it looks like. No getting started guide to help with the setup, and simple things like passing a `-h` or `--help` flag to the interpreter not giving any help messages. Whilst arguably trivial, I personally see these are large issues for a language with it's first non pre-release version having been released in February 2018.

1.2 How it works

fck works as a general purpose compiler and JIT interpreter through LLVM. It uses a mixture of built-in languages and optional custom languages from language files. There is one binary that handles all the compilation, linting, testing, and documentation. There is a requirement for a `~/.fck` file to specify the default language to be used.

1.2.1 Language files

The language files are simple text files that define a language. These are fairly simple files, and all follow the same format (the specification is defined in Appendix B). All available built-in languages are included by reading their language files at compile time.

The languages all go through a validation process to ensure that:

1. They make sense
2. Comply with some restrictions that make parsing them easier

The first point is mainly just ensuring that there are no repeated keywords or symbols, ensuring that everything is unique. The second point imposes some small restrictions on language files that we can then assume is true when making the lexer. The restrictions are detailed in the specification B.

Finally, one term comes from this: "uppercase hexidecimal digit variants". Part of the language file includes specifying the digits of the language from zero to nine, as well as hexidecimal digits for 10 to 15 (`a` to `f` in English). You may also wan to specify uppercase variants for these digits (`A` to `F` in English). These uppercase digits are referred to as *uppercase hexidecimal digit variants*.

2 Language Design

2.1 Primitives

There are 10 primitives in fck. Each will be explained below.

2.1.1 Integers

There are four integer primitives:

`int` Signed integer. Is as many bits as a memory reference; so on a 32-bit platform this will be 32 bits long, and 64 bits long on a 64-bit platform

`uint` Unsigned version of `int`

`dint` Dynamically sized signed integer

`udint` Unsigned version of `dint`

2.1.2 Floats

There are two types of float:

`float` Floating point number with the same number of bits as an `int` type. Specifically, it's either "binary32" or "binary64" from IEEE 754-2019.

`bfloat` Floating point value stored in base 10. See below for more information

`bfloat` uses two `int` values. The first `int` value represents the value if you removed the decimal point ("mantissa"); with the second `int` representing the base 10 exponent of the value ("exponent"). See Table 1 for examples.

Value	**Mantissa**	**Exponent**
`25.038`	`25038`	`+1`
`0.00196`	`196`	`-3`
`-120.4`	`-124`	`+2`

Table 1: Constituent parts of `bfloat` example values

2.1.3 Strings

There is only one type of string in fck (`str`); it's dynamically sized. There is also a `char` type for singular characters which is formally a "Unicode Scalar Value".

Internally, a `str` type is just a `[char]` type.

2.1.4 Lists

Lists are dynamically sized arrays of the same type of value and use the notation `List<T>` or `[T]`.

Lists can also have a static size using the notation `StaticList<T, N>` or `[T; N]` where `N` is the length of the list.

2.1.5 Booleans

The `bool` type represents a single boolean value. This, internally, takes up 1 byte.

2.2 Pointers and cloning

Types in fck are implicitly make cloneable, that is you can make a clone[1] of any value whenever you want. However, cloning can be expensive sometimes. Cloning can be done implicitly (Listing 1), or explicitly (Listing 2).

```
set value = 1;
some_function(value)
value++
```

Listing 1: Implicit cloning

```
set value = 1;
some_function(value.clone())
value++
```

Listing 2: Explicit cloning

Pointers are passed by prefixing the variable with a `*` in the style of C-like code (Listing 3).

```
set value = 1;
some_function(*value)
value++
```

Listing 3: Implicit cloning

2.3 Mutability

All values are mutable. Only constants are immutable.

2.4 Data Types

This section contains the design for data types:
- Structs
- Enums
- Constant

[1]Clone is used, not copy, to indicate that the cloned value is a copy of the original, but separate from it after being cloned.

- Type aliases

2.4.1 Structs

```
struct StructName {
  properties {
    prop_type a
    prob_type b
  }

  fn add(*self) -> prop_type { a + b }
}
```

Structs are defined using the `struct` keyword, optionally prefixed with either `pub` or `pri` to change the visibility.

2.4.2 Enums

```
enum EnumName {
  variants {
    Variant1 {
      int field1
      float field2
    }
    Variant2(int, [int])
  }
}
```

Enums are defined using the `enum` keyword, optionally prefixed with either `pub` or `pri` to change the visibility. Enum variants are either tuple- or struct-type variants:
Tuple-type `Enum::TupleVariant(field1, field2)`
Struct-type `Enum::StructVariant { field1: value1, field2: value2 }`

2.5 Visibility

There are three levels of visibility in fck (in order):

1. Private using `pri`
2. Restricted using nothing
3. Public using `pub`

To explain these, we'll go through them one by one in the context of a module, and for a data type (struct or enum).

2.5.1 Module level visibility

Consider the following module

```
pub const int CONST1 = 1
const float CONST2 = 1
pri const bool CONST3 = false
```

Within the module, all of the constants are visible. Within the project, only `CONST1` and `CONST2` are visible. Outside of the project, where it's used as a dependency, only `CONST1` is visible[2].

One notable case might be the following module example:

```
pri mod inner {
  pri const int CONST = 1
}

use inner::*
```

This does not expose `CONST` to the module. You cannot increase the visibility of anything, only decrease it; `pub` can become `pri` but not the other way around.

2.5.2 Data types

When defining functions for either a struct or enum the visibility markers mean different things. A `pri` function is only accessible within the struct or enum (not public). A `pub` function is visible everywhere, including outside the project. No visibility marker on a function means it's visible only within the project.

Consider the following module:

```
pub struct MyStruct {
  properties {
    int a
    int b
  }

  pub fn new(int a, int b) -> Self {
    return Self { a: a, b: b }
  }

  fn inc_a(*self) {
    self.a++
  }

  pri fn inc_b(*self) {
    self.b++
  }
}
```

Within the module, project, and externally, we can see `MyStruct` and `MyStruct::new`. Externally, we can't see `MyStruct::inc_a`, but we can see this within the project. `MyStruct::inc_b` cannot be seen anywhere. Private functions for structs and enums can only be used internally. If we added a function to `MyStruct` that looked like this:

[2]This assumes that `CONST1` is exported from the package.

```
fn local_inc_b(*self) {
  self.inc_b()
}
```

this would be okay, and we would be able to use `MyStruct::local_inc_b` from anywhere within the project, but not externally.

2.6 Extensions

Extensions are groups of functions that can be applied to any data type. We use the notation (in this book and in code) `A: B` to indicate that data type `A` is extended by extension `B`; for example, `int: Add<Self>`.

```
extend DataType with Extension {
  /* ... */
}
```

Some useful extensions are:

-
  ```
  /// Convert one value into another using the `as` keyword
  extension Into<T> {
    /// Convert the value after cloning it
    fn into(self) -> T
  }
  ```

  You can pair this with `extend *DataType with Into<T> { /* ... */ }` to prevent implicit cloning before converting one value into another

-
  ```
  /// Format `self` into a string. Used when printing the value or when
  putting it in an f-string
  extension Format {
    /// Format `self` using the provided formatter to return a string
    /// @param(formatter) Requested format to provide
    fn fmt(*self, Formatter formatter) -> str
  }
  ```

2.6.1 Extension function visibility

Extensions are treated as public APIs for the type that's extended by the extension. As such, you can't have a `pri` function in an extension. You can't have a `pub` fn either for the same reason. To use functions from an extension, you need to import the extension first. Note that you don't need to import extensions that get used implicitly (you don't need to import `Into` if you use the `as` keyword for example).

2.6.2 What can go in an extension

Extensions can have:

1. function call signatures
2. complete functions that can be overwritten
3. constant signatures

```
extension ExampleExtension {
  // Constant signature. Must be filled when extending a type with this
extension
  const int EXTENSION_CONST

  // Function signature
  fn func1(*self) -> int

  // Function with a body. This can be overwritten when extending using
this extension, but must keep the same function signature
  fn func2(*self) -> str {
    return <Self as ExampleExtension>::func1(self) as str
  }
```

2.7 Iteration

Iteration is handled in two main ways in fck. The first method is using the `repeat` keyword, and the second is using the `for` and `in` keywords.

2.7.1 Repeat

Repeat statements are a simple way to repeat a block of code any number of times:

```
set reps = 10
repeat reps {
  print("Hello world")
}
```

`repeat` is followed by a variable or literal that tells it how many times it should repeat. You do not have access to a counter of how many times the statement has run without adding one yourself. For this, you can use `for` with a range.

2.7.2 For

For statements iterate over a given value:

```
set primes = [2, 3, 5, 7, 11]
for prime in primes {
  print(prime)
}
```

One (possibly) important thing to note here is that iterating over a list in this way can be quite expensive, since `prime` will be clone of each value in `primes`, requiring each value to be cloned each iteration. In essence, the following it happening:

```
set primes = [1, 2, 5, 7, 11]
for i in 0 to primes.len() {
  set prime = primes[i].clone()
  print(prime)
}
```

If you need to make your code run as efficiently as possible, you can iterate over references instead using the `List::iter` function:

```
extend List<T> {
  pub fn iter(*self) -> Iterator<T> { \* ... *\ }
}
```

where `Iterator<T>: Iterable<*T>`.

3 Lexing

3.1 Introduction

This chapter explains how lexing is handled by the fck compiler. This is a more involved process than for most, and requires a significant amount when compared to much simpler lexers, such as regex-based lexers.

The lexer is very similar to an NFA or nondeterministic finite automata. As such, a good postion of this chapter will be much easier to understand with some knowledge of formal language theory.

This chapter will cover the following key points:

- Generating an NFA from a given language file (Section 3.2)
- Storing and compressing an NFA (Section 3.3)
- The actual process the lexer goes through when lexing an input

This section has one associated appendix: Appendix A.

3.1.1 What is lexing

For those unaware, lexing is the process of taking the raw code (as a byte stream in our case) and turning this into a token stream. A good analogy here is to consider reading a sentence. A sentence, in it's simplest form, is a stream of characters. But we read it as a stream of words. The process of turing characters into words is analogous to turning bytes into tokens when lexing. Tokens are simply more manageble blocks we can split code into that the compiler can deal with much better.

As a very simple example, consider the expression `set a = 5`. This is split into four tokens:

1. `set`: A set keyword token
2. `a`: An identifier token
3. `=`: An equals token
4. `5`: An integer token

This is the goal of lexing, or tokenization, and is the subject of this chapter.

3.2 NFA generation

This section will discuss the generation of the NFA from a given language file.

3.2.1 Initialization

The NFA is initialized with some standard transitions. These are for operators and comparisons only since these are the only things that are constant between languages.

The next thing we add, which is still considered initialization, is brackets. This does not include curly brackets (`{` and `}`) since these are handled differently because they indicate block start and ends. Brackets are not constant because they depend on the direction the language is writen in. If the code is left to right (LTR), then the bytes for square brackets will be 91 and 93 for open and closed brackets respectively. In a right to left (RTL) language, these bytes will be reversed with 93 for open and 91 for closed.

We also initialise an identifier state which we will refer to as q_i. This is initialized here because it will be required when adding in keywords for unmatched keywords to go to, so we need to have a state here to go to.

3.2.2 Keywords

> Whilst we refer to the lexer using an NFA, this is not quite accurate. The lexer uses what we will refer to as a *restricted NFA*. This is the same as an NFA with the only difference being that we have a different definition of the transition function $\delta : Q \to Q \cup \{\emptyset\}$ with the definition for δ^* following from this. The reason for this will be explained later.

To encode the keywords in the NFA, we go through each keyword $k \in K$ (K being the set of all keywords) we need to accept and do the following:

1. Find the longest prefix p of k s.t. $\delta^*(q_0, p) \neq \emptyset$. $p \neq k$ since this would require that $\exists k_1, k_2 \in K$ where $k_1 = k_2$ which is not true
2. For each byte k_n in k where $n > |p|$, make a new state q' with a transition from $\delta^*(q_0, k_{1..n-1})$ to q' requiring k_n (i.e. $\delta^*(q_0, k_{1..n} = q')$

Once complete, the NFA will accept all keywords in K.

However, it will not accept identifiers with prefixes that match prefixes of keywords. Formally, the NFA will not accept any $a = bc \in \Sigma^+$ where $\exists k \in K$ s.t. $k = bd$ for some $d \in \Sigma^+$ and $|b| > 0$. To fix this, we again go through each keyword $k \in K$ and for each $n \in [1, |k|]$, add the transition from $\delta^*(q_0, k_{1..n})$ to q_i for all allowable byte extensions of $k_{1..n}$. This means we allow extensions of $k_{1..n}$ that would be an identifier, and not indicate a different token. For example, we would not allow a transition from $\delta^*(q_0, k_{1..n})$ to q_i for the byte 43, since this is a + and would be either a plus, plus equals, or increment token. This is not done in two separate steps and was only written this way to make this simpler to understand.

3.2.3 Digits

Digits are more complicated that keywords. Digits are specified in language files, and a single digit can be multiple bytes long. This means that we could potentially require several states to recognise a single digit.

Figure 1 gives the general structure of states for accepting a valid digit. You may notice something interesting here; namely that this accepts any string of digits followed by a single `.` as a float without the requirement for a trailing `0`. This is intentional. Consider `set a = 1.` and `set a: float = 1`. Both do the same thing, but differently. The first uses the trailing `.` to indicate a float, whereas the second explicitly states the type of a. Both are valid, but I find the first simpler to write and so included it.

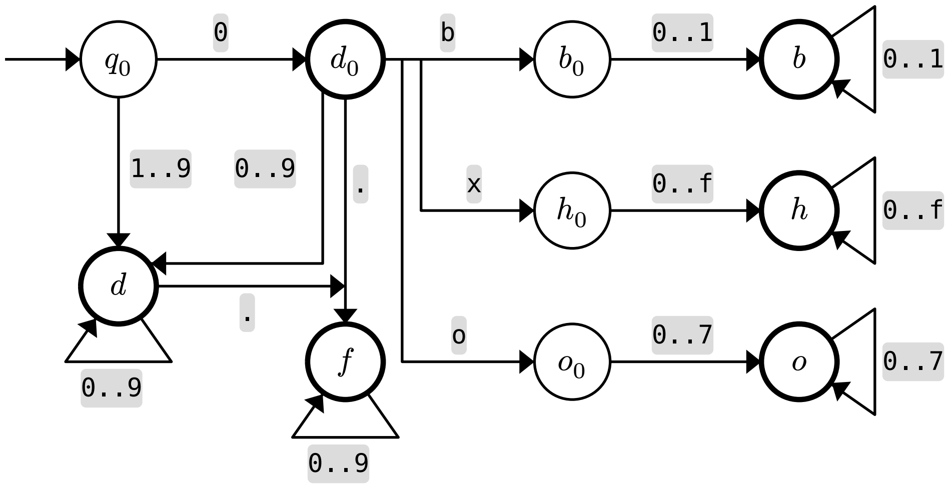

Figure 1: Simplified NFA for parsing digits

> An important note about Figure 1. The labels on the transitions are not absolute, but representative of matching the equivalent of that for some general language. The transitions relating to hexidecimal numbers also accept uppercase letters if they are provided in the language file

The code for this is split into two parts; one where all digits are a single byte long, and one for all other digits. The single digits are relatively simple to add in because we're just following the diagram in it's simplest form. When the digits are multiple bytes long, we need to add several additional states. Specifically, if the digits are all $n + 1$ bytes long, we need an additional $82n$ or $94n$ states, plus any additional bytes for the prefixes for binary, hexidecimal, or octal numbers, depending if the language has uppercase hexidecimal digit variants.

Also, when adding multi-byte digits into our NFA, we need to consider partial matching. Consider the digit $d = ab \in D$ where $a, b \neq \varepsilon$ and $a \notin D$. If we match a with no b, then we have matched an identifier. However, consider another $d' = cd \in D$ where $c, d \neq \varepsilon$ and $c \notin D$. If we matched dc, this would be two separate tokens (matching d and c) since identifiers cannot start with a digit. Thus, for multi-byte digits, we must include a default transition to q_i when matching a digit from q_0 to either d (state) or d_0.

One important note here is that if we match a digit, we only know that we have a digit, and not it's value. Encoding this in an NFA would be practically impossible since we have an upper bound on the number of states in code. Instead, we decode the matched digit by comparing the matched byte stream with the digits bit-by-bit until the whole matched section has been matched with a digit stream. This is then converted to a number (integer of float) depending on the base of the number. This means that for all the digits $d \in D$, it must hold that $\nexists d_1, d_2 \in D$ s.t. $d_1 = d_2 \alpha$ for some $\alpha \in \Sigma^+$. If this were true then when decoding the matched bytes we may match the start of the unmatched section as d_2 when it was actually d_1. This would leave α prefixing the remainder of the unmatched section which could either mean that we cannot match the unmatched section, or that we match an incorrect digit if α is the prefix of some other $d_3 \in D$. Either outcome is incorrect.

3.3 Storing an NFA

The primary way to store an NFA M is to store its *adjacency matrix*. This is an $m \times n$ matrix of 0s and 1s (formally $\{0, 1\}^{m \times n}$) where $m = |\Sigma|$ and $n = |Q|$, that encodes an NFAs states, initial state, transition function, and Σ[3]. We also store F as a set of values from 1 to n inclusive for which the state is accepting.

This is generally good enough. However, it will only allow us to check if some input a is in $\mathcal{L}(M)$. We need a bit more than this. We need to know for a given $a \in \mathcal{L}(M)$ what kind of token matches a. For this reason, we store three matrices all of the same shape. These are as follows:

1. A transition (Δ) table
2. Token type (TT) table
3. Token descriptor (TD)[4] table

> **Notation**:
> We're going to go back to indexing starting at 0 now. We're also going to define $K(r, c)$ as the element in row r and column c for a table K.

[3]Encoding Σ here really means that we assume that Σ has some mapping $m : \Sigma \to [1, |\Sigma|]$ that we have also encoded or is already known

[4]This name may seem strange, and it is a bit. The original meaning has been forgotten, but the term `td` is used so much throughout the codebase that changing it would be more effort than it's worth

The transition table encodes δ. If $\Delta(5, 3)$ was a 9, we would go to row 9. The TT tells us if we've matched a token, and what type of token it is. Using $(5, 3)$ again, if $\mathrm{TT}(5, 3)$ was non-zero, that would mean we've matched a token. In this case, we use $\mathrm{TT}(5, 3)$ and $\mathrm{TD}(5, 3)$ to determine what kind of token has been matched (see Appendix A.2 for the specific TT,TD value pairs).

3.3.1 Table compression

All of the tables are really sparse, or not dense. We formally define density as the proportion of the table that does not have the most common value. More simply for us, the proportion of non-zero elements since zero will be the most common value. Currently, the densities for the English Δ, TT, and TD are 0.67708%, 0.49154%, and 0.47526% respectively[5] which would result in large quantities of data being wasted entirely with only a small amount of useful data. So, we need to compress the tables.

There are a few ways to do this. We'll consider three:
1. Single stream duplicate compression
2. Multi-row group compression
3. Single stream unique compression

To fully explain these, we'll use Table 1 as our sample table to compress. This table has not row numbers or column headings. These aren't necessary for here so they haven't been included.

0	2	3	4	1	0	0
1	0	0	2	0	0	2
0	0	0	4	0	4	0
0	2	3	0	2	0	0
1	0	0	0	3	0	0

Table 1: Sample table for demonstrating compression methods

We will use K to refer to the example table. We will also use examples when exapling the compression methods. Each example will be accessing $K(r, c)$ and we will use r and c to denote general rows and columns.

3.3.1.1 Single stream duplicate compression

This compression method compresses the table into a single row where each row is offset to that when merged only zeros are replaced and duplicates can be merged. To explain, we'll offset the rows:

0 |	0	2	3	4	1	0	0						
4 |					1	0	0	2	0	0	2		
0 |	0	0	0	4	0	4	0						
6 |							0	2	3	0	2	0	0
4 |					1	0	0	0	3	0	0		

Table 2: Offset rows for single stream duplicate compression

[5]The tables each have 208, 151, and 146 non-zero elements out of a total of 30720 elements respectively

You'll notice that in each column, there is either only zeros, or some number of zeros (possible none) and one non-zero value at least once. This means that we can merge these offset rows into a single stream. To do this, we need to save the row offsets as well as a bitmap of the original table. When we try to access an element in this stream, for example $K(3, 4)$ using K to denote our sample table, we first check if the value in the bitmap is 1, indicating that a non-zero element is there in the original table. If it was 0, we would return a 0. Once we've checked that there's a value there, we get the row offset $O(r)$, which is 6 in our example. We then access the element at $O(r) + c$, 10 in our example, to get the element we want.

3.3.1.2 Multi-row group compression

This method doesn't offset rows, but instead groups them into groups of rows that can be merged using the same merging rules as single stream duplicate compression. When applied to our sample table, we get the following groups (with the row index prefixing the row):

```
0 | 0 2 3 4 1 0 0
2 | 0 0 0 4 0 4 0

1 | 1 0 0 2 0 0 2
3 | 0 2 3 0 2 0 0

4 | 1 0 0 0 3 0 0
```

Table 3: Row groups for multi-row group compression

We then merge each group of rows into a single row, saving what new row each original row went into along with a bitmap of the original of table. When getting an element, we first check the bitmap. If we're expecting a value, we then go to the new row for the row we wanted, and then the column we wanted.

I find this method to perform worse than single stream duplicate compression.

3.3.1.3 Single stream unique compression

This method is very similar to single stream duplicate compression, but it does not allow rows to be merged if it would involve merging duplicate non-zero elements. The offset rows for this method would be as follows:

```
 0 | 0 2 3 4 1 0 0
 5 |           1 0 0 2 0 0 2
 4 |         0 0 0 4 0 4 0
11 |                       0 2 3 0 2 0 0
 6 |             1 0 0 0 3 0 0
```

Table 4: Offset rows for single stream unique compression

Notice that each column has either only zeros, or some number of zeros and one non-repeated non-zero value. This method means that we don't have to save a bitmap and can instead use an *origin map*. The origin map tells us which row each value in the compressed stream came from. If we added the origin map onto the offset rows we would have the following:

```
 0 | 0 2 3 4 1 0 0
 5 |           1 0 0 2 0 0 2
 4 |         0 0 0 4 0 4 0
11 |                       0 2 3 0 2 0 0
 6 |             1 0 0 0 3 0 0
     0 0 0 0 0 1 4 2 1 2 4 1 3 3 0 3 0 0
```

Table 5: Offset rows for single stream unique compression with an offset map

> You'll also notice that columns of entirely zeros have an origin of zero. This could be any row in reality since we'd give zero regardless, but having it as zero makes the code slightly quicker.

To access an element such as $K(1, 3)$, get the offset of the row $O(r)$ which is 5 in our example. We then set $c' := O(r) + c$. If $S[c'] = r$ where $S[i]$ is the i-th element in the origin stream, then we return $C[c']$ where C is the compressed stream.

The code for this is as follows (from inside a trait impl):

```rust
fn element(&self, row: u16, col: u8) -> D {
    let index = self.offsets[row as usize] + col as usize;
    if self.origin[index] == row { self.stream[index] } else { *D::ZERO }
}
```

> `D` is a generic here. The trait impl is for a generic `D` which is the return type of the function `element`.

3.3.1.4 The best one

Out of the three, single stream unique compression was by far the best method. It frequently compressed the tables smaller than the others, and had the fastest element access by far. I did not consider a unique variant of multi-row group compression since this method was almost always the worst performing. As such all tables are compressed using single stream unique compression, which we'll now call UStream compression.

When applied to the English tables with original sizes of 122880 bytes for Δ and 61440 bytes for TT and TD, they were compressed down into 2604 bytes for Δ and 2049 for TT and TD which is 4.24% and 3.33% of the original sizes.

> One addition method used to compress the tables is that the types in the tables are quite specific. Token TT and TD values have been designed so that they fit into the range for a *u8*, so we can store the values in TT and TD in a single byte. We also assume that we will never have more than 2^{16} states, so we can store the values in Δ as *u16*s which are smaller that *usize*s which is what we would have previously used.

3.4 NFA table serialization

3.5 Saving pre-compressed tables

Compressing tables is a fairly intensive task when compared to using the tables. As such, we want to limit the number of times we do this. When using custom languages, we have the following file structure under `$FCK/languages`[6]:

```
- languages
  - language_files.fckl
  - comp
    - language_files.bin
```

This directory (`languages`) includes all the custom language files with the `.fckl` extension. Let's assume that one of these language files is `eg.fckl` as an example. The firs time we use `!!eg` in code, the lexer looks for `eg.fckl` in the `languages` directory. If it's not there, then we can't load the language so we get an error. If it is there, the lexer then looks for `eg.bin` in the `comp` directory[7]. If it finds this file, it will read it and deserialize it into the three tables (assuming it doesn't find any errors). If it does not exist, it will compress the language it just read and write the tables to `comp/eg.bin`. It can then parse the input using the tables.

3.5.1 Serialization specification

This section describes the serialization of several types:

- `u8`
- `u16`
- `u32`
- `usize`
- `[T; 256]`
- `Vec<T>`
- `UStream`

Serialization is handled by the following trait

```rust
pub trait SerializeBin {
    fn serialize(&self, out: &mut Vec<u8>);
}
```

where `out` is the buffer being written to with each element being a single byte. This will be referred to as "the buffer" from now on, with "written to the buffer" meaning elements being appended to this vector.

Firstly, the types `u8`, `u16`, and `u32` are all written to the buffer in their respective sizes (2 bytes for `u16` for example). It is up to the deserialization to know what type it's deserializing and take the appropriate number of bytes to do this.

`usize` is serialized by casting it as a `u32` then serializing that. This is done because any serialized `usize` is never expected to exceed `u32::MAX` in this case, but still needs to be kept as a `usize` in code.

[6]**`languages`** is a language specific term. This will depend on the language set in `$FCK/.fck`

[7]`comp` is not language specific since it's only intended for the lexer and anyone wanting to make some strange custom languages

`[T; 256]` and `Vec<T>` are serialized by serializing each element with no break. This is only implemented where `T: SerializeBin`. As with the primitive types, it's up to the deserialization to know what type is being deserialized and act accordingly.

Finally, to serialize `UStream`, we do the following for the stream, origin, then offsets in that order where all the types are vectors:

1. Serialize the length of the vector
2. Serialize the vector by serializing each row for which we know the length

3.5.1.1 Deserialization

Deserialization is fairly simple since we defined the serialization process above. Deserialization is used to read a compressed table (`UStream`) from a binary file. This is done with a deserialization trait

```
pub(crate) trait Deserialize<'a> {
    fn deserialize<T>(s: &mut T) -> Result<Self, String> where
        Self: Sized,
        T: Iterator<Item = &'a str>;
}
```

This is only a `pub(crate)` because it has a public function to deserialize `UStreams` since that's all that needs to be public.

3.6 The actual lexer

Something important to note here, is that although the majority of this chapter has been devoted to describing the NFA we can use to recognise fck, you may have noticed some omissions; namely:

- Comments
- Spaces, tabs, and newline
- Not mentioned but curly braces

These are not handled by the NFA, and in fact, the language is not recognised by an NFA entirely and has a second part.

3.6.1 Language scoping

fck is *language scoped* meaning blocks have languages. For example, consider the following

```
!!en
/* some English code */
!!de
/* some German code */
```

```
!!en
/* back to English code */
```

Here we're imagining that the German code (ln 3-4) was added into a block of English code with the second `!!en` being added in at the end of the German code. However, this may not always be easy. If the first block of English code (ln 2) is quite long, it may be quite tedious to find the current language, and missing out changing back to the previous language would cause the code to be parsed incorrectly[8]. Because of this, the current language is scoped. Instead of the above code, a more sensible addition would be the following

```
!!en
/* some English code */
{
!!de
/* some German code */
}
/* back to English code */
```

Notice how we don't need a second `!!en` since the German is contained within a block, and thus won't change the language of the block above it.

3.6.2 NFA branch parsing

When parsing some input, we parse $\mathcal{L}^*$ where $\mathcal{L}$ is the set of all valid single tokens. We parse input using a maximal-munch tokenization method meaning we always try to find the longest token. This subsection will use formal language theory notation heavily.

Consider some input $i = i_1 i_2 i_3 ... i_n$ and some set of tokens T, along with some matcher $m : \Sigma^+ \to T \cup \{\emptyset\}$. The NFA simply moves through i character by character until it finds some $k \in (1, n]$ s.t. $m(i_{1..k}) \in T$. At this point, we have that $\exists k' > k$ s.t. $m(i_{1..k'}) \in T$ or the inverse, that $\nexists k' > k$ s.t. $m(i_{1..k'}) \in T$. However, we don't know which of these is true, and we want to avoid having to go backwards in i. to deal with this, we create a *branch*. This branch assumes that $\nexists k' > k$ and thus $i_{1..k}$ is the longest match for any k. It then restarts the parsing process from i_{k+1}. We also continue checking if $\exists k' > k$. If we do find some $\exists k' > k$, then we delete the branch and make a new one.

To demonstrate, consider lexing *set*. We start with one main branch. This then gets given an `s`. `s` is a valid identifier, but it could also be longer, so we make a branch. We then give `e` to the main branch. This means the main branch has `se` which is also a valid identifier and so we need a new branch. Since `se` is longer than `s`, we remove the previous branch and replace it with `se`. We then finally give the main branch `t` which means it now has `set` which is a keyword so it makes a new branch

[8]The code would be parsed correctly, but we would consider it incorrect because we missed out changing the language back

as before. We've now reached the end of the input so we traverse through the branches until we find one with no unmatched input and use the tokens it has matched.

> **example here please thanks**

3.6.3 Why we need a second part

Because fck is language scoped, if we wanted to parse the language entirely using an NFA, we would need the NFA to:

1. Recognise when the language has changed
2. Add in the appropriate language to itself
3. Revert back to the previous language when a scope ends

NFAs cannot do this. As such, some tokens are parsed manually; those tokens being those mentioned at the start of this section.

Spaces Spaces are parsed manually because it means we can ignore them. Spaces are only used to separate keywords and identifiers from each other, and newlines indicate the end of an expression, so we can just skip over them, performing the appropriate method each time.

Comments Because spaces are parsed manually, we can't parse anything with a space in it with the NFA. Comments are allowed to contain spaces and thus must be parsed manually.

Curly braces These need to be parsed manually because open(closed) curly braces indicate the start(end) of scopes.

Language changes These have to be parsed manually to allow us to change language when needed.

The reasoning for not parsing spaces may seem a bit strange. So, let's explain it a bit further. Consider the following code

```
set a = 5
!!de
setz b = 9
```

> We're assuming that we start in English. We normally state this a the first line for good practice, but we're not here. This is an exception.

As a reminder, we parse an input using the NFA until it either fails or ends with no branches. If the NFA could parse spaces, then we would parse the entire example input, with the language change to German (ln 2) not being parsed as a language change, but as two `TokType::Not` tokens and an identifier after it. By not parsing spaces, we ensure that we always capture language changes since they must be at the start of a line, so we'd never have started parsing something else before we get to them.

A Tokens

This appendix contains the token enums used by the compiler as well as the TT,TD pairings used to construct tokens.

A.1 Token enums

These are not exact copies; they have all comments and derive macros removed. The specific layout too has been altered to take up less space.

```rust
pub enum TokType {
    Int(u64), Float(f64), Bool(bool), String(Vec<u8>),
    Op(Op), Cmp(Cmp), Increment, Decrement, Set(Option<Op>),
    LParen, RParen,
    LParenCurly, RParenCurly,
    LParenSquare, RParenSquare,
    Label(Vec<u8>),
    Not, Colon, QuestionMark, Dot,
    Identifier(String, Vec<u8>),
    Keyword(u8),
    Comment(String, Vec<u8>),
}

pub enum Op {
    Plus, Minus,
    Mod, Mult,
    Div, Pow,
}

pub enum Cmp {
    Eq, NE,
    LT, LTE,
    GT, GTE,
}
```

A.2 Token (TT, TD) pairs

		TD														
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	255
TT	1	`Bool(true)`	`Bool(false)`	Int base 10	Int base 2	Int base 16	Int base 8	`Float`	`String`	`Char`						
	2	`Op(Op::Plus)`	`Op(Op::Minus)`	`Op(Op::Mod)`	`Op(Op::Mult)`	`Op(Op::Div)`	`Op(Op::Pow)`	`Increment`	`Decrement`	`Not`	`Colon`	`QuestionMark`	`Dot`	`Arrow(Single)`	`Arrow(Double)`	
	3	`Cmp(Cmp::Eq)`	`Cmp(Cmp::NE)`	`Cmp(Cmp::LT)`	`Cmp(Cmp::GT)`	`Cmp(Cmp::LTE)`	`Cmp(Cmp::GTE)`									
	4	`LParen`	`RParen`	`LParenCurly`	`RParenCurly`	`LParenSquare`	`RParenSquare`									
	5	`Set(Some(Op))`														`Set(None)`
	6	`Control Keyword`														
	7	`Data Keyword`														
	8	`Primitive Keyword`														
	9	`Identifier`														
	255	`Comment`														

Table 1: TT and TD value pairs for token types

The `Set` token type contains an `Option<Op>` (TD in `0..6`). This is used to represent operations such as += (`Set(`*`Some`*`(Op::Plus))`) as well as assignment = (`Set(`*`None`*`)`. This is also why operators such as `Increment` are not `Op(Op::Increment)` tokens, since this would indicate the possibility of a set increment operator ++= which does not exist.

The two arrow token types (`Arrow(Single)` and `Arrow(Double)` at (TT, TD) `(2, 12)` and `(2, 13)` respectively) are actually `Arrow(Arrow::Single)` and `Arrow(Arrow::Double)` but have been shortened for compactness.

All the keyword token types take their specific TD value from indexes in the language file specification, which can be found in Appendix B.2[9]. Note that the `Control Keyword` token type is restricted to `0..=17`.

Finally, the `Identifier` and `Comment` token types are indicated as accepting all TD values. This is technically true, since they never check the TD value, but will never see a TD value other than `0`.

The enums for the tokens can be found in

[9]The indexes are one more than the keyword TD value. For example, `else` would have a (TT, TD) of `(6, 5)`.

B Language file specification

This appendix contains the specification for the fckl language format. It is seperated into a numbered list, with the nth element being for the nth line of the language file.

fckl language files do not have the ability to include comments. Blank lines will also cause errors when not specified.

B.1 Terminology

fckl language files use spaces to determine term separation (UTF-8 0x26). This byte may be repeated multiple times between terms.

The term 'character' refers to a Unicode scalar value.

B.2 Specification

1. This line is split into three parts:
   1. An opening curly bracket. This is used to determine if the language is LTR or RTL
   2. The full name of the language
   3. The code for the language. This should match the name of the file with no file extension
2. This is split into three main parts:
   1. Three characters representing the number prefixes for (in order) binary, hexidecimal, and octal
   2. 16 characters for digits from zero to nine followed by the hexidecimal digits for values from 10 to 15
   3. An optional additional six digits for the uppercase variants of the hexidecimal digits for values from 10 to 15
3. Keywords for:
   1. `set`
   2. `and`
   3. `or`
   4. `not`
   5. `if`
   6. `else`
   7. `match`
   8. `repeat`
   9. `for`
   10. `in`
   11. `to`
   12. `as`
   13. `while`
   14. `fn`
   15. `return`
   16. `continue`
   17. `break`
   18. `where`
4. Keywords for:
   1. *`struct`*
   2. `properties`
   3. *`enum`*
   4. `variants`
   5. *`self`*
   6. *`Self`*
   7. *`extension`*
   8. *`extend`*
5. Keywords for:
   1. `int`
   2. `uint`
   3. `dint`
   4. `udint`
   5. `float`
   6. `bfloat`

7. `str`
8. `char`
9. `list`
10. `bool`

Note: See Section 2.1 for descriptions of the primitive types.

6. Constants for `true` and `false`
7. Delimiters for a string and character (string start, string end, character start, character end)

B.3 English language file

The below code is the English fckl language file. This is included to aid with understanding the specification.

The line numbers are not part of the file and are purely to aid with comparing the file with the specification

```
{ English en
b x o 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F
set and or not if else match repeat for in to as while fn return continue
break where
struct properties enum variants self Self extension extend
int uint dint udint float bfloat str char list bool
true false
package name src tests benches type lib app version authors github gitlab
email license description readme homepage repo features dependencies usage
git branch path dev build main
Compiling Building Built Linking Emitted Error errors Warning warnings
e0001 placeholder
e0002 placeholder
e0003 placeholder
e0004 placeholder
e0005 placeholder
e0006 placeholder
e0007 placeholder
e0101 placeholder
e0102 placeholder
e0201 placeholder
e0202 placeholder
e0203 placeholder
e0204 placeholder
e0205 placeholder
e0206 placeholder
e0207 placeholder
e0208 placeholder
e0209 placeholder
e0301 placeholder
e0401 placeholder
e0402 placeholder
fck command line interface
new
```

```
Generate a new project
shell
Run the shell
build
Build the specified project or file
run
Run the specified project after (optionally) building
test
Test the given project using all or some tests
info
Get info about the current fck version
lint
Lint a project depending on the style file
raw
Run a raw piece of fck code
doc
Generate the documentation for a project
translate
Translate a file or project into a target language
help h
Show help information
path p
Path to file or directory
git g
Initialise the new project as a git repository
dump-llvm d
Dump the LLVM IR to a file
no-build n
Don't build before running the command
test t
Path like string to a specific file module or test function to run. Can be
given more than once
raw r
Raw string to run
target l
Language to translate the code into
output o
Path to output the translated file to
comment c
Include the comments in translation using LibreTranslate
```