CMSC330 - 0406

Ash Dorsey
ash@ash.lgbt

Contents

Purpose 4
Assignments 4
Discussions 5
Course Overview 5
Advice 5
OCaml 5
Typing 5
Examples 5
Dynamic type system 5
Static type system 6
Manifest (explicit) typing 6
Latent (implicit) typing 6
Functional Programming 6
Programming Paradigm 6
Features 6
Program state 6
Declarative programming 7
Functions and Expressions 7
Helpful programs 8
Expressions 8
If 8
Lets 9
Making a function 9
Recursion 9
Lists and Pattern Matching 10
Lists 10
List Manipulation 10
Pattern Matching 10
Datatypes 11
Regular Expressions 11
How we describe sets in other contexts 11
Regular expressions meaning 11
Inventing Regex: Product Requirements 11
Intro to Regex 11
Our first regex 11
Example of concatenation 12
Examples of or 12
Examples of precedence (parens) 12
Examples of quantification 12
Bracket Expression 13
ASCII Table 13
More examples of Bracket expressions 13

. operator/Any character 13
^ operator/Start of text 13
$ operator/End of string 13
Negation 13
Escaping 14
Exact vs Partial Match 14
Regex in OCaml 14
Parsing with regex 14
Language-independence of Regex 15
Creating Regex through alternative means 15
Finite State Machines 15
Basics of Finite State Machines 16
FSMs 16
Back to Regex 16
Reviewing Operations 16
Alphabet 16
Concatenation 16
Branching (Union) 17
Big Example of set $\leftrightarrow$ regex 17
Repetition (Kleene closure) 17
What is a regular language 17
Creating the Regex 17
Prove 20
Finite Automata 20
Deterministism 20
DFAs 20
NFAs 20
Regex $\to$ NFA 20
Example 22
NFA $\to$ DFA 23
Intro Examples 23
What is a NFA? 24
What is a DFA? 25
Type of FA 25
NFA $\to$ DFA 25
Underlying Idea 25
ε-closure 25
Move 25
Example 25
Psuedocode 26
Example, by the books 27
Context-free Grammars 28
What can we say about context-free grammar? 28
Parts of a grammar 28
CFG Example 28
Terminals 28
Non-terminals 28
Prove that $4 * 3 - 2$ is a valid sentence 28

Interpreter 29
Lexing in OCaml Example 30
Ambiguous Grammars 31
Parsing in OCaml 31
Review 33
Semantics of a language 34
Operational Semantics 34
Target Language 34
Meta Langauge (English/Public Knowledge) 34
Review 34
Example 34
LOLCODE example 36
Property-Based Testing (PBT) 37
Joke about testing 38
Introducing PBT 38
Typing 38
Type System 38
Type 38
Type checker à la Operational Semantics 39
Example 39
Example 40
Well-typed 40
Undefined semantics 40
Type-safety 40
Soundness 40
Completeness 41
Dynamic type checking 41
Static type checking 41
Goals 41
Generic Typing 41
Subtyping 41
Dependent Typing 41
Liskov Substitution Principle 41
Addition Example 41
Records can Subtype in OCaml 42
Subtyping properties 43
Random note 43
Lambda Calculus 43
Turing Machine 43
Parsing 43
Rules 44
Ambiguous Example 44
Rule #1 44
Operational Semantics 44
Example Application 44
Beta Reduction 45
Rule #2 45
Example 45

Lambda Calculus Continued ... 45
... 47
Garbage Collection (GC) ... 48
Tracing garbage collection ... 49
Mark and Sweep ... 49
Stop and copy ... 49
Tiered data ... 50
Rust ... 50
First Rust Program ... 50
Rust’s Goals for cleaning up memory ... 50
Another rust program ... 51
Semicolons ... 52
What Rust is protecting you from ... 53
Traits in Rust ... 55
String Literals ... 55
Borrowing/References ... 55
Mutability ... 55
Non-lexical lifetimes ... 57
Borrow Checker ... 57
Lifetime Parameters ... 58
Structs and Enums ... 59
Impl in paramater ... 61
Generics ... 61
Smart Pointers ... 62
Security ... 63
Correctness ... 63
Attacks ... 63
Buffer Overflow ... 64
Command Injection ... 64

Purpose

What is a language?

A means of communication.

Why is language important?

They allow you to share your ideas.

How do we design a language?

How do we implement a language?

Explored further later in class.

Is `[->+<]` a sentence?

Maybe, its Brainfuck.

Assignments

Exams (in-person)	12%, 12%	March 5th, April 26th

Projects	3%, 5%, 8%, 8%, 8%, 3%, 5%	January 31st, February 7th
Quizzes	2.5%, 2.5%, 2.5%, 2.5%	February 9th, February 23rd, March 29th, April 26th
Lecture Quizzes	4%	
Final Exam	22%	May 17th

Discussions

- Ungraded
- Coding Exercises
- Project Implementation

Course Overview

Programming languages are like spoken languages: a form of communication to make your ideas work.

Why do so many languages exist? They have different strengths and weaknesses.

Advice

- Ask questions if you are confused
- Make friends, be nice
- Start projects early
- Feel emotions, be in touch with yourself
- Expect to get things wrong
- Look at the notes instead of the slides

OCaml

Typing

Type systems determine what data is and how it's used.

Examples

Java

```
3; // int
1.2; // float
true; // boolean
3 + 1; // 4
10 && 40 // ERROR: bad operand types for binary operator
x + y //-> how to determine it works
```

C

```
3; // int
1.2; // float
true; // ERROR: 'true' does not exist
3 + 1; // 4
10 && 40 // 1
x + y //-> how to determine it works (although it's less strict than Java)
```

Dynamic type system

While the program is running, check the types.

- Java (polymorphic types)
- Ruby and Python

Static type system

Before the program runs, check the types.
- C
- Java (sometimes)

Manifest (explicit) typing

Explicitly telling the compiler the type of new variables.

Java:

```
int x = 3; // explicit
var y = "Hello"; // implicit, but set (so Java doesn't have a manifest typing system)
```

This implies that the types of things are associated with variables

Latent (implicit) typing

Not needing to give a type to a variable.

Python:

```
x = 4
print(x) # 4
y = "hello"
x = True
print(x) # True
```

Types are typically associated with values.

Functional Programming

A $\underbrace{\text{programming paradigm}}_{\text{the classification of programming approach based behavior of code}}$ based on functions.

Certain ways of structuring the solution to a problem. Depending on your language, you will see problems differently. Ocaml enforces functional programming, so you will do things differently.

Programming Paradigm

Typically associated with language features. Many languages have a ton of overlap:
- Python is imperative, object-oriented, and functional
- Ocaml is imperative, object-oriented, and functional

But different languages *emphasize* different paradigms.

Features

- Immutable State
- Functional Programming $\subset$ Declarative Programming
- Referential transparency

Program state

The state of the machine at any given time. Typically in the form of the contents of variables.

Mutable state (C):

```
x = x + 1;
a[0] = 42
```

Imperative programming has mutable state, where it can change and destroy data. It can cause side effects, which are often very hard to reason about.

Example (Java):

```
int count = 0;
int f(Node node) {
  node.data = count;
  count += 1;
  return count;
}
```

This is not a mathematical function because $f(0) = \{0, 1, 2, ...\}$, which is not valid because function outputs must be unique.

No referential transparency: $f(x) + f(x) + f(x) \neq 3 \times f(x)$

Reality Check: There is not necessarily one state because of threading and many other things.

```
int x = 1;
if (fork() == 0)
  x = x + 1;
else
  x = x * 5;
wait(NULL)
printf("x: %d\n", x);
```

Functional Programming uses immutable state:

- Will minimize side effects
- Assumes referential transparency
- Which helps us build correct programs (no unexpected outcomes)

Declarative programming

Imperative programming commands the program to do things (Python):t

```
def evens(arr):
    ret = []
    for x in arr:
        remainder = x % 2
        if remainder == 0:
            ret.append(x)
    return ret
```

Declarative instead declares the result:

```
def evens(arr):
    return [x for x in arr if x % 2 == 0]
```

Functions and Expressions

Our first program!

```
(* hello.ml *)
print_string "Hello world!\n"
```

- It has a comment with parenthesis, and it appears to be typically multiline
- `print_string` uses no parenthesis
- No semicolons (for now)
- Runs from top to bottom without a specific start
- `print_string` does not add a new line on its own
- `print_string` explicitly takes a string instead of being generic in some way

- OCaml is a compiled language

```
$ ocamlc hello.ml
$ ls
a.out  hello.cmi  hello.cmo  hello.ml
$ ./a.out
Hello World!
$
```

Helpful programs

- `ocaml`: repl (Read-eval-print loop) like `python`
- `utop`: better `ocaml`
- `dune`: like Make, tests, compiles, runs
- `opam`: package manager for OCaml

You probably want to run `dune utop src`.

`;;` to end expressions

Expressions

- Everything is an expression
  - Denoted (e)
- Expressions evaluate to values
  - Denoted (v)
- All values are expressions, but not vice versa
- Expressions all have types
  - Denoted (t)

`e: t` means the expression `e` has type `t`:

```
3:int
true:bool
3.1:float

3+1:int
true && false:bool
```

If

OCaml

```
if guard then true_branch else false_branch (* guard must be a bool, true_branch must
have the same type as false_branch, and the resulting type is that shard type *)
```

C

```
if (guard) { // guard is an expression
    true_branch
} else {
    false_branch
}
```

Examples

```
if true then false else true;; (* false:bool *)
if (if true then false else true) then 3 else 4;; (* 4:int *)
1+(if (if true then false else true) then 3 else 4);; (* 5:int *)
2<(1+(if (if true then false else true) then 3 else 4));; (* true:bool *)
if (if true then false else true) then (if 3 > 2 then 5 else 6) else (if false then
1 else 2);; (* 2:int *)
```

```
f true then 3 else 1.2 (*Error: This (1.2) expression has type float, but an expression
was expected of type int *)
```

Lets

Making a function

C

```
void add1(int x) {
  return x + 1
}
return_type func_name(type1 arg1, type2 arg2, ...) {
  body
}
```

OCaml

```
let add1 x = x + 1 (* this is specifically int -> int by virtue of adding the int 1
*)
let func_name arg1 arg2 ... = body
```

Adding `floats`:

```
1. + 2. (* error: expected int *)
1. +. 2. (* 3. *)
```

`float` to `int`:

```
(int_of_float 2.3) + 4 (* 6 *)
```

Comparison requires both values to be the same type:

```
true > false: true
3 > 4: false
false > 3: error: 3 is not an expression of type bool
```

ocaml

```
let cmp x y = x > y : 'a -> 'a -> bool
```

Wow, generics! `'a -> 'a` implies that both of them have the same type!

```
let f x y z = 5: 'a -> 'b -> 'c -> int
```

You can even have different types, pretty cool.

Recursion

```
let rec fact x = if x = 0 then 1 else x * fact (x - 1)

fact 4 = if 4 = 0 then 1 else 4 * fact (4 - 1)
fact 4 = 4 * fact (3)
fact 4 = 4 * ( if 3 = 0 then 1 else 3 * fact (3 - 1) )
fact 4 = 4 * 3 * fact 2
fact 4 = 4 * 3 * ( if 2 = 0 then 1 else 2 * fact (2 - 1) )
fact 4 = 4 * 3 * 2 * fact (1)
fact 4 = 4 * 3 * 2 * ( if 1 = 0 then 1 else 1 * fact (1 - 1) )
fact 4 = 4 * 3 * 2 * 1 * fact (0)
fact 4 = 4 * 3 * 2 * 1 * ( if 0 = 0 then 1 else 0 * fact (0 - 1) )
fact 4 = 4 * 3 * 2 * 1 * 1
fact 4 = 4 * 3 * 2 * 1
fact 4 = 4 * 3 * 2
```

```
fact 4 = 4 * 6
fact 4 = 24
```

Lists and Pattern Matching

Lists

```
[1; 2; 3]: int list
```

The only thing that matters for the type of a list is the datatype it contains. They also have to be homogenous in data type. All the syntax is a series of expressions, semicolon separated, so you can even do stuff like this:

```
[1; int_of_float 4.3; if 1 > 3 then 3 else 6]
```

List Manipulation

Appending

```
[1; 2; 3] @ [4; 5; 6] (* [1; 2; 3; 4; 5; 6] *)
```

Cons operator

```
1 :: [] (* [1] *)
1 :: 2 :: [] (* [1; 2] *)
let f x y = x :: y (* 'a -> 'a list *)
```

Pattern Matching

```
switch (3) {
  case 1: return "a"
  case 2: return "b"
  case 3: return "c"
  default: return "d"
}
```

```
match val:t with
   1 -> "a"
  |2 -> "b"
  |3 -> "c"
  |_ -> "d"
```

```
let get_head lst = match lst with
   [] -> -1
  |head::tail -> head
let add_first_two lst = match lst with
   [] -> -1
  |[x] -> -1
  |h1::h2::tail -> h1 + h2

let rec sum_lst lst = match lst with
   [] -> 0
  |x::xs -> x + (sum_lst xs)
```

```
(1, 2, 3) : int * int * int
(1, false, 3) : int * bool * int
let rot t = match t with (a,b,c) -> (b, c, a) : 'a * 'b * 'c -> 'b * 'c * 'a
let new_func t = match t with (a, b, c) -> (b + 1, c, a && false): bool * int * 'a -
> int  * 'a * bool
```

Datatypes

Regular Expressions

A pattern to describe a set of strings.

How we describe sets in other contexts

Do operations on other sets:

$$\mathbb{R} \cap \mathbb{Z}$$

List items:

$$A = \{1, 2, 3, 4, 5\}$$

Set builders:

$$B = \{x \mid x \in \mathbb{R} \land x > 5\}$$

Complements:

$$C = B^{\complement}$$

Recursively:

$$0 \in D$$
$$x \in D \Rightarrow x + 1 \in D$$

By definition:

$$\mathbb{R} = \text{reals}$$

For regular expressions, we, as a CS community (or rather, we listened to some dead guy), decided on a way to describe sets of strings.

Regular expressions meaning

Regular Expressions are used to describe regular languages (this will be explored further in a future lecture). For now, it is just a tool used to search for text.

Inventing Regex: Product Requirements

A pattern that describes a set of strings

- An alphabet (the symbols in a string)
- Concatenation (combining strings)
- Boolean or
- Precedence
  - We now have both boolean or and concatenation, so now we have to decide precedence
- Quantification
  - How much of a string do we need to use when checking

All can be converted to formal math to prove something.

Intro to Regex

We write patterns, called regular expressions or regex.

Our first regex

```
a
```

What a complicated regular expression!

What does this describe?

Just the single letter a? Yep!

$$/\texttt{a}/ = \{\texttt{"a"}\}$$

What about this one:

```
hello
```

Perhaps “h”, “he”, “hel”, “hell” and “hello”?

Perhaps just “hello”?

Example of concatenation

Yep!

$$/\texttt{hello}/ = \{\texttt{"hello"}\}$$

Through this, we have learned how concatenation is expressed: through putting patterns next to each other.

Examples of or

What about this one:

```
hello|hi
```

Perhaps “helloi” and “hellhi”? Perhaps “hellohi”? Perhaps “hello|hi”? Perhaps “hello” and “hi”? Yep!

$$/\texttt{hello|hi}/ = \{\texttt{"hello"}, \texttt{"hi"}\}$$

Now we know that | is the or operator and its precedence: it goes to the start or end or to the next or operator.

$$/\texttt{this|that}/ = \{\texttt{"this"}, \texttt{"that"}\}$$

$$/\texttt{this|that|the other thing}/ = \{\texttt{"this"}, \texttt{"that"}, \texttt{"the other things"}\}$$

Examples of precedence (parens)

$$/\texttt{cliff|clyff}/ = \{\texttt{"cliff"}, \texttt{"clyff"}\}$$

There are a lot of shared characters there; let’s use parenthesis instead to make it shorter:

$$/\texttt{cl(i|y)ff}/ = \{\texttt{"cliff"}, \texttt{"clyff"}\}$$

Examples of quantification

$$/\texttt{0|1|2|3|4|5|6|7|8|9}/ = \{\texttt{"0"}, \texttt{"1"}, \texttt{"2"}, \texttt{"3"}, \texttt{"4"}, \texttt{"5"}, \texttt{"6"}, \texttt{"7"}, \texttt{"8"}, \texttt{"9"}\}$$

What about all two-digit strings?

$$/\texttt{(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)}/ = \{\texttt{"00"}, \texttt{"01"}, \texttt{"02"}, \ldots, \texttt{"99"}\}$$

But that’s repetitive! Use the quantification operator instead!

$$/\texttt{(0|1|2|3|4|5|6|7|8|9)\{2\}}/ = \{\texttt{"00"}, \texttt{"01"}, \texttt{"02"}, \ldots, \texttt{"99"}\}$$

You can also add bounds:

$$/\texttt{(0|1)\{4,7\}}/$$

That results in 4 to 7 character long strings composed of `0`s or `1`s.

Kleene Operator

The Kleene operator (`*`) repeats 0 or more times:

$$/\texttt{(ha)*}/ = \{\texttt{""}, \texttt{"ha"}, \texttt{"haha"}, \texttt{"hahaha"}, \ldots\}$$

`+` operator

$$/\texttt{a+}/ = /\texttt{aa*}/ = /\texttt{\{1, \}}/$$

`?` operator

$$/\texttt{-?[0-9]+}/ = /\texttt{(|-)[0-9]+}/ = /\texttt{0\{0, 1\}[0-9]+}/$$

Bracket Expression

$$/\texttt{0|1|2|3|4|5|6|7|8|9}/ = /\texttt{[0123456789]}/ = /\texttt{[0-9]}/$$

The hyphen works via the Unicode code point values associated with the characters. For most common characters, this is via ASCII.

ASCII Table

Here is a helpful table of ASCII values:

```
   30 40 50 60 70 80 90 100 110 120
 ---------------------------------
0:    (  2  <  F  P  Z  d    n    x
1:    )  3  =  G  Q  [  e    o    y
2:    *  4  >  H  R  \  f    p    z
3: !  +  5  ?  I  S  ]  g    q    {
4: "  ,  6  @  J  T  ^  h    r    |
5: #  -  7  A  K  U  _  i    s    }
6: $  .  8  B  L  V  `  j    t    ~
7: %  /  9  C  M  W  a  k    u    DEL
8: &  0  :  D  N  X  b  l    v
9: '  1  ;  E  O  Y  c  m    w
```

The most important ones are `"A"` = 65 and `"a"` = 97 and `" "` = 32

More examples of Bracket expressions

$$/\texttt{abcdefghigklmnopqrstuvwxyz}/ = /\texttt{[a-z]}/$$

$$/\texttt{[a-zA-Z]}/ \neq /\texttt{[A-z]}/$$

`.` operator/Any character

Any character.

`^` operator/Start of text

The start of the text.

`$` operator/End of string

The end of the string.

Negation

$$/\texttt{[\^{}aeiou]}/ = \text{all characters that are not a, e, i, o, u}$$

Escaping

If you want to actually use a period or any other operator, you must escape it:

$$/\texttt{-?[0-9]+\textbackslash.[0-9]+}/ \neq /\texttt{-?[0-9]+.[0-9]+}/$$

Exact vs Partial Match

For an exact match, the entire input string must be in the resulting set of the regex.

```
(* let match be a function x y *)
match /[0-9]/ "8" -> true
match /[0-9]/ "13" -> false
match /[0-9]/ "A8" -> false
match /[0-9]/ "A" -> false
```

For partial matches, the input string simply must contain a match.

```
(* let match be a function x y *)
match /[0-9]/ "8" -> true
match /[0-9]/ "13" -> true
match /[0-9]/ "A8" -> true
match /[0-9]/ "A" -> false
```

$$\underbrace{/\texttt{[0-9]}/}_{\text{exact}} = \underbrace{/\texttt{\^{}[0-9]\$}/}_{\text{partial}}$$

$$/\texttt{@gmail\textbackslash.com\$}/ = \text{string that ends with } \texttt{@gmail.com}$$

Regex in OCaml

```
#require "re";; (* include regex library in utop *)

(* How to create a regex: *)
let regex = Re.Posix.re "[0-9]";;

(* compile regex *)
let compiled_regex = Re.compile regex;;

(* convenience function *)
let str2regex s = Re.compile (Re.Posix.re s);;

let r1 = str2regex "[0-9]";;
Re.execp r1 "0" = true;;
Re.execp r1 "A" = false;;
Re.execp r1 "ABC4" = true;;

let r2 = str2regex "^[0-9]$";;
Re.execp r2 "ABC4" = false;;
Re.execp r2 "0" = true;;

let emailre = str2regex "@gmail\.com$";;
Re.execp emailre "umd@gmail.com" = true;;
Re.execp emailre "umd@gmail.comn" = false;;
Re.execp emailre "cliff@gmail.com" = true;;
```

Parsing with regex

Searching is helpful, but parsing is better.

For long-term storage, stuff must be put onto a hard drive. Everything is a text file, so searching is always helpful.

Grouping is helpful to pull data out.

This is continuing from above:

```
(* NOTE: never use regex on phone numbers. They are horrible and are extremely context
sensitive *)
let pn = str2regex "^[0-9]{10}$$";;
let pn = str2regex "^[0-9]{3}[0-9]{7}$";;
let pn = str2regex "^([0-9]{3})[0-9]{7}$";;
(* this has now made a match group, and we can get from that *)

let groups = Re.exec pn "0123456789";;
Re.Group.get groups 1 = "012";;

let pn = str2regex "^([0-9]{3})([0-9]{7})$";;
let groups = Re.exec pn "0123456789";;
Re.Group.get groups 1 = "012";;
Re.Group.get groups 2 = "3456789";;
```

The index of groups is decided by the position of the opening parenthesis. This may be important with nested groups.

Continuing:

```
let kv = str2regex "([a-z]+):([0-9a-zA-Z]+)";;

let getkv s =
  let groups = Re.exec kv s in
  let key = Re.Group.get groups 1 in
  let value = Re.Group.get groups 2 in
  (key, value);;

getkv "password:verysecure123" = ("password", "verysecure123");;
```

Language-independence of Regex

Regular expressions are "language-independent"; it's not specifically about OCaml or any particular language; most languages have support for using them.

Creating Regex through alternative means

Instead of using a string to create a regex, you can manually instantiate one by building it up, piece by piece.

```
Re.Posix.re "I am ([0-9]+) years old"
```

```
Re.seq [Re.str "I am "; Re.group (Re.rep1 Re.digit); Re.str " years old"]
```

These are equivalent statements. The latter might be useful if you are trying to programmatically create a regex because dealing with strings is a pain.

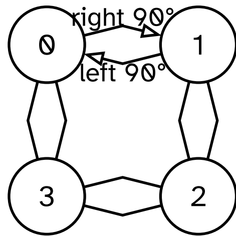
Finite State Machines

Although this may seem like a large leap from regex to finite state machines or FSMs, good regex engines are implemented via finite state machines to ensure linear time complexity with the input size.

This can be an actual real world problem.

Basics of Finite State Machines

Here is an example of a basic Finite State Machine (FSM):



(imagine the labels continue)

Each node (circle) in this FSM represents a state, and each arrow tells you what makes you change state. Here, each state represents a direction, and the arrows represent turns something would make.

Let 0 represent you facing north. As you continue around the FSM, you can tell that 1 represents facing east, 2 represents facing south, and 3 represents facing south. It also is self-consistent because as you traverse the graph, if you go 360°, you will return to the same place you started, and going 90 degrees back also puts you back to where you just were.

Therefore, this is a correct FSM for keeping track of your direction.

FSMs

Each node is a state, and they are finite. This is not the same as Turing machines and can represent fewer problems. However, they are still useful to solve problems

Back to Regex

Regex can be expressed as an FSM because each of its operations can be, and FSMs can be composed.

Reviewing Operations

Fundamentally, regex only needs to support

- An alphabet
  - The set of symbols allowed
- Repetition
- Branching
- Concatenation (combining strings)

Recall that Regex is describing a set of strings. This set of strings is called the “language”. The FSM will then describe that set.

Alphabet

Set of symbols allowed. Denoteed by Σ. A string is then a finite sequence of symbols from Σ

Concatenation

Suppose L_1 and L_2 are languages. Then denote L_1 concatenated with L_2 as $L_1 L_2$.

$$L_1 L_2 = \{xy \mid x \in L_1 \land y \in L_2\}$$

$$\text{Let } L_1 = \{\texttt{"c"}\}, L_2 = \{\texttt{"m"}\}, L_3 = \{\texttt{"s"}\}$$
$$\text{then, } /\texttt{cmsc}/ = L_1 L_2 L_3 L_1$$

Branching (Union)

Suppose L_1 and L_2 are languages. Then denote L_1 or L_2 as $L_1 \cup L_2$.

$$L_1 \cup L_2 = \{x \mid x \in L_1 \vee x \in L_2\}$$

Example of range

$$L_1 = \{\texttt{"a"}\}, L_2 = \{\texttt{"b"}\}, ..., L_{26} = \{\texttt{"z"}\}$$
$$L_1 \cup L_2 \cup \cdots \cup L_{26} = /\texttt{[a-z]}/$$

Big Example of set ↔ regex

$$L_1 = \{\texttt{"a"}\}, L_2 = \{\texttt{"b"}\}, ..., L_{26} = \{\texttt{"z"}\}$$
$$L_1 \cup L_2 \cup \cdots \cup L_{26} = /\texttt{[a-z]}/$$
$$L_{27} = \{\texttt{"0"}\}, L_{28} = \{\texttt{"1"}\}, ..., L_{37} = \{\texttt{"9"}\}$$
$$L_{27} \cup L_2 \cup \cdots \cup L_{26} = /\texttt{[0-9]}/ = /\texttt{\textbackslash d}/$$
$$\therefore \text{ by concatenation:}$$
$$(L_1 \cup \cdots \cup L_{26})(L_{27} \cup \cdots \cup L_{37}) = /\texttt{[a-z][0-9]}/$$
$$((L_1 \cup L_2 \cup L_3)(L_1 \cup L_2 \cup L_3)) \cup (L_1 \cup L_2 \cup L_3) = /\texttt{[abc][abc]|[abc]}/ = /\texttt{[abc]\{1,2\}}/$$
$$L_{38} = \{\texttt{"-"}\}$$
$$(L_{38}(L_{27} \cup \cdots \cup L_{37})) \cup (L_{27} \cup \cdots \cup L_{37}) = /\texttt{-\textbackslash d|\textbackslash d}/ = /\texttt{-?\textbackslash d}/$$
$$(L_1 \cup ... \cup L_n) = /\texttt{.}/, \text{assuming single character languages}$$

Repetition (Kleene closure)

Suppose L_1 is a language. Then, the Kleene closure of L_1 is denoted L_1^*

$$L_1^* = \{x \mid x = \varnothing \vee x \in L_1 \vee x \in L_1 L_1 \vee \cdots\}$$
$$L_1 = \{\texttt{"a"}\}$$
$$L_1^* = /\texttt{a*}/$$
$$L_1 L_1^* = /\texttt{a+}/$$
$$\therefore \text{ you can form other repetitions from this basic one}$$

What is a regular language

Languages created by regex are regular. Regular expressions and FSMs are equivalent in their (limited) power.

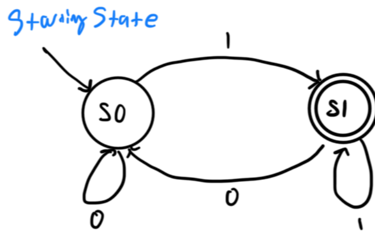
For example:

- FSMs cannot find balanced parentheses
- FSMs cannot describe palindromes

Creating the Regex

Suppose L is a language created by a regex r. If string $s \in L$, then r accepts s.

Goal: create a machine to see if r accepts s. This machine can be represented by a finite state machine.



This FSM has:

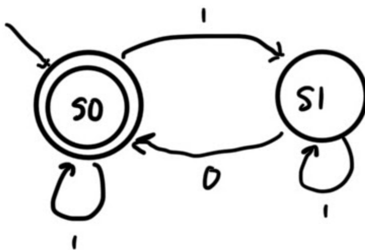
- 2 states: S0, S1
- Initial State: S0
  - Only allowed one
- Accepting State: S1
  - Any number of states (including zero).
- Transitions: 0,1
  - Symbols in the alphabet

Assume this FSM is operating over the binary language. If at S0, if you see a 0, remain at S0, and if you see a 1, go to S1. If at S1, if you see a 0, go to S0, and if you see a 1, remain at S1.

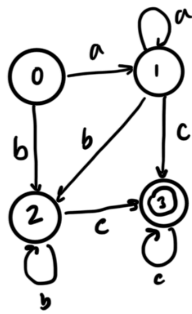
Example string: `00101`. What would this result in? S1, because you go S0 → S0 → S0 → S1 → S0 → S1.

The double circle around S1 represents the fact that the answer is yes; otherwise, you rejected the string. Example string: `10100`. What would this result in? S0, because you go S0 → S1 → S0 → S1 → S0 → S0.

This regex represents any string that ends in 1 or `/(0|1)*1/`.



This new FSM is represented by `/(.*0)?/`. Note the fact that this could be the empty string, as it starts at an accepting state.



Brute force Solution

`/aa*(bb*cc*|cc*)|bb*cc*/`

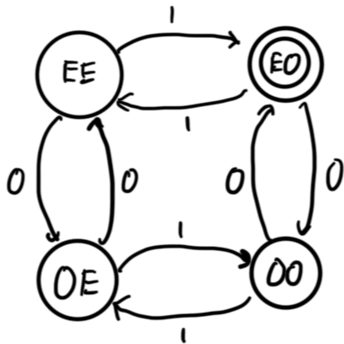
⇔ `/a+(b+c+|c+)|b+c+/`

⇔ `/a+b*c+|b+c+/`

⇔ `/(a+b*|b+)c+/`

This FSM is represented by `/(a+b*|b+)c+/`

Make a state machine for strings with an even number of 0s and odd numbers of 1s:



An FSM is a graph, and tracing through that graph will let you solve a Regex. (FSMs can be used for other things, but we don't care.)
Another example: `/a/`

→ (1) —a→ ((2)) = `/a/` = { "a" }

Prove

Finite Automata

NFAs are DFAs are two different types of FSMs. FA stands for Finite Automata. The N stands for nondeterministic, and conversely, the D stands for deterministic.

Deterministism

If you know the starting conditions, then you know exactly what will happen.

For example, if you throw a pen up, it will always result in the pen coming down, but if you look at an atom, you have no idea where you will observe the electrons to be.

DFAs

Only one path through the FSM.

NFAs

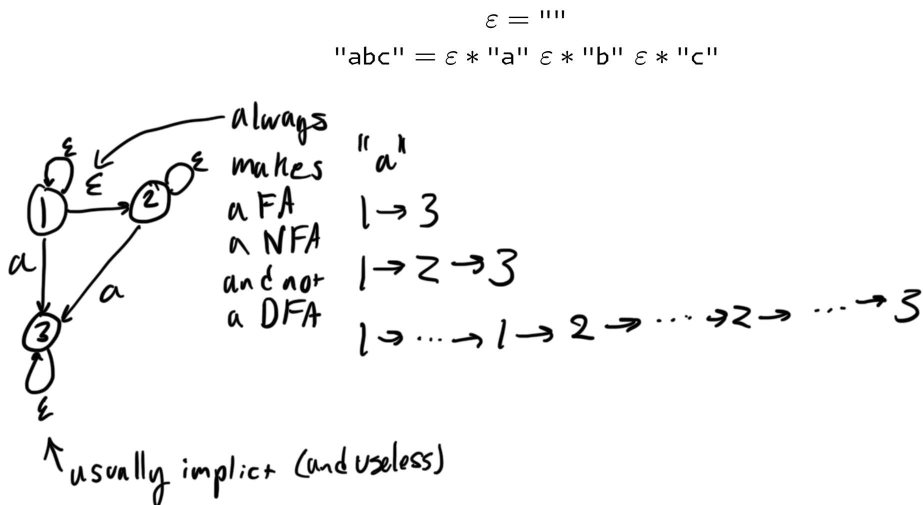
Two or more possible paths through the FSM with the same input.

When at least one possible output is a yes, then the overall machine says yes. Why does the NFA exist? Well, all DFAs are NFAs, and NFAs are much easier to build from a regular expression, so we shall use them.

The plan: $\text{Regex} \to \text{NFA} \to \text{DFA} \to \text{Regex}$

Regex → NFA

$$\varepsilon = \text{""}$$
$$\text{"abc"} = \varepsilon * \text{"a"}\ \varepsilon * \text{"b"}\ \varepsilon * \text{"c"}$$

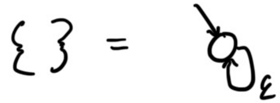


Regex only requires:

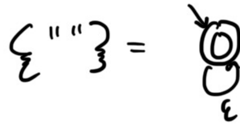
- An alphabet
- Repetition
- Branching
- Concatenation

Therefore, if we can convert these operations into NFAs, then we can make any regex into a NFA.

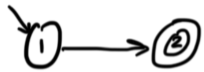
Base case 1: $\varnothing = \{\}$

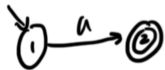


Base case 2: $\{""\} = \{\varepsilon\}$



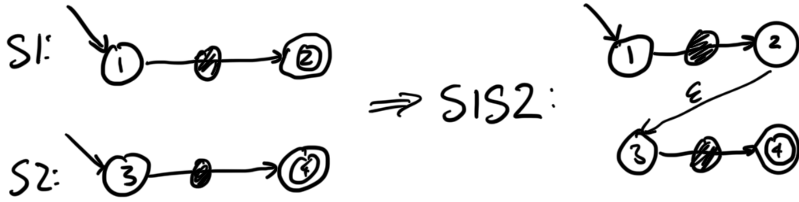
Base case 3: Alphabet

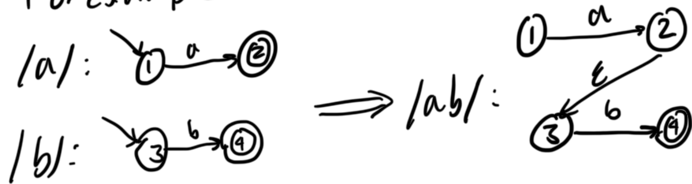




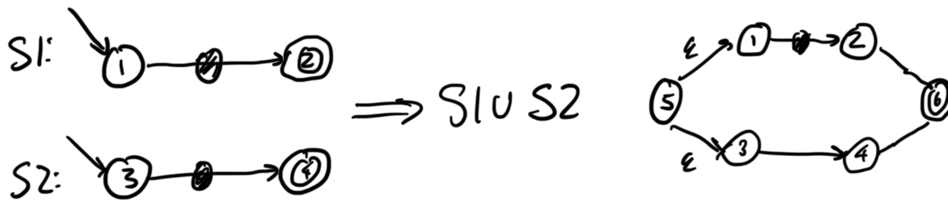
Inductive Hypothesis: We have a machine with one start and one stop state.

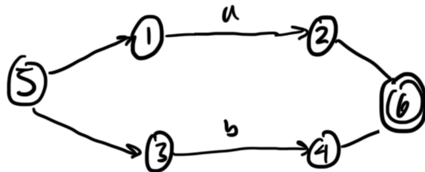
Concatenation:



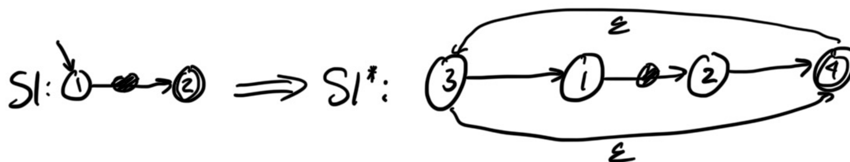


Branching:



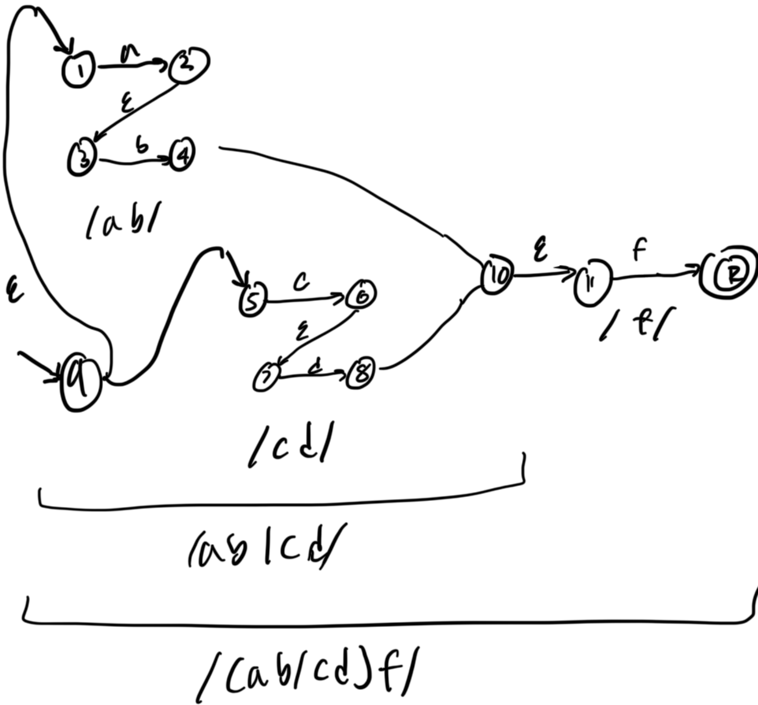


Repetition: Finite repetition is just concatenation and (maybe) branching, so we don't care or need to prove it. We just need to prove infinite repetition (/*/, Kleene Closure).



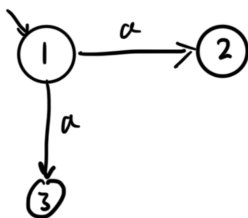
Example

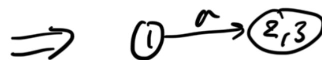
/(ab|cd)f/



NFA $\to$ DFA

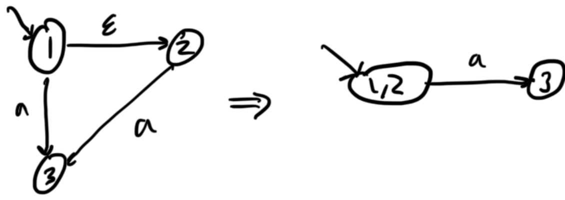
Intro Examples

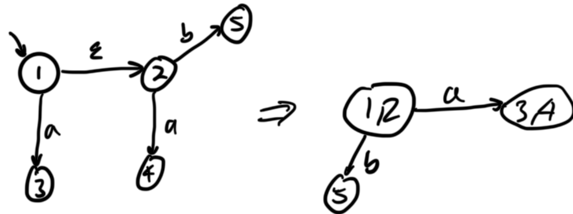


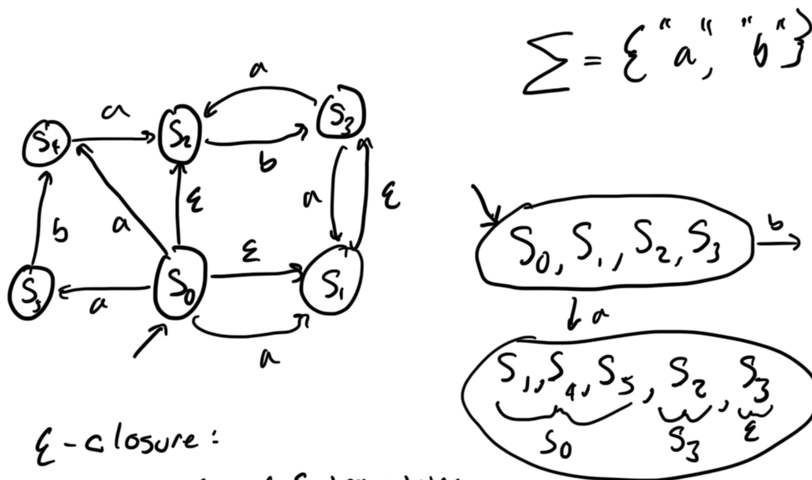


Uncertainty:
50% of 2 or
50% of 3.

Certainty
100% chance of being
in 2 or 3.



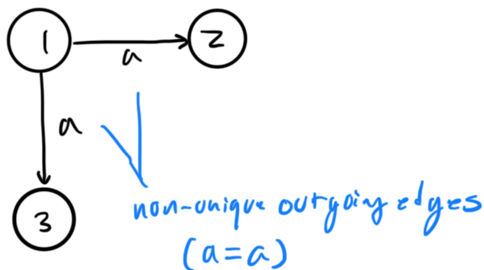


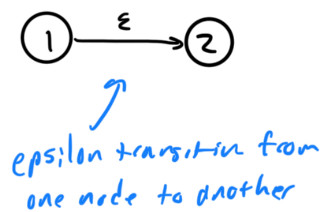


Previously, in CMSC330, we proved that all regexes can be converted into NFAs.

What is a NFA?

A graph that has nodes that have non-unique outgoing edges or there are epsilon transitions between nodes.





What is a DFA?

A graph in which every node has unique outgoing costs. (mentioning epsilon transitions is unnecessary because of the implicit transition)

Type of FA

$$
\begin{gathered}
\text{FSM type} : (\Sigma, Q_s, q_0, F_s, \delta) \\
\Sigma = \text{alphabet set} \\
Q_s = \text{set of states} \\
q_0 = \text{starting state} \\
F_s = \text{set of final states} \\
\delta = \text{transition set} = \text{set of } (\text{source}, \text{cost}, \text{destination}),
\end{gathered}
$$

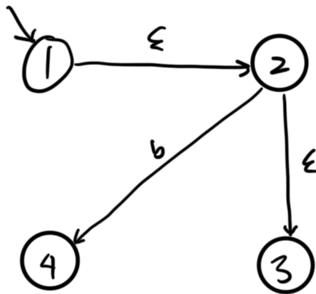
NFA $\to$ DFA

Underlying Idea

Group all possible outcomes as a singular outcome

ε-closure

Answer to the question "Where do I go using only epsilon transitions?"



e-closure of {1}
⇒ {1, 2, 3}

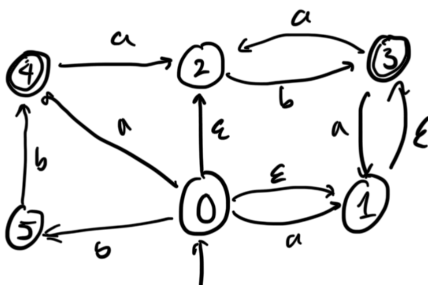
eclosure (S) → S', S ⊆ S'

Move

If I know where to begin and I see an action, where do I end up?

This *does not* include epsilon closure.

Example



DFA:

$$\Sigma = \{a, b\}(\text{NFA}.\Sigma)$$

We will find Q_s, F_s and δ with time.

Start with the starting state: where does 0 correspond to?

$$\text{eclosure}(\{0\}) = \{0,1,2,3\} \Rightarrow q_0 = \{0,1,2,3\}$$

Where can we go from there?

$$\forall x \in \Sigma, y = \text{move}(\{0,1,2,3\}, x), N = \text{eclosure}(y) \Rightarrow N \in Q_s, (\underbrace{\{0,1,2,3\}}_{\text{source}}, \underbrace{x}_{\text{cost}}, \underbrace{N}_{\text{destination}}) \in \delta$$

then simply repeat the process with the newly generated nodes until you finish processing every node.

Let's crack on!

$$\text{eclosure}(\text{move}(\{0,1,2,3\}, a)) = \{1,2,3,4\}$$

$$\text{eclosure}(\text{move}(\{0,1,2,3\}, b)) = \{3,5\}$$

$$\text{eclosure}(\text{move}(\{1,2,3,4\}, a)) = \{1,2,3\}$$

$$\text{eclosure}(\text{move}(\{1,2,3,4\}, b)) = \{3\}$$

$$\text{eclosure}(\text{move}(\{3,5\}, a)) = \{1,2,3\}$$

$$\text{eclosure}(\text{move}(\{3,5\}, b)) = \{4\}$$

$$\text{eclosure}(\text{move}(\{1,2,3\}, a)) = \{1,2,3\}$$

$$\text{eclosure}(\text{move}(\{1,2,3\}, b)) = \{3\}$$

$$\text{eclosure}(\text{move}(\{3\}, a)) = \{1,2,3\}$$

$$\text{eclosure}(\text{move}(\{3\}, b)) = \{\}$$

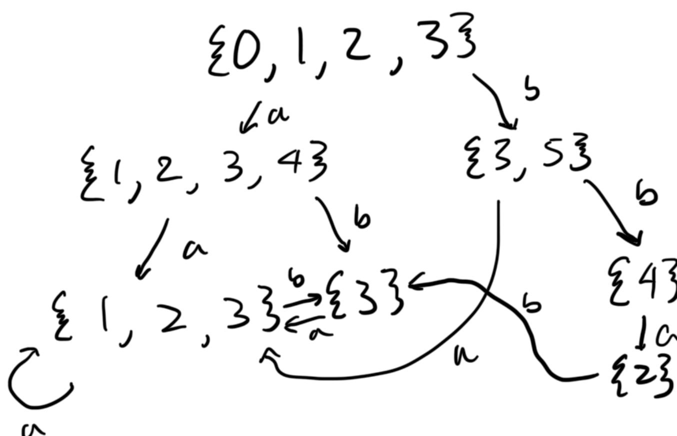
$$\text{eclosure}(\text{move}(\{4\}, a)) = \{2\}$$

$$\text{eclosure}(\text{move}(\{4\}, b)) = \{\}$$

$$\text{eclosure}(\text{move}(\{2\}, a)) = \{\}$$

$$\text{eclosure}(\text{move}(\{2\}, b)) = \{3\}$$

Then just assemble the graph:

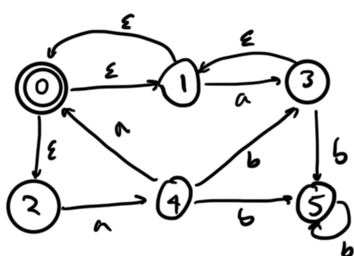


Psuedocode

```
visited = set()
DFA.Sigma = NFA.Sigma
DFA.Qs = set()
```

```
DFA.q0 = eclosure(NFA.q0)
DFA.delta = set()
DFA.delta
DFA.Qs.add(DFA.q0)
while visited != DFA.Qs:
  # add on unvisited state S in DFA.Qs to visited
  S = (DFA.Qs - visited).pop()
  for x in DFA.Sigma:
    y = move(S, x)
    z = eclosure(y)
    if z not in DFA.Qs:
      DFA.Qs.add(z)
    DFA.delta.add((s, x, z))
DFA.fs = {r for r in DFA.Qs if s in r and s in NFA.Fs}
```

Example, by the books



$$\text{DFA}.\Sigma = \{a, b\}$$

$$\text{visited} = []$$

$$\text{eclosure}(\{0\}) = \{0, 1, 2\} \Rightarrow q_0 = \{0, 1, 2\}, \text{DFA}.Q_s = \{\{0, 1, 2\}\}$$

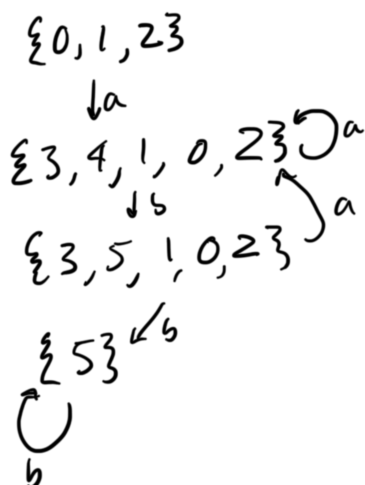
$$\Rightarrow \text{visited} = \{\{0, 1, 2\}\}$$

$$\text{eclosure}(\text{move}(\{0, 1, 2\}, a)) = \{0, 1, 2, 3, 4\}$$

$$\Rightarrow \text{DFA}.Q_s = \{\{3, 4, 1, 0, 2\}\} \cup \text{DFA}.Q_s$$

$$\text{eclosure}(\text{move}(\{0, 1, 2\}, b)) = \{\}$$

$$\vdots$$



Context-free Grammars

What can we say about context-free grammar?

They describe the rules of language.

Grammars have machines that are associated with them. But, unlike ReGex, there is no one-to-one transformation to a machine associated with one.

Grammars:
- can describe “valid” sentences.
- involve syntax (a.k.a. syntactically correct)
- verifies the order of words in a sentence is correct

Succinctly, grammars describe the structure of a sentence. They ensure that sentences are grammatically correct. In doing so, this generates a set of strings.

Parts of a grammar

CFG Example

$$\begin{gathered} E \to E + E \mid E - E \\ E \to M \\ M \to N * M \mid N \\ N \to n \in \mathbb{Z} \end{gathered}$$

Terminals

Things that “end” the grammar, as in there is no more recurison that needs to be done.

In the example above, +, -, * and n are terminals.

Non-terminals

Non-terminals are things that don’t end a definition. Typically, these are capital letters. In the example above, E, M, and N are the non-terminals.

Prove that $4 * 3 - 2$ is a valid sentence

$$\begin{gathered} E \Rightarrow E - E \Rightarrow M - E \Rightarrow N * M - E \Rightarrow 4 * M - E \Rightarrow 4 * N - E \Rightarrow 4 * 3 - E \\ \Rightarrow 4 * 3 - M \Rightarrow 4 * 3 - N \Rightarrow 4 * 3 - 2 \\ \therefore \text{this sentence is grammatically correct, by leftmost derivation} \end{gathered}$$

$$S = \begin{cases} n \in \mathbb{Z} \\ x, y \in S \Rightarrow x + y \in S \\ x, y \in S \Rightarrow x - y \in S \\ x, y \in S \Rightarrow x * y \in S \end{cases}$$

Palindromes cannot work in regex, because regex has a finite amount of memory/memoryless. But CFGs can represent them:

$$S \to aSa \mid bSb \mid cSc \mid a \mid b \mid c \mid \varepsilon$$

This represents all palindromes with the alphabet of $\Sigma = \{a, b, c\}$.

Extrapolating, you can also represent ratios of characters, for example, with balanced parenthesis:

$$S \to b \mid (S)$$

Or with other ratios of characters:

$$S \to aSbb \mid \varepsilon$$
$$\Rightarrow a^x b^y, x = 2y$$

$$S \to aSb \mid aSc \mid \varepsilon$$
$$\#\text{a's} = \#\text{b's} + \#\text{c's}$$

Context-free grammars still have restrictions. For example, they can't represent repeated characters with more than one dependent variable, such as:

$$a^x b^y c^z, x = 2y, z = 3y$$

Interpreter

Our goal is to go from text $\to$ machine code.

In sum, for this we need to

1. Lex
2. Parse
3. Interpret

$$\text{The person ran}$$

To "run" this sentence, first, we need to check if all things in the sentence are valid words.

For example, het sondf dfuy would not meet this step because those are not words.

Secondly, we need to check the structure of this sentence by checking if the words are in the right order.

For example, ran person the would be invalid because the order is wrong.

Thirdly, see if the sentence has meaning. The person ran does because it describes an actual situation.

For example, $\text{Colorless green ideas sleep furiously}$ is grammatically correct and contains real words, but it doesn't have any meaning.

Trying this with OCaml:

```
2 + 3
```

is valid.

```
2 12plus 3
```

is invalid by lexing (first step).

```
+ 2 3
```

is invalid by parsing (second step)

```
2 +. 3
```

is invalid by interpreting (third step)

Your project will be doing these three parts to run a subset of OCaml.

For example,

$$\texttt{let x = 3 in let y = 5 in x + y} \to 8$$
$$\texttt{letter x = 3 in x} \to \text{fail}$$
$$\texttt{let = x 3 x in} \to \text{fail}$$
$$\texttt{let x = 4 in x +. 5} \to \text{fail}$$

Lexing/Tokenizing will convert a string into a list of tokens.

Lexing in OCaml Example

$$E \to E + E \mid E - E \mid n \in \mathbb{Z}$$

For example in this language, our terminals in this language are $+$, $-$, n. You could write this in OCaml like:

```
type token = Plus|Sub|Int of int

lexer "2 + 3" = [Int(2); Plus; Int(3)]
lexer "+ - 3 4" = [Plus; Sub; Int(3); Int(4)]
lexer "2 * 4" (* fails, as * is not in the language *)
```

Note that lexer *does not care* if you don't have a valid grammar.

```
(* string -> token list *)
(* example that you probably don't want to use, but is totally possible to use.
let rec lexer string =
  (* character by character *)
  let charlist = explode string in
  let lex_help char_list = match char_list with
    |' '::t -> lex_help char_list
    |'+'::t -> Plus::(lex_help t)
    |x::t if x = 0 || x = 1 || x = 2 || x = 3 || x = 4 || x = 5 || x = 6 || x = 7 ||
x = 8 || x = 9 -> match lex_help t with
    ..........

*)

let rec lexer string =
  if string = "" then
    []
  else if Re.match string /^(+|-)?([0-9])+/ then
    let num = re.match_group 0 in
    let nlen = len num in
    Int(num) :: (lexer (substring nlen end))
  else if substring string 0 1 = "+" then
    Plus :: (lexer (substring string 1 end))
  else if substring string 0 1 = "-" then
    Sub :: (lexer (substring string 1 end))
  else if Re.match string /^\s+/ then
    let space = len (re.match_group 0) in
    (lexer (substring nlen end))
  else (* anything else *)
    (* raise failure *)
    failwith "this is not valid OCaml"
    (* or skip over value and ignore it *)
    (* "2 * 4" -> [Int(2); Int(4)] | "failure") 
```

You may want to use a map of regexes to tokens to make this less repetitive.

Ambiguous Grammars

Ambiguous grammar example: I saw a dog with a telescope.

This could mean either “I saw a dog and the dog was using a telescope” or “I was using a telescope and saw a dog”.

To deal with these ambigous grammars, you can either

1. Change the grammar to be nonambiguous.
   We will give you non-ambiguous grammar for the project.
2. Change the parser that we will use
   In this class, we will use a LL(1) parser
   Other parsers exist, but we don’t need them for this class
   LL(1) parsers are recursive descent parsers.

Parsing in OCaml

Now for the parser! We should build a tree to represent the sentence structure.

There are two types of trees

1. Parse trees
2. Abstract Syntax Trees

$$E \to A + E \mid A - E \mid A$$
$$A \to B * A \mid B/A \mid B$$
$$B \to \text{sq } B \mid C$$
$$D \to \text{let } var = E \text{ in } E \mid C$$
$$C \to n \in \mathbb{Z} \mid (E) \mid var$$

```
type token =
  | Int of int
  | Plus
  | Minus
  | Star
  | Slash
  | LParen
  | RParen
  | Sq
  | Let
  | Equal
  | In
  | Var of string

(* a project for those at home is to update this to include D as specified about for
the lexer and parser. *)
(* string -> token list *)
let rec lexer input =
  let len = String.length input in
  (* parenthesis are there to capture the value *)
  let numre = Re.compile (Re.Perl.re "^(-?[0-9]+)") in
  let subre = Re.compile (Re.Perl.re "^-") in
  let addre = Re.compile (Re.Perl.re "^\\+") in
  let mulre = Re.compile (Re.Perl.re "^\\*") in
  let divre = Re.compile (Re.Perl.re "^/") in
  let lpre = Re.compile (Re.Perl.re "^\\(") in
  let rpre = Re.compile (Re.Perl.re "^\\)") in
```

```
  let wsre = Re.compile (Re.Perl.re "^(\\s+)") in
  if Re.execp numre input then
    let numgroup = Re.exec numre input in
    let num = Re.Group.get numgroup 1 in
    let numlen = String.length num in
    let numint = int_of_string num in
    Int numint :: lexer (String.sub input numlen (len - numlen))
  else if Re.execp wsre input then
    let wsgroup = Re.exec numre input in
    let ws = Re.Group.get wsgroup 1 in
    let wslen = String.length ws in
    lexer (String.sub input wslen (len - wslen))
  else if Re.execp subre input then
    Minus :: lexer (String.sub input 1 (len - 1))
  else if Re.execp addre input then Plus :: lexer (String.sub input 1 (len - 1))
  else if Re.execp mulre input then Star :: lexer (String.sub input 1 (len - 1))
  else if Re.execp divre input then
    Slash :: lexer (String.sub input 1 (len - 1))
  else if Re.execp lpre input then
    LParen :: lexer (String.sub input 1 (len - 1))
  else if Re.execp rpre input then
    RParen :: lexer (String.sub input 1 (len - 1))
  else failwith "not a valid character"

type ast =
  | Node of int
  | Add of ast * ast
  | Sub of ast * ast
  | Div of ast * ast
  | Mult of ast * ast
  | Square of ast
  | Let of string * ast * ast
  | Id of string

(*
E -> A + E | A - E
A -> B * A | B / A
B -> sq B | C
C -> n | (E)
*)

(* token list -> ast * token list*)
let rec parse_e toks =
  (* token list -> (e tree, rest of tokens) *)
  let atree, arest = parse_a toks in
  match arest with
  | Plus :: t ->
      let etree, erest = parse_e t in
      (Add (atree, etree), erest)
  | Minus :: t ->
      let etree, erest = parse_e t in
      (Sub (atree, etree), erest)
  | _ -> (atree, arest)

and parse_a toks =
  (* token list -> (a tree, rest of tokens) *)
  let btree, brest = parse_b toks in
  match brest with
```

```
  | Star :: t ->
      let atree, arest = parse_a t in
      (Mult (btree, atree), arest)
  | Slash :: t ->
      let atree, arest = parse_a t in
      (Div (btree, atree), arest)
  | _ -> (btree, brest)

and parse_b toks =
  match toks with
  | Sq :: t ->
      let btree, brest = parse_b t in
      (Square btree, brest)
  | _ -> parse_c toks

and parse_c toks =
  (* token list -> (c tree, rest of tokens) *)
  match toks with
  | Int n :: t -> (Node n, t)
  | LParen :: t -> (
      let etree, erest = parse_e t in
      match erest with
      | RParen :: t -> (etree, t)
      | _ -> failwith "missing closing parenthesis, not grammatically correct")
  | _ -> failwith "too few tokens, not grammatically correct"

(* token list -> ast *)
let rec parse toks =
  match parse_e toks with
  | etree, [] -> etree
  | _ -> failwith "extra tokens left over, not grammatically correct"

(* ast -> (string * int) list -> int *)
let rec search var env =
  match env with
  | [] -> failwith "unbound variable"
  | (x, y) :: _ when x = var -> y
  | _ :: t -> search var t

let rec eval ast env =
  match ast with
  | Node x -> x
  | Add (l, r) -> eval l env + eval r env
  | Sub (l, r) -> eval l env - eval r env
  | Div (l, r) -> eval l env / eval r env
  | Mult (l, r) -> eval l env * eval r env
  | Square x ->
      let ex = eval x env in
      ex * ex
  | Let (var, value, body) ->
      let evalue = eval value env in
      eval body ((var, evalue) :: env)
  | Id var -> search var env
```

Review

Lexer: breaks down a string into a list of tokens. If a “word” isn’t valid, it will error.

Parser: It converts a list of tokens to a tree (in our case, an AST). If the “words” don’t form a “sentence” (it checks the order/structure), it will error.

Interpreter $\rightarrow$ Value
Evalulator $\rightarrow$ Value or AST
Compiler $\rightarrow$ Source Code

Semantics of a language

What is it: the meaning.

Etymology/morphology $\rightarrow$ parts of known things create meaning

But ultimately, meaning comes from how something is used.

Dictionaries are typically where people go to know those meanings. Words enter the dictionary through agreement in a society about what something means.

Operational Semantics

Target Language

The language we are describing.

Meta Langauge (English/Public Knowledge)

The language used to describe the target language

From these specifications, you can then create proofs of the correctness of implementations and operations, like $3 + 4 \rightarrow 7$

Review

Semantics is the meaning (of a program).

Operational semantics is the derivation of meaning through how the program operates.

This can be helpful for proofs.

Target language: the language you are describing.

Meta language: the language we are using to describe the target language.

Example

A grammar:

$$E \rightarrow E + E \mid n \in \mathbb{Z}$$

This shows two types of sentences: a plus sentence and a number sentence. Now, we need to make rules to evaluate those sentences:

Rule A

$\therefore n \Rightarrow n$

Rule B

$$\frac{\begin{array}{c} e_1 \Rightarrow v_1 \\ e_2 \Rightarrow v_2 \\ v_3 \text{ is } v_1 + v_2 \end{array}}{\therefore e_1 + e_2 \Rightarrow v_3}$$

Then, you can use this to prove statements:

$$\begin{array}{c} 1 \overset{\text{by rule a}}{\Rightarrow} 1 \\ 2 \overset{\text{by rule a}}{\Rightarrow} 2 \\ 3 \text{ is } 1+2 \\ \hline \therefore 1+2 \Rightarrow 3 \end{array}$$

Another example:

$$\begin{array}{l} \underbrace{6}_{e_1} \overset{\text{by rule a}}{\Rightarrow} \underbrace{6}_{v_1} \\ \underbrace{1}_{e_3} \overset{\text{by rule a}}{\Rightarrow} \underbrace{1}_{v_4} \\ \underbrace{3}_{e_4} \overset{\text{by rule a}}{\Rightarrow} \underbrace{3}_{v_5} \\ \underbrace{4}_{v_6} \text{ is } \underbrace{1}_{v_4} + \underbrace{3}_{v_5} \\ \underbrace{1}_{e_3} + \underbrace{3}_{e_4} \Rightarrow \underbrace{4}_{v_6, v_2} \\ \underbrace{10}_{v_3} \text{ is } \underbrace{6}_{v_1} + \underbrace{4}_{v_2} \\ \hline \underbrace{6}_{e_1} + \underbrace{1+3}_{e_2} \Rightarrow \underbrace{10}_{v_3} \end{array}$$

Now try it with this new grammar:

$$E \to E + E \mid n \in \mathbb{Z} \mid v \in \text{strings}$$

Now rules:

Rule A

$$\frac{}{\therefore A; n \Rightarrow n}$$

Rule B

$$\begin{array}{c} A; e_1 \Rightarrow v_1 \\ A; e_2 \Rightarrow v_2 \\ v_3 \text{ is } v_1 + v_2 \\ \hline \therefore A; e_1 + e_2 \Rightarrow v_3 \end{array}$$

Rule C

$$\frac{A(x) = v}{A; x \Rightarrow v}$$

Now we can prove a statement:

$$\dfrac{\dfrac{A(x) = 5}{A, x:5, y:6, x \Rightarrow 5} \quad \dfrac{A(y) = 6}{A; x:5, y:6, y \Rightarrow 6} \quad 11 \text{ is } 5+6}{A, x:5, y:6;\ \underbrace{x}_{e_1} + \underbrace{y}_{e_2} \Rightarrow \underbrace{11}_{v}}$$

Let's add a new rule and update the grammar:

$$E \to E + E \mid n \in \mathbb{Z} \mid v \in \text{strings} \mid \text{let } x \in E \text{ in } E$$

Rule D

$$\frac{\begin{array}{c} A; e_1 \Rightarrow v_1 \\ A, x : v_1; e_2 \Rightarrow v_2 \end{array}}{A; \text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2}$$

Now, another example:

$$\dfrac{\dfrac{\begin{array}{c} 3 \Rightarrow 3 \\ 4 \Rightarrow 4 \\ 7 \text{ is } 3+4 \end{array}}{A; 3+4 \Rightarrow 7} \quad \dfrac{\dfrac{A(y) \Rightarrow 7}{y \Rightarrow 7} \quad \dfrac{}{1 \Rightarrow 1} \quad 8 \text{ is } 7+1}{A, y:7; y+1 \Rightarrow 8}}{A; \text{let } \underbrace{y}_{x} = \underbrace{3+4}_{e_1} \text{ in } \underbrace{y+1}_{e_2} \Rightarrow 8}$$

LOLCODE example

```
I has a var itz 3
sum of var on 6
```

Rule A

$$\frac{}{A; n \Rightarrow n}$$

Rule B

$$\frac{A; e_1 \Rightarrow v_1 \qquad A; e_2 \Rightarrow v_2 \qquad v_3 \text{ is } v_1 + v_3}{A; \text{SUM OF } e_1 \text{ AN } e_2 \Rightarrow v_3}$$

Rule C

$$\frac{A(x) = v_1}{A; x \Rightarrow v_1}$$

Rule D

$$\frac{A; e_1 \Rightarrow v_1 \qquad A, x : v_1; e_2 \Rightarrow v_2}{A; \text{I HAS A } x \text{ ITZ } e_1 \text{ \\n } e_2 \Rightarrow v_2}$$

Rule E

$$\frac{A; e_1 \Rightarrow v_1 \qquad A; e_2 \Rightarrow v_2 \qquad v_3 \text{ is if } v_1 <> v_2 \text{ then 1 else 0}}{A; \text{DIFFRINT } e_1 \text{ AN } e_2 \Rightarrow v_3}$$

And then we can implement this in OCaml:

```
type ast =
  | Int of int
  | SUM of ast * ast
  | ID of string
  | HAS of string * ast * ast
  | DIFFRINT of ast * ast

let lolinterp tree =
  let rec leval tree env =
    match tree with
    | Int n -> n
    | SUM (e1, e2) ->
        let v1 = leval e1 env in
        let v2 = leval e2 env in
        let v3 = v1 + v2 in
        v3
    | ID x -> search x env
    | HAS (var, e1, e2) ->
        let v1 = leval e1 env in
        let env' = (var, v1) :: env in
        let v2 = leval e2 env' in
        v2
    | DIFFRINT (e1, e2) ->
        let v1 = leval e1 env in
        let v2 = leval e2 env in
        if v1 <> v2 then 1 else 0
  in
  leval tree []
```

Property-Based Testing (PBT)

This is a testing paradigm that works as an extension to unit testing.

Typically, when you make a test, you name it, and then you test a particular input and output:

```
#[cfg(test)]
mod tests {
```

```
    #[test]
    fn a_name() {
        assert_eq!(function(input), expected_output);
    }
}
```

The problem with this is that the programmer must consider the edge cases, which is not usually easy.

Joke about testing

A QA tester walks into a bar. Walks into a bar, runs into a bar, crawls into a bar, dances into a bar, flies into a bar, jumps into a bar. And orders: a beer. 2 beers. 0 beers. 99999999 beers. a lizard in a beer glass. –1 beer. “qwertyuiop” beers.

Testing complete.

A real customer walks into the bar and asks where the bathroom is.

The bar goes up in flames.

Introducing PBT

PBT tries to get around the fact that a programmer needs to write tests and, therefore, may not consider edge cases.

It does so by asking the developer to instead assert properties rather than input-output pairs.

For example, for a reverse function on lists, `x = reverse (reverse x)`, for all lists `x`.

If you can get a program to generate random inputs, you can link the generator to this property and, therefore, have many more cases than you would write by hand.

```
let random_int_lists = generate_int_list 1000 in
fold (fun a x -> reverse (reverse x) = x && a) true random_int_lists
```

1. It is good to test multiple properties.
2. It cannot catch all bugs; it depends on the properties you choose.

(`qcheck` is the OCaml library for this)

Typing

Type System

A type system is a series of rules that dictate
1. What a type is
2. What we can do with types

Type

A category of items that share properties.

For example, for an e

Remember

$$\text{lexer} \to \text{parser} \to \text{evaluator}$$

This was a lie, for many practical purposes

$$\text{lexer} \to \text{parser} \xrightarrow{\text{type checker}} \text{evaluator}$$

The type checker is sometimes part of the evaluator in dynamically typed languages, but it happens before the evaluator in statically typed languages.

Type checker à la Operational Semantics

The type checker rules can be written in a way similar to what we did with operational semantics:

$$\frac{}{\underbrace{G}_{\text{context}} \vdash \underbrace{\text{true}}_{\text{expression}} : \underbrace{\text{bool}}_{\text{type}}}$$

Compare that to:

$$\frac{}{\underbrace{A}_{\text{environment}} \; ; \; \underbrace{\text{true}}_{\text{expression}} \Rightarrow \underbrace{\text{true}}_{\text{value}}}$$

Another rule for type checking, for the other boolean constant:

$$\frac{}{G \vdash \text{false} : \text{bool}}$$

Another, but for integers:

$$\frac{}{G \vdash n : \text{int}, n \in \mathbb{Z}}$$

Now to deal with variables:

$$\frac{G(t) = t}{G \vdash x : t}$$

And for addition:

$$\frac{G \vdash e_1 : \text{int} \qquad G \vdash e_2 : \text{int} \qquad + : \text{int} \to \text{int} \to \text{int}}{G \vdash e_1 + e_2 : \text{int}}$$

Example

$1 + 2$ becomes:

$$\frac{G \vdash 1 : \text{int} \qquad G \vdash 2 : \text{int} \qquad + : \text{int} \to \text{int} \to \text{int}}{G \vdash 1 + 2 : \text{int}}$$

Therefore, you can use the expression $1 + 2$ wherever an int is expected.

Another rule, for if:

$$\frac{G \vdash e_1 : \text{bool} \qquad G \vdash e_2 : t \qquad G \vdash e_3 : t}{G \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3}$$

Another rule, for let:

$$\frac{G \vdash e_1 : b_1 \qquad G, x : t \vdash e_2 : t_2}{G \vdash \text{let } x = e_1 \text{ in } e_2 : t_2}$$

Another rule for functions:

$$\frac{G, x : t_1 \vdash e : t_2}{G \vdash \text{fun } (x : t_1) \to e : t_1 \to t_2}$$

Example

You can build up multiple function calls from a single function call:

$$\frac{}{G, x : t_1 \vdash \text{fun } x \to \text{fun } y \to x + y : t_1 \to t_2 \to t_3}$$

Another rule for function calls:

$$\frac{G \vdash e_1 : t_1 \to t_2 \qquad G \vdash e_2 : t_1}{G \vdash e_1\ e_2 : t_2}$$

Note that $(\text{fun } x \to x + 1)4$ and $\text{let } x = 4 \text{ in } x + 1$ have *very* similar meanings: they both bind the variable x to a particular value, 4.

Well-typed

A program is well-typed if the language's type system accepts the program.

This is well-typed:

```
if true then 1 else 4
```

This is not well-typed and not well-defined:

```
if "hello" then 2 else 5.0
```

This is not well-typed but is well-defined:

```
if true then 1 else "hi"
```

Undefined semantics

An expression e is undefined if the langauge has no semantical definiton of e.

This does not mean that it is an error: $4/0$ will result in an error, but it is defined as that, and so it is not undefined. (It also passes type checking.)

Type-safety

A language is type-safe iff for all well-typed programs p, p is well-defined. Well-defined is **not** well typed.

For example:

```
if true then 0 else "hello"
```

is not well-typed, despite being well-defined.

C is not type-safe:

```
char[4] buff;
buff[3];
```

That program is well-typed but not well-defined because the buffer has uninitialized memory, and that results in undefined behavior according to the C standard.

Soundness

A static analyzer (e.g., type checker) is sound if when it claims a property hold, it does.

Completeness

If when a property holds true for a program, the analysis supports it.

Dynamic type checking

Type checking is done at runtime.

Static type checking

Type checking is done at compile time before the program runs.

Goals

Why are there so many type systems?

We'd like to prove type systems. In doing so, we want to have a sound and complete argument, so we want soundness and completeness in our analysis.

Unfortunately, it is impossible to create a perfect type system by the halting problem. It is impossible to do all three of these:

1. The type checker can halt
2. The type checker is sound
3. The type checker is complete

Typically, languages choose to never halt and then they choose between soundness and completeness.

Let $f : \text{program} \to \text{bool}$ be the function that determines if a program will halt. Create a new function/program called g and halt if g will halt, using f. Now, what is $f(g)$? You can't give an answer, so you can't solve this problem perfectly.

Generic Typing

Such as polymorphism.

Subtyping

If you have some datatype that is a subtype of something else, you can replace the supertype with the subtype, and it will still work.

Dependent Typing

Types of particular properties.

For example, `List.hd: int list -> int` iff the list's length is greater than `0`.

Liskov Substitution Principle

$P(x)$ is a property about objects x of type T. Then, $P(y)$ is a property about objects y of type S, if $S \underbrace{\sqsubseteq}_{\text{subtype}} T$.

$$\forall x \in T, P(x) \Rightarrow \text{if } S \sqsubseteq T, \forall y \in S, P(y)$$

For example, if:

```
class Circle extends Shape { /* ... */ }
```

Then if $P(\text{Shape})$, $P(\text{Circle})$

Addition Example

This is the case in OCaml but it doesn't allow adding floats, as you can in, e.g., C and Java.

$$\frac{G \vdash e_1 : \text{int} \qquad G \vdash e_2 : \text{int} \qquad + : \text{int} \to \text{int} \to \text{int}}{G \vdash e_1 + e_2 : \text{int}}$$

This is an expansion of the property to stuff of the same type. This still is not complete, because you can often do stuff like $4 + 7.3$.

$$\frac{G \vdash e_1 : t \qquad G \vdash e_2 : t \qquad t : t \to t \to t}{G \vdash e_1 + e_2 : t}$$

Using subtyping, you can add this:

$$\frac{\vdash e : t_1 \qquad t_1 \sqsubseteq: t_2}{G \vdash e : t_2}$$

where floats and ints are both subtypes of a larger type, such as number, and therefore can be added by the previous definition.

Then:

$$\frac{G \vdash e_1 : \text{number} \qquad G \vdash e_2 : \text{number}}{G \vdash e_1 + e_2 : \text{number}}$$

$$\frac{\dfrac{}{G \vdash n : \text{int}} \quad e : \text{int}}{G \vdash e : \text{number}} \qquad \frac{\dfrac{}{G \vdash f : \text{float}} \quad e : \text{float}}{G \vdash e : \text{number}}$$

Records can Subtype in OCaml

```
match record1 with
  {x = a} -> (* ... *)
  {y = b} -> (* ... *)
  _ -> failwith "die"
```

Either of these types works for the type of `record1`:

```
type r1 = {x: int; y: int}
type r2 = {x: int; y: int; z: bool}
```

This can be explained with subtyping relations as:

$$\{x: int; y: int\} \sqsubseteq \{x: int\}$$
$$\{x: int; y: int; z: bool\} \sqsubseteq \{x: int\}$$
$$\{x: int; y: int\} \sqsubseteq \{y: int\}$$
$$\{x: int; y: int; z: bool\} \sqsubseteq \{y: int\}$$

For completeness, these are also true:

$$\{x: int\} \sqsubseteq \{x: int\}$$
$$\{y: int\} \sqsubseteq \{y: int\}$$

This works because `{x: int}` is a more general type than its subtypes.

This can be expressed as:

$$\frac{m \geq n}{G \vdash \{k_1 : T_{im}\} : \{k_i : T_i\}}$$

You can also subtype types inside the type:

$$\frac{G \vdash T_i \sqsubseteq P_i}{G \vdash \{k_1 : T_i\} : \{k_i : T_i\}}$$

Subtyping properties

Subtyping relation has some properties.

Let A, B, C be types:
1. Reflexivity: $A \sqsubseteq A$
2. Transitivity: $A \sqsubseteq B \land B \sqsubseteq C \Rightarrow A \sqsubseteq C$

Transitivity is a preorder: pre-ordered relation.

Random note

Modules are libraries that you can define within your own code.

You can create interfaces and then use subtyping with that.

Lambda Calculus

Our goal is to create a model of computation. There are different levels of computation power (what can be computed).

Turing Machine

It is the most powerful model of computation. It can solve any solvable problem.

For example, it cannot solve the halting problem because it is unsolvable. But it can solve anything that can be solved, like sorting a list.

A Turing machine has an infinite “tape” of memory with infinite cells. It has a pointer with an attached state machine that moves the pointer left or right and writes and reads data as input to the state machine.

A Turing complete language is one that can simulate or map to a Turing machine.

Most languages, such as C, Java, OCaml, Rust, Python, and others, are Turing complete.

There are other more nontraditional options that are also Turing complete, like Minecraft Redstone or Magic the Gathering.

Lambda calculus is another Turing complete language that is very minimal.

Parsing

Here is the grammar:

$$\begin{aligned} e \rightarrow x & \qquad x \in \text{variable} \\ \mid \lambda x.e & \\ \mid e\ e & \\ \mid (e) & \end{aligned}$$

Examples of things that work:

$$x$$
$$a$$
$$a\ a$$
$$a\ a\ a$$
$$a\ b\ a$$
$$\lambda x.x$$
$$\lambda x.y$$
$$\lambda y.a\ b$$

Rules

This grammar is ambiguous:

Ambiguous Example

$\lambda x.x\ x\ a$ could be $(\lambda x.x\ x\ a)$, $(\lambda x.x\ x)\ a$, $(\lambda x.x)\ x\ a$

Rule #1

The scope of a function's body extends to the end of the expression or the first unmatched (within the body of the lambda expression) right parenthesis.

Operational Semantics

Free variables (a binding does not exist in A):

$$\frac{}{A; x \to x}$$

Bound variables (a binding exists in A):

$$\frac{A(x) = y}{A; x \to y}$$

Functions:

$$\frac{e \to e'}{A; \lambda x.e \to \lambda x.e'}$$

Reduction:

$$\frac{A; e_1 \to e_3 \qquad A; e_2 \to e_4}{A; e_1\ e_2 \to e_3\ e_4}$$

Function application:

$$\frac{A; e_1 \to \lambda x.e_3 \qquad A; e_2 \to e_4 \qquad A, x : e_4; e_3 \to e_5}{A; e_1 e_2 \to e_5}$$

Example Application

$$\frac{\dfrac{}{A; \lambda x.x \to \lambda x.x} \qquad \dfrac{}{A; y \to y} \qquad \dfrac{A(x) = y}{A, x : y; x \to y}}{A : \underbrace{(\lambda x.x)}_{e_1}\ \underbrace{y}_{e_2} \to y}$$

Beta Reduction

It is the process of calling a function:

$$(\lambda x.x\ y)\ a \to a\ y$$

$$((\lambda x.(\lambda y.x\ y))\ a)\ b \to (\lambda y.a\ y)b \to a\ b$$

After 2 beta reductions, it is now in β-normal form (BNF), which means that the expression cannot be reduced further.

This is also in β-normal form:

$$(a(\lambda x.x))$$

Rule #2

If we have ≥ 3 expressions, evaluate the leftmost two first.

$$a\ b\ c\ d = ((a\ b)\ c)\ d$$

Example

$$(\lambda x.x\ a)\ b\ c \to ((\lambda x.x\ a)\ b)\ c \to ((b\ a)\ c) \to b\ a\ c$$

$$a\ (\lambda x.x\ y)\ b \to (a\ (\lambda x.x\ y))\ b$$

Since $(a\ (\lambda x.x\ y)$ is not a function, it cannot be applied to, so it is already in β-normal forml

$$(\lambda x.\lambda x.x\ y)\ a\ b \to (\lambda x.x\ y)\ b \to b\ y$$

This is the case because of shadowing.

Alternatively, you could have done $(\lambda x.\lambda x.x\ y)\ a\ b \to (\lambda a.a\ y)$ because they are α-equivalent: they represent the same thing, a function call that adds y to your input.

Free variables must the same for α-equivalence. For example, $(\lambda x.x)\ b \not\leftrightarrow (\lambda x.x)a$.

Lambda Calculus Continued

A Turing complete language is one that can map or simulate a Turing machine.

A Turing machine (TM) is a theoretical machine that can solve any solveable problem.

A universal TM (UTM) is a Turing machine that can produce other Turing machines.

Lambda calculus is a minimal Turing complete language.

$$\begin{aligned} e \to\ & x \qquad x \in \text{variable} \\ & \mid \lambda x.e \\ & \mid e\ e \\ & \mid (e) \end{aligned}$$

The scope of a lambda function starts at . and extends right until the end of the expression or until an unmatched) is seen, whichever comes first.

I will use square brackets for the scope:

Example 1:
$\lambda x.x\ x \to \lambda x.[x\ x]$

Example 2:
$(\lambda x.x)\ x \to (\lambda x.[x])\ x$

Example 3:
$(\lambda x.x\ x\ x)\ x\ y \to (\lambda x.[x\ x\ x])\ x\ y$

Example 4:
$(\lambda x.x\ (x\ y)\ z)\ a \to (\lambda x.[x\ (x\ y)\ z])\ a$

Example 5 (scopes can be nested):
$(\lambda x.y\ z\ (\lambda y.b\ y\ (\lambda a.a\ a)\ x\ y)\ u)\ b \to (\lambda x.[y\ z\ (\lambda y.[b\ y\ (\lambda a.[a\ a])\ x\ y])\ u])\ b$

Example 6 (rules can end at the same time):
$\lambda x.a\ b\ (\lambda y.y\ z) \to \lambda x.[a\ b\ (\lambda y.[y\ z])]$

Example 7 (rules don't have to end at the end):
$a\ (\lambda x.\ x\ \lambda y.\ y\ y)\ b \to a\ (\lambda x.\ [x\ \lambda y.\ [y\ y]])\ b$

Lambda calculus is left associative: when given more than 3 expressions, evaluate the leftmost first.

$$a\ b\ c \Rightarrow (a\ b)\ c \Rightarrow ((a\ b)\ c)$$

You can go against that order by adding explicit parenthesis:

$$a\ (b\ c) \neq a\ b\ c$$

This rule can be chained:

$$a\ b\ c\ d \Rightarrow (a\ b)\ c\ d \Rightarrow ((a\ b)\ c)\ d \Rightarrow (((a\ b)\ c)\ d) \Rightarrow$$

Explicit parentheses take precedence:

$$a\ (b\ c)\ d \Rightarrow (a\ (b\ c))\ d \Rightarrow ((a\ (b\ c))\ d)$$

More:

$$\begin{gathered}
a\ (b\ c\ d)\ (e\ f)\ g \\
\Rightarrow (a\ (b\ c\ d))\ (e\ f)\ g \\
\Rightarrow ((a\ (b\ c\ d))\ (e\ f))\ g \\
\Rightarrow (((a\ (b\ c\ d))\ (e\ f))\ g) \\
\Rightarrow (((a\ ((b\ c)\ d))\ (e\ f))\ g)
\end{gathered}$$

Modifying lambda calculus expressions:

2 ways:

β-reduction: calling lambda functions.

If you chain multiple beta reductions together, you can get to β-normal form.

Note that β-normal form is a *property* of an expression that states it cannot be reduced any further, and β-reduction is an *action* performed on an expression.

Example 1:

$$(\lambda x.x\ x)\ a \Rightarrow a\ a$$

Definitionally, β-reduction is taking an expression of the form $(\lambda \text{ var. body}) \text{ param}$ and replacing var anywhere in the body with param.

Keep in mind bindings and shadowing while doing so.

$x\ \lambda x.\ x$ is in beta normal form, even though there is a lambda function.

An example of this is

$$(\lambda x.\ x\ y(\lambda y.\ y)\ x)\ a \Rightarrow\ a\ y\ (\lambda y.\ y)\ a$$

Note that you cannot go any further due to left associativity.

Example of β-reduction:

$$\begin{gathered}(\lambda x.\ (\lambda y.\ x\ y)\ x\ y)\ a\ b \\ \Rightarrow (\lambda y.\ a\ y)\ a\ y\ b \\ \Rightarrow a\ a\ y\ b\end{gathered}$$

Almost the same thing:

$$\begin{gathered}(\lambda x.\ (\lambda y.\ x\ y)\ x\ y)\ (a\ b) \\ \Rightarrow (\lambda y.\ (a\ b)\ y)\ (a\ b)\ y \\ \Rightarrow ((a\ b)\ (a\ b))\ y\end{gathered}$$

You can see how parenthesis can really make a difference.

α-conversion: A process that renames all of the bound variables in a consistent manner.

We can then say two expressions e_1 and e_2 are α-equivalent if e_1 can be α-converted to e_2 or e_2 can be α-converted to e_1.

This can help readability by making shadowing not a problem.

$$(\lambda x.\ x\ \lambda x.\ x\ x) \to (\lambda y.\ y\ \lambda x.\ x\ x)$$

But you need to keep the semantics of the expression the same:

$$(\lambda x.\ (\lambda y.\ x\ y)\ a)\ y \Rightarrow (\lambda y.\ y\ y)\ a \Rightarrow a\ a$$

If we α-convert then reduce:

$$\begin{gathered}(\lambda x.\ (\lambda y.\ x\ y)\ a)\ y \\ \to (\lambda x.\ (\lambda z.\ x\ z)\ a)\ y \\ \Rightarrow ((\lambda z.\ y\ z)\ a) \\ \Rightarrow y\ a\end{gathered}$$

The second one is correct:
- All bound variables should stay bound
- All free variables should stay free

This sometimes makes α-conversion necessary for preserving the meaning of an expression.

$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \Rightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$

Oh no! Infinite loop!

Look at this example:

$$(\lambda x.\ x\ x)((\lambda y.\ y\ a)\ b)$$

This can be evaluated in two different ways:

$$(\lambda x.\ x\ x)((\lambda y.\ y\ a)\ b) \Rightarrow (\lambda x.\ x\ x)(b\ a) \Rightarrow ((b\ a)\ (b\ a))$$

Or

$$((\lambda y.\ y\ a)\ b)\ ((\lambda y.\ y\ a)\ b)) \Rightarrow (b\ a)\ ((\lambda y.\ y\ a)\ b)) \Rightarrow (b\ a)\ (b\ a)$$

The first example is called eager evaluation, and the second is called lazy evaluation.

Now check this out:

$$(\lambda x.\ y)((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))$$

Try evaluating this eagerly (parameters first); you will simply have an infinite loop.

If you have lazy evaluation: y.

If an expression has an infinite loop, we say that an expression has no β-normal form.

If there is no infinite loop, then both eager and lazy evaluations will evaluate to the same β-normal form.

If lambda calculus could do imperative programming, lazy evaluation might run, e.g., a print statement twice, which would not be the same as eager evaluation.

Let's start mapping lambda calculus to calculations, which then can be turned into a Turing machine.

This is called encoding, for example, $0101 : 5$.

You encode/give meaning to the language of things you care about.

Here is an example of an encoding, church encoding:

$$(\lambda x.\lambda y.x) : \text{true}$$
$$(\lambda x.\lambda y.y) : \text{false}$$

Additionally:

$$a\ b\ c : \text{if } a \text{ then } b \text{ else } c$$

For example:

$$\text{if true then false else true} \leftrightarrow (((\lambda x.\ \lambda y.\ x)\ (\lambda x.\ \lambda y.\ y))\ (\lambda x.\ \lambda y.x))$$

$$\text{if true then false else true} = \text{false}$$

$$(((\lambda x.\ \lambda y.\ x)\ (\lambda x.\ \lambda y.\ y))\ (\lambda x.\ \lambda y.x)) \Rightarrow$$
$$(\lambda y.\ (\lambda x.\ \lambda y.\ y))\ (\lambda x.\ \lambda y.\ x) \Rightarrow$$
$$(\lambda x.\lambda y.y) = \text{false}$$

You can use this to then become and use logic from there to build up. You can also make fixed-point loops and then use those to make recursive functions. From there, you can finally make a function like factorial, all in lambda calculus.

Garbage Collection (GC)

It frees data that you no longer need.

Different languages do it differently:
- Automatic garbage collection (OCaml, Java)
- Manual garbage collection (C)
- No garbage collector (Rust)

Garbage collection only acts on the heap, not the stack. This is because on the stack, popping and pushing is already handling getting rid of the garbage.

Languages/runtime systems “know” when to push and pop to the stack.

We know two things:
- the lifetime of stack variables: they stop existing at the end of the scope
- The size of data on the stack

Actions/States	In use	Not in use
Free	Really bad (use-after-free, UB)	Good!
Not Free	Good!	Not ideal: memory leak, you can eventually run out of memory/go to swap and slow down significantly, and you keep memory that may have sensitive data

How can we determine if something is in use?
- We need to take a conservative approach: if you can’t tell, don’t free

We know something is alive if the stack can see it.

Reference counting: Can anything be seen in the data? If not, free it.

In practice, you keep a counter to how many times something can be seen. If another item can see something, you add one to the count for that element. When that something is removed, subtract one.

Problems:
- Cycles will make data leak.
- You have to spend space on storing the counts for every piece of data in the heap.

Tracing garbage collection

Mark and Sweep

Trace through the stack and see what is reachable. Then, trace through the heap and free everything that isn’t reachable.

Problems:
- You have to keep track of all of the different places of memory
- You have to come up with times to stop the entire program, and the runtime is dependent on the number of objects allocated.
- Does not defragment

Stop and copy

Partition the heap into two sections, dead and alive.

When allocating from the stack, allocate into the alive section.

Try to copy everything that is reachable from the stack into the dead section.

Then, swap the names of the sections.

The next time you switch again, you will overwrite the data in the dead section.

This does defragment.

Problems:
- Halves your memory
- You have to stop the entire program

Tiered data

Stuff that has just been allocated is more likely to be deallocated sooner and to be less complex, so you can use a faster method, like reference counting, on that part.

Although you don't have memory safety issues in GC'ed languages, they can be rather slow. Rust seeks to make it both safe and fast.

Rust does this by creating a subset of safe programs that the compiler can determine are safe. As time goes on, the difference in the size of sets between safe programs and programs the compiler can determine are safe shrinks.

Rust

First Rust Program

```rust
// hw.rs
fn main() {
    println!("Hello, World!")
}
```

1. No semicolon on line 3. (because `println!` returns nothing, and `main` also returns nothing (and by nothing I mean `()`, the unit type))
2. `!` as part of `println!`
3. We have a function `main` defined using the `fn` keyword
4. Uses {} for scopes
5. No public keyword on the main function
6. `println!` seems generic: compare to `print_string`/`print_int` in OCaml.
7. No class at the top unlike Java
8. No importing for the `println!`
9. No types on the main function

```
[ash@ashpc rust]$ bat hw.rs
───────┬──────────────────────────────────────────────────────────
       │ File: hw.rs
───────┼──────────────────────────────────────────────────────────
   1   │ // hw.rs
   2   │ fn main() {
   3   │     println!("Hello, World!")
   4   │ }
───────┴──────────────────────────────────────────────────────────
[ash@ashpc rust]$ rustc hw.rs
[ash@ashpc rust]$ ls
hw  hw.rs
[ash@ashpc rust]$ ./hw
Hello, World!
```

Rust's Goals for cleaning up memory

Why is C considered faster than e.g. Java?

- It doesn't have a garbage collector.
  - Like Rust
- It can cast things easily
  - unlike rust (usually, ignoring `bytemuck` and `unsafe`)
- memory arithmetic
  - Rarely useful
- Pointers
  - Rust also has these, safely in the form of pointers
- We can control how much memory we will allocate.
  - Useful when we don't have a lot of memory, like on microcontrollers, IOT devices, and other low-spec devices.
  - Rust allows for this (partially)

Last time, we discussed GCs. They were typically slow and bulky, particularly so for the tracing ones. Rust doesn't want to use a GC; it wants to do something else.

We know that the stack automatically GCs, so can we use that?

For the stack, we need to know:
1. How long does it exist on the stack?
2. How much space does it take up?

The problems with GC's are:
1. Use after free
2. Double free
3. Dangling pointers
4. Memory leaks (not too serious)

Rust wants to prevent these to the best of its abilities. Some it can, some it can't. Regardless, we need rules to terminate if something is safe.

Another rust program

```
fn main() {
    let x = 37; // <- we have let bindings, like in OCaml
    // the type of `x` was guessed via type inference to be `i32` (the default integer
type)
}
```

Rust doesn't have an `int` type, but it does have:
- `i8`/`u8`: signed/unsigned 1 byte of memory
- `i16`/`u16`: signed/unsigned 2 bytes of memory
- `i32`/`u32`: signed/unsigned 4 bytes of memory
- `i64`/`u64`: signed/unsigned 8 bytes of memory
- `i128`/`u128`: signed/unsigned 16 bytes of memory
- `isize`/`usize`: signed/unsigned machine-sized pieces of memory (size of an offset, usually used for indexing into arrays)

And for floats:
- `f32`/`f64`, single and double precision floats (`f16`/`f128` coming soonish)

If statement

```
if true {
    3
} else {
```

```
    4
}
```

1. Uses brackets and has no then keyword
2. Both branches must return the same type

Semicolons

In OCaml, there were very few semicolons: OCaml has expressions (things that evaluate to values), not statements.

Languages that use statements (C, Java) primarily usually use semicolons, and those that use expressions (OCaml) primarily don't.

Rust has both.

`if` is an expression in rust, therefore you can do

```
let y = if true {
    3
} else {
    4
};
```

and then `y` will contain the value `3` after evaluation.

Consider the difference between the type of a statement and the type of an expression: expressions have values, some `T` and statements do not. The type of nothing in OCaml is the unit type (`()`), and Rust is the same.

Rust has code blocks:

```
{ *(statement;)* expression:T?}: T
{ *(statement;)* }: ()
```

For concrete examples:

```
{1; 2; 3}: i32
{1; true; false}: bool
{false; true;}: ()
{println!("hi")}: () // println! returns ()
```

Because `println!` returns `()`, it doesn't matter if you put one at the end of the block. This means that the following also does the same thing:

```
{println!("hi");}: ()
```

If statements must have the same type on both branches, this will not compile:

```
if true {
  3;
} else {
  4
}
```

Without an else branch, `()` is the type of the `if`. Concretely:

This does not compile:

```
if true {
  4
}
```

This does compile:

```
if true {
  4;
}
```

What Rust is protecting you from

```
void main() {
    char* x = malloc(sizeof(char) * 6);
    strcpy(x, "hello");
    char* y = x;
    free(y);
    printf("%s", x); // OH NO! use-after-free
    free(x); // OH NO! double-free
}
```

These errors are problematic and annoying.

Rust solves this by having ownership.

The Three Rules of Ownership

1. Every value has an owner.
2. Each value has one owner.
3. When the owner goes out of scope, the value is freed.

An example of this:

```
fn main() {
    // Create a string on the heap with the value "hello"
    let x = String::from("hello");
    // x is the owner of the "hello" data
    let y = x; // ownership has moved from y to x.
    // y is the owner of the "hello" data.

    // This code would not compile: x has been moved out of
    // y is now the owner; x cannot refer to that memory anymore.
    // println!("{x}");

    // y goes out of scope, so the string is dropped and the "hello" data is freed
     // x also goes out of scope, but because it does not owner anything, nothing
happens
}
```

```
fn main() {
    let x = String::from("hello");

    let y = if false {
        x
    } else {
        String::from("bye")
    };

    println!("{y}")
    // Does not compile, because x _could_ have been moved out of.
    // println!("{x}")
}
```

An example with functions:

```rust
fn main() {
    let x = String::from("hello");
    // x is the owner of "hello"
    // then ownership is passed from `x` to `a` in the function
    let y = f(x);
    // upon function return, y has ownership of f's return.

    // Does not compile: we don't own `x` anymore, `f` does
    println!("{x}");

}

fn f(a: String) -> String {
    // a is the owner of "hello"
    let b = a.len();
    // b owns the result of the len call
    println!("{b}");
    // ownership goes to the caller
    a
}
```

Alternative `f` function:

```rust
fn f(a: String) -> usize {
    // a is the owner of "hello"
    let b = a.len();
    // b owns the result of the len call
    b
    // a is dropped, as it has gone out of scope
}
```

Borrowing:

```rust
fn main() {
    let x = String::from("hello");
    let y = f(&x); // f borrows "hello"
    // upon return, y is also borrowing hello.
    // This does compile:
    println!("{x}");
    println!("{y}");
}

fn f(a: &String) -> &String {
    // `a` is borrowing "hello"
    let b = a.len();
    println!("{b}");
    a
}
```

2 Rules of References

1. We can have only one of these:
    1. We can have > 1 immutable borrows
    2. We can have 1 mutable borrow
2. References must always be valid

```rust
fn main() {
    let x = String::from("hello"); // x owns a "hello" on the heap
    let y = String::from("hello"); // y owns a _different_ "hello" on the heap.
}
```

There is no string interning by default in rust. If you want that, there are crates or you could use `Arc<str>`.

```
fn main() {
    let x = 42; // x is the owner of 42 on the stack
    let y = x; // the value of 42 is copied on the stack and y is an owner of the
new 42.
    println!("{},{}", x, y);
}
```

The above works because `i32` is a copy type. Previous instances that have been moved from remain valid.

The copyness of a type is defined by the `Copy` trait.

Traits in Rust

Traits are contracts or guarantees about something. If a datatype has the copy trait, we expect it to behave in a certain way and have certain properties. This is similar to interfaces in Java.

The `Copy` trait says that instead of ownership being transferred, the value will be copied into another instance.

Other examples of traits are `Clone`, `Drop`, `PartialEq` and `Iterator`.

String Literals

String literals are different from `String`s. String literals are references with a `'static` lifetime (lives forever).

Borrowing/References

If you transfer ownership, you cannot use it anymore. This is not always ideal: it's often better to use references.

References are pointers, but safer. “References are Rust's lingo for a pointer.”

2 Rules of References

1. Only one of the following can be true:
   1. We can have ≥ 0 immutable references
   2. We can have only 1 mutable reference.
2. References must always be valid

Mutability

Mutability is a property of the variable, not the value:

```
let x = 4; // A 4 exists somewhere, but you cannot write data there.
x += 1; // Compile error: x is not mutable

let mut x = 4; // A 4 exists somewhere and you _can_ write data there.
x += 1; // Totally valid to mutate
```

An example with strings:

```
let x = String::from("hello");
x.push_str("bye"); // x does not have write access.

let mut x = String::from("hello");
x.push_str("bye"); // Valid to mutate
```

Another example, but now with references:

```
let y;
{
    let x = String::from("hello");
    y = &x;
} // x goes out of scope
println!("{}", y); // y points to data that was freed, so this is not sound.
// the "borrowed value does not live long enough"
```

On the other hand, this is fine:

```
fn main() {
    let x = String::from("hello");
    let y = &x;
    println!("{},{}", x, y); // this is fine
    // note that println! automatically borrows
}
```

This is also fine:

```
fn main() {
    let mut x = String::from("hello");
    let y = &x;
    println!("{},{}", x, y); // this is fine
    // note that println! automatically borrows
}
```

The mutability of a reference does not rely on the variable binding.

However, this is not fine:

```
fn main() {
    let mut x = String::from("hello");
    let y = &x;
    x.push_str("hello");
    println!("{},{}", x, y); // y cannot be used anymore because you invalidated the
reference by mutating the owner of the data.
}
```

This is not fine:

```
let mut x = String::from("hello");
let y = &mut x;
println!("{},{}", x, y); // y exists as an mutable reference at the same time x is
trying to be used.
```

This is fine:

```
let mut x = String::from("hello");
{
    let mut y = &mut x; // y is a mutable reference to x, you cannot access x through
x anymore
    y.push_str(" world"); // x is mutated through y
} // y goes out of scope, so x is now accessible through x again.
println!("{}", x) // x is valid.
```

This is not fine:

```
let x = String::from("hello");
let y = &mut x; // cannot borrow immutable variable as mutable
println!("{}, {}", x, y);
```

What does this prevent? Data races. Not having readers when you have a writer means that you cannot have data races.

This also prevents use-after-free and dangling pointers.

In sum:
1. `&mut` → `&mut`: First becomes invalid
2. `&mut` → `&`: Not possible/but you can do `&mut` → `&` and `&`, and if the first is a variable, it will become mutable again after the second lifetime ends.
3. `&` → `&mut`: not possible
4. `&` → `&`: both remain valid

Non-lexical lifetimes

```rust
let mut x = String::from("hello");
let y = &x;
println!("{}, {}", y, x);
x.push_str("bye"); // valid because y stopped existing before this line due to NLL
println!("{}", x)
```

If you had assumed that this wouldn't compile because of the above logic, this actually does compile because of NLL. Lifetimes[1] end at the place they are last used.

Borrow Checker

The borrow checker is the name of the part of Rust's type checker that checks if references are correct. It validates that reference and ownership rules are being followed.

All references have a lifetime that is part of it's type:

```rust
let x:     i32 = 32;
let y: &'a i32 = &x;
```

This lifetime, `'a`, tells you how long a reference should live. This lifetime is always implicit (except for `'static`) inside functions, and may be explicit in function declarations.

The `'static` lifetime means that the variable will exist for the entirety of the program.

`'a` is just some lifetime. It doesn't have to exist for `'static`.

Example of lifetime errors:

```rust
fn identity(x: &String) -> &String {
    x
}

fn main() {
    let y;
    {
        let x = String::from("hello");
        y = identity(&x);
        println!("{y}");
    }
    println!("{y}"); // This does not compile. The lifetime of y, 'a is only for the
existence of the owner, `x`, and `x` no longer exists.
    // If this did compile, it would be a use-after-free, which is impossible in safe
rust
}
```

[1] of non-`Drop` types, including mutable and immutable references

Another example

```rust
// This function does not compile, because the rust compiler does not know what lifetime the output should have.
fn longest(x: &String, y: &String) -> &String {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}

fn main() {
    let y;
    let z = String::from("Hello");
    {
        let x = String::from("bye");
        y = longest(&z, &x);
        println!("{y}")
    }
    println!("{y}") // Even after correcting `longest,` rust doesn't know the dynamic values of z and x, so it cannot know that z, the owner of the reference, will exist.

}
```

Lifetime Parameters

Rust has rules to automatically determine lifetime parameters if none are given. That is why the first example *did* compile.

1. Every input reference gets a unique lifetime parameter.
2. If we have one input lifetime and one output lifetime, the output lifetime is constrained to the input lifetime.
3. If any of the input lifetimes are for `self`, and the output has a lifetime, then the output lifetime is constrained to `self`.

This does not cover multiple input references forming one output reference. You may also want to use explicit lifetimes to override the default behavior.

By default, Rust infers this:

```rust
fn longest<'a, 'b>(x: &'a String, y: &'b String) -> &String { ... }
```

The output does not have a defined lifetime, so this does not compile.

This is the corrected version:

```rust
fn longest<'a, 'b, 'c>(x: &'a String, y: &'b String) -> &'c String
where
    'a: 'c,
    'b: 'c,
{
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

(Read `:` as “outlines”, so `'a` outlives `'c` and `'b` outlives `'c`)

Note that in this case, because of variance[2],

```
fn longest<'a, 'b, 'c>(x: &'a String, y: &'a String) -> &'a String { ... }
```

is equivalent to the above.

If `'a` is longer than `'b`, `'a` can be used wherever you expect a `'b`.

Another inference example:

```
fn f(x: &i32, y: &f64, z: bool, w: &String) -> bool
```

The above is equivalent to:

```
fn f<'a, 'b, 'c>(x: &'a i32, y: &'b f64, z: bool, w: &'c String) -> bool
```

If we want to, we can reject Rust's default choices, but that requires explicit lifetime parameters:

```
fn f<'a>(x: &'a i32, y: &'a f64, z: bool, w: &'a String) -> bool
```

Another inference example:

```
fn f(x: &i32, y: f64, z: bool, w: String) -> &bool
```

The above is equivalent to:

```
fn f<'a>(x: &'a i32, y: f64, z: bool, w: String) -> &'a bool
```

You can override this (but why would you):

```
fn f<'a, 'b>(x: &'a i32, y: f64, z: bool, w: String) -> &'b bool
```

Another inference example:

```
fn f(&self, x: &i32) -> &bool
```

The above is equivalent to:

```
fn f<'a, 'b>(&'a self, x: &'b i32) -> &'a bool
```

You can override this:

```
fn f<'a>(&self, x: &'a i32) -> &'a bool
```

Structs and Enums

Here is an example of a struct:

```
struct Rectangle {
    height: u32,
    width: u32,
}
```

This is similar to Java's classes. Note that it does not have inheritance, but it does have something similar to interfaces through traits.

```
let r1 = Rectangle {
    height: 42,
    width: 84
};
let r2 = Rectangle {
    height: 24,
    width: 48
};
```

[2] See the nomicion if you want more details.

```rust
println!("{}", r1.height); // you can access fields of a struct
```

You can create methods and associated functions on a struct:

```rust
impl Rectangle {
    // An example of a method
    // Below, there is a doc-comment.
    /// Returns the area of the rectangle
    fn area(&self) -> u32 {
        self.height * self.width
    }
    // An example of an associated function
    fn new(height: u32, width: u32) -> Self { // Self refers to the type of the object you are implementing on
          Rectangle { height, width } // if you have the same variable names as attributes, you don't need to duplicate it like `height: height`
    }
}

println!("{}", r1.area()); // -> 3444
println!("{}", Rectangle::new(50, 50).width); // -> 50
```

You can implement traits:

```rust
impl Clone for Rectangle {
    fn clone(&self) -> Rectangle {
        Rectangle::new(self.height, self.width)
    }
}

let r4 = r1.clone()
```

Enums are also a thing in rust:

```rust
enum Color {
  Red,
  Blue,
  Green
}

let c1 = Color::Red;
```

Note that structs add types together, and enums can or types.

You can match on enums:

```rust
let out = match c1 {
    Color::Red => 0,
    Color::Blue => 1,
    Color::Green => 2,
};
```

You can use an underscore (_) to ignore all other branches if you don't want to be exhaustive, but matches must be exhaustive.

Enums can contain types:

```rust
enum Pixel {
    Rgb(u32, u32, u32),
    Rgba(u32, u32, u32, u32),
}
```

```rust
let p1 = Pixel::Rgb(0, 0, 255);
let p2 = Pixel::Rgb(255, 0, 255, 0);

match p1 {
  Pixel::Rgba(r, g, b, a) => r + g + b + a,
  Pixel::Rgb(r, g, b) => r + g + b,
}
impl Pixel {
    fn blue_pixel() -> Self {
        Pixel::Rgb(0,0,255)
    }
}
```

You can create types and implement on them in many ways:

```rust
struct User {
    name: String,
    age: u32,
}

impl User {
    // fn name<'a>(&'a self) -> &'a String is what is assumed for the lifetimes
    fn name(&self) -> &String {
        &self.name
    }
    // fn weird(&self, x: &String) -> &String does not compile,
    // because the rust compiler assumes the lifetime comes from self,
    // but in reality, it's coming from x, not self.

    // Further, fn weird<'a>(&'a self, x: &'a String) -> &'a String
    // works, but is suboptimal because it then requires both x and self to have the
same lifetime.
    fn weird<'a>(&self, x: &'a String) -> &'a String {
        x // Does not
    }
}

let u1 = User {
    name: String::from("name"),
    age: 10_000,
}
```

Impl in paramater

```rust
pub trait MyTrait {}

fn myfunc(x: impl MyTrait, y: u32) -> bool {
    false
}
```

This requires `x` to implement `MyTrait`, and you can only call functions that require `x` to be `MyTrait`, or require less.

Generics

```rust
enum MyOption<T: MyTrait> {
    Some(T),
```

```
    None,
}
```

`T` then must implement `MyTrait`.

Smart Pointers

This can be thought of as a struct:

```
struct &[T] {
    ptr: &T,
    len: usize,
}
```

Wrappers of pointers are generally called "smart pointers." A subset of these in Rust are fat pointers, specifically anything that points to an unsized type. These are the unsized types:

- `[T]`
- Any struct containing an unsized type
- `dyn Trait`

There are also smart pointers, like `String`:

```
let s1 = String::from("hello");
struct String {
    // *mut is the "real" rust pointer that allows you to do stuff like you would in
C
    ptr: *mut u8, // -> "hello"
    len: usize, // 5
    cap: usize, // 5
}
```

Typically, smart pointers have the `Deref`, `DerefMut` and `Drop` traits.

The `Deref` traits allow you to use a smart pointer like a real pointer but with more safety and niceness. It has the `deref` function that returns a reference.

The `Drop` trait includes a `drop` function that should deallocate all of the memory the structure owns. This is similar to a destructor in C++.

The most basic example of a smart pointer is `Box`, which is explicitly allocated on the heap. Because it is on the heap, moving data only moves the pointer rather than all the data.

`Box` can also store unsized types (such as trait objects) and is always a constant size. This is useful for recursive data structures.

This would fail:

```
// ERROR: the size of the list cannot be found because finding the size is recursive
enum List<T> {
    Cons(T, List<T>),
    Nil,
}
```

But this does work:

```
// Note the generic: this means it can store any type. See the last lecture for
slightly more information
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}
```

Now to use the list:

```
use List::{Cons, Nil};
let x = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));
```

Note that `Box::new` takes ownership. This can cause problems. For example, you can't do:

```
let y = Cons(5, Box::new(x));
let z = Cons(6, Box::new(x)); // x was moved in the above line.
```

You can add `Clone` to make this possible, but that can be inefficient and requires `T: Clone`. `Rc` can make that process more efficient by not requiring `T: Clone` and not allocating more memory.

`Rc` does this via reference collection. It increments on `clone` and `new`, and decrements on `Drop`. It deallocates when the count reaches zero.

```
use std::rc::Rc;

enum List<T> {
    Cons(T, Rc<List<T>>),
    Nil,
}
```

And then to use it:

```
use List::{Cons, Nil};

let x = Rc::new(Cons(1, Rc::new(Cons(2, Rc::new(Cons(3, Rc::new(Nil)))))));
let y = Cons(5, x.clone());
let z = Cons(6, x.clone());
drop(x);
// I can still use `x` and `y`.
```

Security

Yesterday, China hacked the UK's Ministry of Defense, gaining access to the armed forces' personal details. Change Healthcare, a UnitedHealth subsidiary, was attacked by ransomware, which interrupted payment processing. Equifax's data was breached, leaking tons of PII (personally identifiable information).

PII includes Social Security Numbers, credit cards, addresses, payment history and more.

It would be good if these things didn't happen. How can we prevent that? Through security.

We want only authorized users to have access to this information. This means ensuring correctness.

When creating a secure product, you should always assume an *active* and *malicious* adversary.

Testing doesn't solve this problem; it just shows that some inputs don't result in bugs.

Correctness

Does the program behave the way you want it to?

Attacks

Attacks want to:

- Break confidentiality (unauthorized reads)
  - Buffer overflows, etc
- Break integrity (unauthorized writes)

    - Ransomware
- Break availability
    - DDOS

To prevent attacks:
- Make attacking harder
- Make attacking more expensive
- Thereby lowering the ROI

Attacks are common because
- There are millions of lines of code
    - there will be bugs
- the barrier to entry for attacks is very low
- Normal users don't try to find bugs
- Fixing bugs costs time and money

Our goal is to minimize undesired behavior:
- Think like attackers
- Address bugs and design flaws
- Understand both the software and hardware problems

Buffer Overflow

A family of vulnerabilities stems from going out of the intended bounds of an array.

```
void other() {
    char password[5];
    char username[5];
    // ...
}
```

In this code, both the password and the username are on the stack, and if you index past `username`, you will hit the `password`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void other() {
    char password[5];
    char username[5];
    strcpy(password, "pass");
    gets(username); // !!! you can now easily buffer overflow into the password
    // ...
}
```

This was the basis for the Heartbleed OpenSSL vulnerability. Enforcing type safety prevents this.

Command injection cannot be prevented with type-safety.

Command Injection

Code as part of user input is evaluated accidentally.

```
import os
a = input("enter a file to cat: ")
os.system("cat " + a)
```

If you input, for example, `; rm–rf ––no-preserve-root /`, that would delete all files on the computer because the semicolon separates commands.

You should instead actually use the correct methods for this that allow you to specify arguments directly rather than going through a shell.