CMSC330

Hari

Contents

Chapter 1

Introduction

1.1 Introduction

- Professor Kauffman
- kauffman77@gmail.com
- Office Hours Tue/Wed 1–2pm in IRB2226
- Synced with 1xx and 2xx sections

1.2 Programming Language Definitions

Programming languages can follow two different typing disciplines. Either they are

Definition 1 (Static)**.** Static typing means that variables are bound to the same type of data over their lifetime.

or

Definition 2 (Dynamic)**.** Variables may be bound to data of differing types over their lifetime.

Definition 3 (Explicit)**.** Explicit typing (or manifest typing) means that we must specify the type when assigning a variable.

Definition 4 (Implicit)**.** Implicit typing (or latent typing) means that we do not have to specify the type, so the compiler can infer types from expressions.

They can also broadly fall into two paradigms.

Definition 5 (Imperative Paradigm)**.** Emphasizes variables with values that change over time, similar to a Turing Machine.

Definition 6 (Functional Paradigm)**.** Favors immutable data, bindings are fixed but new versions can be created, similar to Lambda Calculus

1.3 Object Orientedness

The real question we have to be asking is **what makes a language object oriented** and **what is object orientedness?**.

The answers to these questions are a little complicated, as object orientedness does not have a small easy definition. The best way to distinguish is through dynamic dispatch.

Definition 7 (Dynamic Dispatch)**.** Automatically selecting and executing one of several versions functions based on the type of data. This occurs with **polymorphism**, where you can change the definition of a function with subclasses. This package is known as a method.

Another distinguishing feature is the use of polymorphism and class hierarchies with inheritance.

However, object oriented programming comes with a grave flaw, seen in the Kingdom of Nouns.

In this class we will be looking at 4 languages. Two imperative as reference: Java and Python. And the two new ones are OCaml and Racket, which are functional languages!

> "These are your father's parens, elegant weapons from a more civilized time."
>
> – xkcd

1.4 PL Theory and Tech Introduction

A glimpse into the future:

Finite state machines are models in which formal languages can be recognized, and can be used to build implementations of regular expressions.

Deterministic Finite State Automata needs each transition to be uniquely determined by its source state and input symbol and reading an input symbol is required for each transition.

Nondeterministic Finite State Automata relaxes those restrictions and can be smaller. Therefore, we can multiple transitions which are accepted, and if any of the possible paths are accepted, then the string is accepted.

Regular expressions are used for complex string matching and replacement.

Lexing and Parsing involves processing the raw input text of a program into a data structure, which can be interpreted or compiled to assembly language. The input text is transformed into tokens which is then converted to an abstract syntax tree. We can hand roll code for this or use `Lex` or `YACC` to automate it.

Chapter 2

Python

2.1 Logistics

Reading: here

2.2 History

- Development started in late 1980s
- Version 2 released in 2000
- Version 3 released in 2008, NOT backwards compatible
- Comes with a lot of bells and whistles builtin

2.3 Stuff

- Comments
- Statements
- Variable Types
- Assignment
- I/O
- Functions
- Conditionals
- Iterations
- Aggregate data such as arrays, maps
- Library System

2.4 Example

```python
# a global variable
verbose = True
# a function definition taking a couple parameters
def collatz(start,maxsteps):
  cur = start
  step = 0
  if verbose:
    print("start:",start,"maxsteps:",maxsteps)
    print("Step  Current")
    print(f"{step:3}: {cur:5}")
  while cur != 1 and step < maxsteps:
    step += 1
    if cur % 2 == 0:
      cur = cur // 2            # // for integer division
    else:
      cur = cur*3 + 1
    if verbose:
      print(f"{step:3}: {cur:5}")
  return (cur,step)

start_str = input("Collatz start val:\n")
start = int(start_str)
(final,steps) = collatz(start, 500)
print(f"Reached {final} after {steps} iters")
```

Listing 1: Collatz

Chapter 3

Higher Order Functions

We can treat functions like "values". These are referred to as **First-Class Functions**, though this term carries additional obligations of which python does not fulfill everything.

Definition 8 (Higher Order Functions (kinda)). Higher order functions are functions that accept function arguments or return functions (or both).

```python
def apply_all(func_list, data):
    data_list = []
    for func in func_list:
        data_list.append(func(data))
    return data_list
```

Listing 2: Higher order function

3.1 Standard Higher-Order Functions

- **Map**: Creates a new data structure with the function applied to each element.
- **Filter**: Creates a new data structure with elements that return `True` from a function.
- **Reduce**: Repeatedly apply function to an element of the data structure and an accumulator, converting the data structure to a single value.

Python supports **generators** or **iterators**. These only contain a function `next()` which returns the next value in the iterator, or none if it is empty. This makes a much more efficient way to store large lists and apply higher order functions.

Definition 9 (Lambda syntax)**.** Python contains **Lambda expressions**, which is a syntax to create a function body without naming the function. Sometimes referred to as an anonymous function.

```python
print(list(map(lambda y: 2*y, [1,2,3,4,5])))
# [2, 4, 6, 8, 10]
```

Listing 3: Lambda

Unfortunately in python, though lamdbas can accept multiple arguments, it is only a single line, cannot use conditionals, and it must be a single expression. OCaml and Racket have richer support for Lambdas and **lexical closures**.

Chapter 4

Regular Expressions and FSA

Definition 10 (Regular Expression)**.** Regular expressions is a domain specific mini-language used to describe text patterns, to easily recognize and select patterns.

Summary of symbols:

Table 4.1: Symbols

Syntax	Matches
`ab`	The fixed string `ab`
`a+`	One or more of `a`, as many as possible
`a*`	Zero or more of `a`, as many as possible
`a`	Match `a` or `b`
`a{2, 5}`	Match 2 to 5 `a` as in `aa`, `aaa`, ...
`a{2,}`	Match 2 or more `a`
`a{,5}`	Match 0 to 5 `a`
`a?`	Match 0 or 1 `a`
`[0-9]`	Char range `0` to `9`
`\d`	Any digit character `0-9`
`[a-z]`	Any lowercase character
`\w`	Any word character
`.`	Any single character
`\b`	A boundary (but don't include in match)
`\s`	Whitespace (spaces, tabs, newlines)

- Regular expressions are a mini language or DSL often embedded in other programming langauges
- Regex text is usually compiled to a lower form, typically a finite state machine.

- They have their own syntax in different programming languages
- They are not a full programming language

Typically python strings have their own escape sequences using backslash, so we typically use raw strings prefixing with *r*.

```
>>> print("Hello \bworld")
Helloworld
>>> print(r"Hello \bworld")
Hello \bworld
```

Listing 4: Raw String

Essential functions:

Table 4.2: Essential functions

`import re`	Use regex module
`re.findall(regex, text)`	Produce a list of all matching substrings
`re.split(regex, text)`	Produce a list of strings between matches
`re.search(regex, text)`	Produce a `Match` object or `None`
`re.finditer(regex, text)`	Produce an iterable object for `Matches`
`m.group()`	Produce the whole string that matched the regex
`m[0]`	Produce the whole string that matched the regex
`m.span()`	Produce a pair of `(beg, end)` index of match in string

If we only want to search for if matches are found, we can use `re.search(regex, text)` and check if its `Match` or `None`.

4.1 Replacements with Regex

We can use **regex groups** to find and replace using regex. For example, to replace `Chapter X Section Y` with `Chapter X.Y`, we can first capture the X and Y using a regex group.

```
r"Chapter (\d+) Section (\d+)"
```

Listing 5: Regex Groups

Remark. Group 0 is the whole match, Group 1 is the leftmost `(` to its matching `)`. And so on until the number of groups are exhausted.

Substitutions use the syntax `\1` to refer to Group 2, `\2` to refer to Group 3, and so on. We use `re.sub(regex, subst, text, num_occurrences (optional))` to substitute all occurrences of *regex* with *subst* in *text*. If you want to limit the substitutions to the first n occurrences, there is also an optional parameter.

Finally, we can compile regexs down to a finite state machine for improved efficiency using `re.compile`.

4.2 Finite state automata

Introducing: **Automata Theory**!! We can figure out limits of what is possible and what is not and with what. For example, it is impossible to recognize Regular expressions with Boolean Logic.

Definition 11 (Finite state automata)**.** Formally, a finite state automata is defined to be a tuple $(\Sigma, S, s_0, F, \delta)$ where

- Σ is the alphabet or set of allowable characters on each edge of the graph
- S is the set of states (vertices in the graph)
- s_0 the starting vertex
- F is the set of final / accept states (vertices)
- δ is the set of labeled edges, each labeled with an element from the alphabet

A finite state automata accepts a language (a subset of the powerset of an alphabet) if all words in the language L are accepted (end on an accept state).

Finite state automata are directed graphs which indicate to what state we must go based on an input from the alphabet. We can find multiple regexs which satisfy a finite state automata, and multiple finite state automata which satisfy a regex.

Definition 12 (Deterministic Finite State Automata)**.** An FSA is deterministic if from every state there is one unique edge corresponding to an element from the alphabet.

Definition 13 (Nondeterministic Finite State Automata)**.** An FSA is nondeterministic if there are nonunique edges with the same element from the alphabet. There are three main differences:

- Input characters can appear on multiple edges from a state (choices)
- Input characters can appear on no edges from a state
- The ϵ character can be on an edge, which means we can transition to the next state

without consuming any input.

To evaluate whether the NFA accepts a string, we go over all possible state transitions and see if any one accepts.

Theorem 1 (DFA). The minimal deterministic finite state automata is unique.

Previously we introduced Regular Expressions through code informally. However, we can also provide a formal definition, just as we did for the finite state automata.

Definition 14 (Formal Regular Expression). Formally, a regular expression is defined recursively. It can be

- ϵ: the empty string
- $\varnothing$: the empty set of no regexs
- A single item $a \in \Sigma$ from an alphabet.
- R_1R_2: concatenation of two regexs
- $R_1|R_2$: union or alternation of two regexs
- R_1^*: zero or more of a regex, called the **Kleene Closure**

This parts form a minimal set that allow construction of all the convenient regex mechanisms that we have shown previously.

Example (Shorthand). Informally, we say `[ab]+` , but formally this is $(a|b)(a|b)^*$.

Example (Shorthand). Informally, we say `a?b+aa` , but formally this is $(a|\epsilon)bb^*aa$

Definition 15 (Regular Languages). A language is **Regular** if some Finite State Machine accepts it. The FSM may be either deterministic or non-deterministic.

Theorem 2 (Regularity). A language is regular if and only if some **Regular Expression** describes it.

Proof 1. (Idea). Show a procedure to convert a regular expression into an NFA. We can convert a regular expression into a parse tree with each operation (*Star*, *Concat*, *Union*) and the leaves are from the alphabet. Next we can basically build an NFA from the bottom up, piecing together from simple NFAs which accept on just a character. □

Proof 2. (One sided: The other direction involves converting a DFA into a generalized NFA into a regular expression and is a little more involved). We want to convert any regular expression into an NFA. Let R be an arbitrary regular expression. Here we are going to use structural induction to show any regular expression can be converted to an NFA recognizing the same language.

- If $R = a$ for some $a \in \Sigma$. Then $L(R) = \{a\}$, which is recognized by an NFA with two states, one input and one edge with a going to an accepting state. Formally this NFA is $(\Sigma, \{q_1, q_2\}, q_1, \{q_2\}, \{(q_1, a, q_2)\})$.
- If $R = \epsilon$, then $L(R) = \{\epsilon\}$, and an NFA with one state accepting recognizes this language. Formally this NFA is $(\Sigma, \{q_1\}, q_1, \{q_1\}, \varnothing)$.
- If $R = \varnothing$, then the one state non accepting NFA recognizes it. Formally, this is $(\Sigma, \{q_1\}, q_1, \varnothing, \varnothing)$
- If $R = R_1|R_2$, let $(\Sigma, Q_1, q_1, F_1, E_1)$ be an NFA recognizing R_1 and $(\Sigma, Q_2, q_2, F_2, E_2)$ recognize R_2. Then we see that the following NFA recognizes $L(R)$.
$$(\Sigma, \{i\} \cup Q_1 \cup Q_2, i, F_1 \cup F_2, E_1 \cup E_2 \cup \{(i, \epsilon, q_1), (i, \epsilon, q_2)\})$$
- If $R = R_1R_2$, let $(\Sigma, Q_1, q_1, F_1, E_1, E_1)$ be an NFA recognizing R_1 and $(\Sigma, Q_2, q_2, F_2, E_2)$ be an NFA recognizing R_2. Then the concatenation is recognized by
$$(\Sigma, Q_1 \cup Q_2, q_1, F_2, E_1 \cup E_2 \cup \{(f, \epsilon, q_2) | f \in F_1\})$$
- If $R = R_1^*$, let $(\Sigma, Q_1, q_1, F_1, E_1)$ be an NFA that recognizes R_1. Then the following NFA recognizes R_1^*.
$$\{\Sigma, \{i\} \cup Q_1, i, \{i\} \cup F_1, E_1 \cup \{(f, \epsilon, q_1)\} | f \in F_1\}$$

□

Theorem 3 (Closure). Regular expressions are closed under the 3 **regular operations** of concatenation, union, and Kleene closure. All regular expressions that exist can be built from simpler regular expressions with these operations.

Remark. See *Sipser's Introduction to the Theory of Computation* to see these proofs.

Example (Equal AB). Let Equal-ABs be the set of all strings starting with n "a" characters and followed by n "b" characters.

$$\text{Equal-AB} = \{a^n b^n | n > 0\}$$

No such DFA or Regular Expression exists to match this.

Example (Balanced Parenthesis)**.** No DFA or Regular expression exists that matches properly balanced parenthesis.

Example (Compiling Regular Expressions)**.** How do we compile a regular expression R to use it? We first convert it into an NFA through the definition of regular. Then we convert the NFA into a DFA by the equivalence (all NFAs have an equivalent DFA).

Example (Full Compilation: What we want to get to)**.** By our previous theorems, we can finally do the following:

$$\text{Informal Regex} \longrightarrow R \longrightarrow NFA \longrightarrow DFA$$

Theorem 4 (NFA to DFA)**.** A nondeterministic finite state automata can be converted into a deterministic finite state automata.

Proof 3. We first show a conversion from an arbitrary NFA

$$(\Sigma, S, s_0, F, \delta)$$

without epsilon transitions to a DFA $(\Sigma', S', s_0', F', \delta')$.

- Let $\Sigma' = \Sigma$.
- Let $S' = P(S)$, or the powerset of the set of states.
- Let $s_0' = \{s_0\} \in P(S)$. The starting state corresponds to the set with just the starting state.
- Let $F' \subseteq S' = \{V \in P(S) \mid \exists f \in F, f \in V\}$ The DFA accepts if one of the possible NFA states we could be in is an accepting state (there is an accepting state in our set of NFA states).
- Let
$$\delta' = \left\{ \left(V, a, \bigcup_{v \in V} \{s_i \mid (v_i, a, s_i) \in \delta\} \right) \mid V \in P(S), a \in \Sigma \right\}$$

  The transition for x from a state in our DFA V is constructed by taking the union of all possible x transitions from an NFA state in $v \in V$, eg $(v, x, \text{new state})$. We take all the states it could transition to, and consider the set $\{s_0, s_1, \ldots\}$ as our edge. If the set is empty, we add $(V, x, \varnothing \in P(S))$ as our edge. We also add all edges $(\varnothing, x, \varnothing)$ (covered by the set definition).

□

Definition 16 (Epsilon Closure)**.** We define $E(R)$ to be the epsilon closure of a state R in an NFA as

$$\{q \mid q \text{ can be reached from } R \text{ by traveling along zero or more } \epsilon \text{ transitions}\}$$

Proof 4. We continue our previous proof, (full proof this time, including epsilon transitions). We can redefine

$$s_0' = E\left(\{s_0\}\right)$$

and

$$\delta' = \left\{ \left(V, a, \bigcup_{v \in V} E\left(\{s_i \mid (v_i, a, s_i) \in \delta\}\right) \right) \mid V \in P(S), a \in \Sigma \right\}$$

We allow transitions to now include states that are zero or more epsilon transitions away from the ending state, and let the starting state contain all of the epsilon transitions away from the starting state. And now we are finished, as this is a DFA which accepts A. $\square$

Algorithmically, this is inefficient however. We could eliminate dead states after NFA to DFA conversion by getting rid of the unreachable states after a traversal (with BFS / DFS). However, this algorithm is **not good**. Notice how we have to create a full powerset of the states.

Theorem 5 (Lazy NFA to DFA Algorithmic Evaluation)**.** We can keep track of two collections of states, Completed and Todo.

- Add the Starting state as a Todo state.
- Select one "active" state from the Todo state.
- Determine all of the transitions for the "active" state. **Any transition to a state not already seen gets added to the Todo.**
- Mark Active state as completed, and repeat until no Todo states exist.

Finally, add all garbage states.

Chapter 5

OCaml

5.1 History

- Alonzo Church invents Lambda Calculus, a notation to describe computable functions.
- John McCarthy creates Lisp, a programming language modeled after Lambda Calculus. Lisp influenced almost every language that came after it and created new Lisp flavors such as Common Lisp, Emacs Lisp, Scheme, andRacket.
- Robin Milner developed the LCF theorem prover to do math things.
- They invented a Meta Language (ML) which is like Lisp with a type system to tell LCF how to do proofs.
- Xavier Leroy saw ML and realized that its pretty bad as there was no compiler, and it could not run on a personal computer.
  - Leroy develops CAML to allow separate compilation to bytecode and linking
  - Later work introduced ab object system, which is now called Objective Caml, or OCaml.
  - Native code compiler
  - Time traveling debugger

5.2 Bytecode

Definition 17 (Native Code Compilation)**.** Convert source code to a form directly understandable by a CPU.

Definition 18 (Bytecode Compilation)**.** Convert source code to an intermediate form (bytecode) that must be further converted to native code by an interpreter.

Definition 19 (Source Code Interpreter)**.** Directly execute source code as it is read by doing on the fly convertions to native code.

- Java: Compile to bytecode: `javac`, Interpret to native: `java`
- C / C++: Native code compilation: `gcc` , `clang`
- Python: Interpret source code with on the fly bytecode creation: `python`, REPL: `python`
- OCaml: Compile to bytecode: `ocamlc`, Interpret to native: `ocamlrun`, Native code compilation: `ocamlopt`, REPL: `ocaml`

Native code compilation is much faster, BUT we cannot use OCaml's awesome debugger. OCaml is fun because of functional paradigms!!

```
(* collatz.ml: introductory OCaml example demonstrating a variety of
   its features. Compile/Run this via

   >> ocamlc collatz.ml    # byte-compile to a.out
   >> ./a.out              # run the program
   Alternatively one can run directly via
   >> ocaml collatz.ml
*)
open Printf;;
(* allow printf() calls rather than Printf.printf() *)

let verbose = true;;
(* module-level var; will end these with ;; for *)
(*   clarity but can leave it off if desired *)
let collatz start maxsteps =
(* functions defined via name followed by parameters *)
  let cur = ref start in (* let/in introduce name/value bindings *)
  let step = ref 0 in (* immutable by default, ref allows mutability *)
  if verbose then (* main body of collatz func *)
    begin (* several statements in an if/then require *)
      printf "start: %d maxsteps %d\n"
      (* a begin/end block like { } in other languages  *)
              start maxsteps;
              (* no parens around function calls *)
      printf "Step  Current\n";
      (* Side-effect statements like printing/mutation require ; *)
    end; (* end of if/then  *)

  while !cur != 1 && !step < maxsteps do
    (* deref a reference via ! (bang) *)
    if verbose then
      printf "%3d: %5d\n" !step !cur;
      (* print if verbose is enabled *)
    begin match !cur mod 2 with
    (* MATCH different cases given expression: 0 or 1 *)
    | 0 -> cur := !cur/2;   (* rem=0:  even case *)
    | _ -> cur := !cur*3+1; (* rem!=0: odd case *)
    end;  (* begin/end for match that is done for side-effects *)
    step := !step + 1;
  done;
  (!cur,!step) (* final expression to appear is return value *)
;;  (* end of collatz function *)

let _ =  (* equivalent of a "main" block *)
  print_string "Collatz start val:\n"; (* simple string printing *)
  let start = read_int () in (* read an int and convert it *)

  let (final,steps) = collatz start 500 in
  (* call function and capture return tuple *)
  printf "Reached %d after %d iters\n" final steps;
  (* print result *)
;;
```

Listing 6: Collatz in OCaml

5.3 Stuff

- Comments: `(* abc *)`
- Statements: expressions such as `x+1` or `a && b` or `printf "\%d" a;`

  Variables are introduced via `let x = .. in`
- Variable types: string, integer boolean are obvious. Though OCaml is statically typed, it can infer types.
- Assignment via `let x = expr in` or `x := expr;`
- Basic IO: `printf()` or `read_int()`
- Function declarations: `let funcname param1 param2 =`
- Conditionals (if else): `if cond then .. else ..`
  Multiple statements require `begin`/`end` (Match coming soon)
- Iteration (loops): clearly `while cond do`, others soon
- Aggregate data: (arrays , records, objects, etc): `(ocaml, has, tuples)` and others we'll discuss soon.
- Library system: `open Printf` is like `from Printf import *`

5.4 Types and Type Inference

OCaml has type inference, meaning programs do not need to state types explicitly. While explicit types don't appear during normal compilation, they are always present and will appear in error messages.

OCaml has a variety of basic types, such as `int` , `float`, `bool`, `string`. There are also a few special types: `unit` and `'a` .

OCaml also comes with aggregate types.

```
# let ia = [|1; 2; 3|];;
val ia : int array = [|1; 2; 3|]

# let sl = ["a"; "b";];;
val sl : string list = ["a"; "b"]

# let tup = (true, 4.56, "hi");;
val tup : bool * float * string = (true, 4.56, "hi")
```

Listing 7: Aggregate Types

Finally, there are function types. Every function is associated with a type with each parameter and the final return type. This is denoted by each parameter separated with -> and finally the return type.

```
# let add a b = a+b;;
val add : int -> int -> int = <fun>

# add;;
- : int -> int -> int = <fun>

# let selfcat s = s ^ s;;
val selfcat : string -> string = <fun>

# int_of_string;;
- : string -> int = <fun>

# let add_pair (a, b) = a+b;;
val add_pair : int * int -> int = <fun>

# let give_meaning () = 42;;
val give_meaning : unit -> int = <fun>

# let poly_meaning x = 42;;
val poly_meaning : 'a -> int = <fun>
```

Listing 8: Function Types

- Though types are inferred, we can also annotate the code with types if necessary.
- Conflicts between annotated types and inferred types will cause compiler errors
- We can look at Ocaml's Module System which includes Interface Files that state the types of all functions/variables, known as the module signature.

```
# let x : int = 5;;
val x : int = 5

# let add (a : int) (b : int) : int = a+b;;
val add : int -> int -> int = <fun>
```

Listing 9: Type Annotations

5.5 Unit Type

- The notation `()` means `unit` and is the return value of functions that only perform side effects.
- Roughly equivalent to `void` in C or Java.
- Often appears as return type for output functions
- Usually don't worry about unit returns, don't bind the result, and so on
- Functions with no parameters are passed `()` to call them
- **End statements returning unit with a semi-colon (;)** except at the top level where ;; is used instead.

```
# print_string;;
- : string -> unit = <fun>

# print_string "hi\n";;
hi
- : unit = ()
```

Listing 10: Unit Type

5.6 Generic Types

Generic types are labeled as $'a$. This indicates any type. For example, if we have a type

```
val func : 'a -> 'b -> 'a -> 'a
```

Listing 11: Generics

This is a function that takes 3 arguments. The first two arguments must be the same type, and the second does not, and it returns the type of the first argument.

5.7 User Defined Types

User defined types are like a typedef in C. The type keyword allows for an alias.

```
type ilist = int list ;;
let f x : ilist = [1; 2; 3; 4];;
```

Listing 12: Int List

We can also have Variant types, which act like enums.

```
type parity = Even | Odd;;

let swap x = match x with
    | Even -> Odd
    | Odd -> Even
;;
```

Listing 13: Variant Types

The Variant types can also hold data within them.

```
type parity = Even of int | Odd of int;;

let add x = match x with
 | Even(x) -> Odd(x + 1)
 | Odd(x) -> Even(x + 1)
;;
```

Listing 14: Holding Data

They can also be different.

```
type shape = Rect of int * int | Circle of float;;

let area s = match s with
    | Rect (w, l) -> float_of_int (w * l)
    | Circle r -> r *. r *. 3.14
;;
```

Listing 15: Holding Different Data

They can also be recursive.

```
type linked = Item of string * linked | Null ;;

let head lst = match lst with
    | Item(x, _) -> x
    | Null -> ""
;;
```

Listing 16: Linked Node

The types can also be generic.

```
type 'a option =
    Some of 'a | None;;
```

Listing 17: Data Types

5.8 Binding and Syntax

- Names bound to values are introduced with the `let` keyword.
- At the top level, separate these wit double semi-colon ;;

```
let name = "Chris";;
let doubler a =
    2 * a
;;

let pair_to_list (a,b) =
    [a; b];;
```

Listing 18: Source File Example

Bindings can also be nested arbitrarily, and this can be used to do computations.

Note. When writing `.ml` files, known as **Modules**, ending top level bindings with ;; are optional. This is not necessary in source files, but are required in REPL.

```
let first =
    let x = 1 in
    let y = 5 in
    y*2 + x
;;

let second =
    let s = "TAR" in
    let t = "DIS" in
    s ^ t
;;
```

Listing 19: Bindings

Note however that local bindings are local. Therefore,

```
let a =                          (* top level binding *)
    let x = "hello" in           (* local binding *)
    let y = " " in               (* local binding *)
    let z = "world" in           (* local binding *)
    x^y^z                        (* result *)
;;                               (* x,y,z go out of scope *)

print_endline a;;

print_endline x;; (* x is not defined *)
```

Listing 20: Bad Example

This is an example which will not compile. The **scope** of x is only within the local binding. Meanwhile, a is bound at the top level, so it has a module-level scope.

We can fix this like so:

```
let x = "hello";;

let a =                         (* top level binding *)
    let y = " " in              (* local binding *)
    let z = "world" in          (* local binding *)
    x^y^z                       (* result *)
;;                              (* y,z go out of scope *)

print_endline a;;

print_endline x;; (* x is defined *)
```

Listing 21: Fixed Example

5.9 Mutability

- OCaml's default is **immutable** bindings. Once a name is bound, it holds its value until going out of scope.
- Each `let/in` binding creates a scope where a name is bound to a value.
- We can "approximate" mutability with successive `let/in` bindings.

OCaml, as a functional language, tries to follow referential transparency.

Definition 20 (Referential Transparency)**.** If you replace an expression by the value it evaluates to, it will be the same.

This means that the state does not change when evaluating an expression. For example, with referential transparency, the expression

$$f(x) + f(x) + f(x)$$

will evaluate to $3 \cdot f(x)$. However, with side effects,

$$f(x) + f(x) + f(x)$$

will not necessarily evaluate to $3 \cdot f(x)$.

```
let x = 5 in         (* local: bind FIRST_x to 5 *)
let x = x + 5 in     (* local: SECOND_x is FIRST_x + 5, FIRST_x gone
 ↪  *)
print_int x;;        (* prints 10: most recent SECOND_x *)
                     (* top level: SECOND_x out of scope *)
print_endline "";;
```

Listing 22: Approximate Mutability

OCaml however, does have explicit mutability via several mechanisms.

- `ref` : references which can be explicitly changed
- arrays: cells are mutable by default
- records: fields can be labeled `mutable` and then changed

```
let x = 7;;          (* top level x <------+ *)
let y =              (* top level y <----+ | *)
    let z = x+5 in   (* z = 12 =  7+ 5   | | *)
    let x = x+2 in   (* x =  9 =  7+ 2   | | *)
    let z = z+2 in   (* z = 14 = 12+ 2   | | *)
    z+x;;            (* 14+0 = 23 -------+ | *)
                     (* end local scope  | | *)
print_int y;;        (* prints 23 -------+ | *)
print_endline "";;   (*                    | *)
                     (*                    | *)
print_int x;;        (* prints 7 ----------+ *)
print_endline "";;   (*                      *)
```

Listing 23: Let In Bindings

How do we do things without immutability? Oftentimes we can use recursion, and mutable versus immutable variables often provide advantages to verify program correctness and parallelism.

We can create recursive functions using a `let rec` binding (this is kinda annoying tbh). Typically functional languages use tail call optimization to prevent stack overflow as recursion is very powerful in functional programming.

> **Remark.** Tail call optimization occurs if the recursive call occurs right at the end of the function. This can be optimized, as the stack frame the current function is using can mostly be discarded, so the stack frame of the subcall can replace the current stack frame in use.

5.10 Match

- OCaml allows for destructuring data in various ways.
- **Pattern Matching** is often used with data types in OCaml to determine the structure of data and make decisions on it.

```
match something with
    | pattern1 -> result 1
    | pattern 2 ->
      action;
      result2
    | pattern3 -> result3
```

Listing 24: Matching

```
let yoda_say bool =
    match bool with
        | true -> printf "False, it is not.\n"
        | false -> printf "Not true, it is.\n"
;;
```

Listing 25: Yoda Say

```
let counsel mood =
    let message =
        match mood with
            | "sad" -> "Welcome to adult life"
            | "angry" -> "Blame your parents"
            | "happy" -> "Why are you here"
            | "ecstatic" -> "I'll have some of what you're smoking"
            | s -> "Tell me more about "^s
        in
        print_endline message;
```

Listing 26: Counsel Mood

We can also match tuples. We can use an underscore to mean that we don't care or ignore.

```
open Printf;;

let has_meaning pair =
    match pair with
        | (42,42) -> "full of meaning"
        | (42,_) -> "meaning first" (* _ : don't care / ignore *)
        | (_,42) -> "meaning second"
        | _ -> "there is no meaning"
;;
let print_meaning a b c =
    match a,b,c with
        | 4,2,_      (* both patterns use same action *)
        | _,4,2 -> printf "There is meaning\n";
        | x,y,z -> printf "\%d \%d \%d have no meaning\n" x y z;
;;
```

Listing 27: Matching Tuples

```
let xor a b =
    match a,b with
        | true, false
        | false, true -> true
        | _ -> false
;;
```

Listing 28: xor

```
let rec fib n =
    match n with
        | 0 -> 0
        | 1 -> 1
        | n -> (fib (n - 1)) + (fib (n - 2))
;;
```

Listing 29: Fibonacci

Definition 21 (Declarative Programming). Declarative Programming states how the output relates to the input, does not detail how to produce that output.

Pattern matching is declarative, if data matches this pattern, do the following. However, exactly how this pattern is detected is left to the OCaml compiler. The compiler does guarantee **first-to-last checking** of patterns.

5.11 Lists

- Long tradition of **Cons boxes** and singly linked lists in Lisp / ML languages.
- OCaml has immediate list construction with square braces: `[1;2;3]`

Linked lists are comprised of "cons" boxes in OCaml. Each box has a data part and a pointer to another box (which is possible null).

Linked lists are written as `let ilist = [6; 1; 2];;`.

Arrays in OCaml consist of a length value and a list of pointers to each value in the array. They can be defined as following:

```
# let i = 7;; (* i = [ 7 ] *)

# let str = "e";; (* str = [ p ]-> "e" *)

# let empty = [];; (* empty = [ null ] *)

# let ilist = [6; 1];;
(* ilist = [ p ] -> [ 6 | p ] -> [1 | null ] *)

# let strlist = ["a"; "b"];;
(* strlist = [ p ] -> [ p | p ] -> [ p | null ] *)
(*                      |            |          *)
(*                      V            V          *)
(*                     "a"          "b"         *)

# let iarr = [|6; 1; 2|];;
(* iarr = [ p ] -> [ 3 (len) | 6 | 1 | 2 ] *)
```

Listing 30: Linked Lists and Arrays

- `List.hd` returns the first data element
- `List.tl` returns the remaining list

```
let list1 = [6; 1];;
let first = List.hd list1;;
let rest = List.tl list1;;
let len = List.length rest;
let nothing = List.tl rest;
(* tail of length 1 list is null *)
```

Listing 31: Head and Tail

We can also construct a list using the “Cons” operator. For example,

```
let box1 = 7 :: [];; (* box1 = [ 7 ] *)
let box2 = 6 :: box1;; (* box2 = [6; 7] *)
(* can construct a linked list using cons operator *)
```

Listing 32: Cons Operator

Lists are immutable in Ocaml.
We can use the pattern matching with the Cons operator.

```
let rec length_A list =
    match list with
        | []            -> 0
        | head :: tail  -> 1 + (length_A tail)
;;
```

Listing 33: Length

Line 4 here binds the names head and tail. The compiler generates code such as `let head = List.hd list in` and `let tail = List.tl list in`.

Pattern matching is also relatively safe. The following will work and not generate any errors despite ordering of cases.

```
let rec length_A list =
    match list with
        | head :: tail  -> 1 + (length_A tail)
        | []            -> 0
;;
```

Listing 34: Length Ordered

```
(* Create a list of sum of adjacent pairs of elements in list. Last
↪  element in an odd length list is part of the return as is. *)
let rec sum_adj list =
    if list = [] then
        []
    else
        let a = List.hd list in
        let atail = List.tl list in
        if atail = [] then
            [a]
        else
            let b = List.hd atail in
            let tail = List.tl atail in
            (a+b) :: (sum_adj tail)
;;
(* We destructure the list and select cases based on that *)
```

Listing 35: Summing Adjacent Elements

```
let rec sum_adj list =
    match list with
        | []                 -> []
        | a :: []            -> [a]
        | a :: b :: tail     ->
            (a+b) :: sum_adj tail
;;
```

Listing 36: Sum Adjacent Match

```
let rec swap_adj list =
    match list with
        | a :: b :: tail ->
            b :: a :: swap_adj tail
        | _ -> []
;;
```

Listing 37: Swap Adjacent

Minor Match Details:

- First pattern: the pipe is optional.
- Fall through cases: no action -> is given, use next action
- (Note: fall through won't work when there is a name bound. The name must be bound on both sides for it to work. See here.)
- Underscore matches something, no name bound.
- Arrays work in pattern matching but there is no size generalization. Arrays are not defined inductively, so don't usually process them with pattern matching.

The compiler will check for

- Duplicate cases, where one is unreachable.
- Missing cases, where data does not match a pattern.

There are limits to pattern matching, however! Patterns can check structural parts and constants. However, names are always new bindings, and we cannot compare bindings to each other. We also cannot call functions in a pattern.

```
let rec count_occur elem list =
    match list with
        | [] -> 0
        | head :: tail ->
            if head=elem then
                1 + (count_occur elem tail)
            else
                count_occur elem tail
;;
```

Listing 38: Conditional Pattern Matching

However, introducing when guards!! We can check additional conditions, and call functions inside the when guard!

```
let rec count_occur elem list =
    match list with
        | [] -> 0
        | head :: tail when head=elem ->
            1 + (count_occur elem tail)
        | head :: tail ->
            count_occur elem tail
;;
```

Listing 39: Count Occurrences

Note in this example since it is first to last pattern checking, we hit the when clause first if it is true.

```
let rec longer_minlen minlen list =
    match list with
        | [] -> []
        | str :: tail when String.length str > minlen ->
            str :: (longer_minlen minlen tail)
        | _ :: tail ->
            longer_minlen minlen tail
;;
```

Listing 40: Minlen

```
let elems_between start stop list =
    let rec helper i lst =
        match lst with
            | _ when i > stop ->
              []
            | _ :: tail when i < start ->
              helper (i + 1) tail
            | head :: tail ->
              head :: (helper (i + 1) tail)
            | _ -> failwith "out of bounds"
    in
    helper 0 list
;;
```

Listing 41: Elements Between

5.12 Functions and Lambda Expressions

Rather than `lambda` , OCaml provides anonymous functions via the `fun` syntax.

```
let add1_stand x =
    let xp1 = x + 1 in
    xp1
;;

let add1_lambda =
    (fun x ->
        let xp1 = x + 1 in
        xp1)
;;

let eight = add1_stand 7;;
let ate = add1_lambda 7;;
```

Listing 42: Anonymous Functions

Note. There is an equivalence between `let func a = ..` and `let func = fun a -> ..`. The former is syntactical sugar for the latter.

We can use fun syntax as arguments to higher order functions.

```
let evens list =
    filter (fun n -> n mod 2 = 0) list
```

Listing 43: Higher Order Functions

If predicates are more than a couple lines, favor a named helper function with nicely formatted source code for readability.

```
let is_some list =
    let pred opt =
        match opt with
            | Some a -> true
            | None -> false
        in
        filter pred list
    ;;
```

Listing 44: Some

Fun syntax can be used anywhere a value is expected including but not limited to:

- Top level let bindings
- Local let/in bindings
- Elements of arrays, lists, tuples
- Values referred to by refs
- Fields of records

```
let func_ref = ref (fun s -> s ^ " " ^ s);;
let bambam = !func_ref "bam";;
func_ref := (fun s -> "!!!");;
let exclaim = !func_ref "bam";;
```

Listing 45: Fun syntax

5.13 Higher Order Functions

Higher order functions have 4 major families.

Table 5.1: Higher Order Functions

Pattern	Description	Library Functions
Filter	Selects some elements from a DS `('a -> bool) -> 'a DS -> 'a DS`	List.filter, Array.filter Map.filter, Hashtbl.filter
Iterate	Perform side effects on each element of a DS `('a -> unit) -> 'a DS -> unit`	List.iter, Array.iter Queue.iter, Map.iter
Map	Create a new DS with different elements, same size `('a -> 'b) -> 'a DS -> 'b DS`	List.map, Array.map Map.map
Fold	Compute a single value based on all DS elements `('a -> 'b -> 'a) -> 'a -> 'b DS -> 'a`	List.fold_left Array.fold_right Queue.fold, Map.fold Hashtbl.fold

```
let ilist = [9; 5; 2; 6; 5; 1;];;
let silist = [("a", 2); ("b", 9); ("d", 7)];;
let ref_list = [ref 1.5; ref 3.6; ref 2.4; ref 7.1];;

List.iter (fun i -> printf "\%d\n" i) ilist;;

List.iter (fun (s, i)-> printf "str: \%s int: \%d\n" s i) silist;;

List.iter (fun r-> r := !r *. 2.0) ref_list;;
```

Listing 46: Iter

```
let rec iter func list =
    match list with
        | [] -> ()
        | h :: t -> func hd;
                    iter func t
;;
```

Listing 47: Sample Iter definition

Note. Note that this is tail recursive.

```
let ilist = [9; 5; 2; 6; 5; 1;];;

let doubled_list = List.map (fun n-> 2 * n) ilist;;

let as_strings_list = List.map string_of_int ilist;;

let silist = [("a", 2); ("b", 9); ("d", 7)];;
let ref_list = [ref 1.5; ref 3.6; ref 2.4; ref 7.1];;

let swapped = List.map ( fun (s, i) -> (i, s)) silist;;

let first_only = List.map fst silist;;

let derefed = List.map (!) ref_list;;

let with_square_list =
    List.map (fun r -> (!r, !r *. !r)) ref_list;;
```

Listing 48: Map

```
let rec map trans list =
    match list with
    | [] -> []
    | head :: tail -> (trans head) :: (map trans tail)
;;
```

Listing 49: Simple Map Implementation

Folding allows us to reduce all elements to a computed value or accumulate all elements to a final result. Folding is very general, it is possible to write Iter, Filter, and Map via only Fold. This is left as an exercise, and it is a good thing to do.

```
(*
val List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
                      cur  elem   next   init  the list  result
*)
let fold_left func init list =
    let rec help cur lst =
        match list with
        | [] -> cur
        | head::tail -> let next = func cur head in
                        help next tail
    in
    help init list
;;
```

Listing 50: Fold Left Implementation

```
let ilist = [9; 5; 2; 6; 5; 1;];;

let sum_list = List.fold_left (+) 0 ilist;

let sumsquare =
    List.fold_left (fun sum n -> sum + n * n) 0 ilist;;
```

Listing 51: Fold Left uses

Note that folded values can be data structures. However, since the motion of fold_left from left to right, the resulting lists are in reverse order.

```
let ilist = [9; 5; 2; 6; 5; 1;];;

let rev_list = List.fold_left (fun cur x -> x :: cur) [] ilist ;;

let ref_seqlists =
    List.fold_left
    (fun all x -> (x :: (List.hd all) ) :: all)
    [[]] ilist ;;
```

Listing 52: Fold Left DS

However, we can also fold from right to left. This is NOT tail recursive, but it does allow in order results.

```
let rec fold_right func list init =
    match list with
        | [] -> init
        | head :: tail ->
            let rest = fold_right func tail init in
            func head rest
```

Listing 53: Sample R2L Fold Implementation

Note the differences with the fold_left implementation, where we use a helper function.

```
let nums = [1; 2; 3; 4];;

List.fold_right (fun e l -> e :: l) nums [];;
```

Listing 54: Right to left Folding

Note. How do we fold over a tree? The general functional way of defining a fold is to replace all constructors with functions. For example, if our list is defined by `a :: (b :: (c :: []))`, then we replace the `::` operator with a function.

For trees, we replace each node constructor with a function taking in three arguments (the accumulator and the left and right values). For example,

```
let rec fold_tree f a t =
    match t with
        | Leaf -> a
        | Node (l, x, r) ->
            f x (fold_tree f a l) (fold_tree f a r);;
```

Listing 55: Fold Tree

Note. Map and Fold/Reduce are nice ways to transform lists. However, in some cases, our list is way too large to fit into memory or even on a disk (Big Data). In that case, we can use generators and **distributed map-reduce frameworks** to process large data (such as Apache Hadoop or Google MapReduce). After specifying a few functions to map and reduce data, these frameworks build a distributed way to compute these through map and reduce workers.

5.14 More Occam Data Types

Table 5.2: Aggregate Data Structures

	Elements	Access	Mutable	Example
List	Homogeneous	Index/PatMatch	No	`[1;2;3]`
Array	Homogeneous	Index	Yes	[|1;2;3|]
Tuples	Heterogeneous	PatMatch	No	`(1, "two", 3.0)`
Records	Heterogeneous	Field/PatMatch	No/Yes	`name="Sam"; age=21`
Variant	N/A	PatMatch	No	type letter = A | B | C;

5.15 Records

- Heterogeneous with named fields like structs
- Dot notation is used to access record field values
- Records and fields are immutable by default
- Create new records using with syntax to replace field values
- Fields declared mutable are changeable using the ← operator

```
type hobbit = { name : string; age : int};;
let bilbo = { name = "Bilbo"; age = 111 };;
let sam = { name = "Samwise"; age = 21 };;
let smeagol = { name = "Smeagol"; age = 300 };;

sam.age;;
same.name;;

let old_sam = {sam with age=100};;
let gollum = { smeagol with name = "Gollum"; age = 589 };;

type mut_hob = {
    mutable name : string;
    age : int
};;

let h = { name = "Smeagol"; age = 25};;
h.name <- "Gollum";;

let rec hobbit_bdays (lst : mut_hob list) =
   match lst with
       | [] -> []
       | hob :: tail ->
        { hob with age = hob.age + 1 } :: (hobbit_bdays tail)
;;
```

Listing 56: OCaml Records

Note that the *ref* type in OCaml is actually just a record with a single mutable field. For example:

```
type 'a myref = { mutable contents : 'a };;

(* Create a myref, same as the ref function *)
let make_ref x =
  {contents = x};;

(* Dereference a reference *)
let deref myref =
  myref.contents;;

(* Define symbol to be the same function as above; prefix operator
↪  as above *)
let (!) myref =
  myref.contents;;

(* Define an assignment function *)
let assign myref x =
  myref.contents <- x;;

(* Define symbol to be same as above; infix op same as default *)
let (:=) myref x =
  myref.contents <- x;;
```

Listing 57: Refs

5.16 Algebraic or Variant Data Types

```
type fruit =
    Apple of string | Orange | Grapes of int;;

let a = Apple("Fuji");;
let g = Grapes(7);;

let count_fruit f =
    match f with
        | Apple(_) -> 1
        | Orange -> 1
        | Grapes(n) -> n
;;
```

Listing 58: Fruits

- Kind of like enums in C and java
- More powerful, allows pattern matching
- Algebraic or Variant types allows different kinds of values associated with a label
- Note that though an algebraic type is a single type, its variants may have different kinds of data associated with them.

```
type identifier =
    | Email of string
    | Phone of int
;;

let mixed_list = [
    Email "example@gmail.com";
    Phone 301301301;
    Email "example@yahoo.com";
    Phone 123456789;
];;
```

Listing 59: Variant Types

We can also pattern match.

```
type identifier =
    | Email of string
    | Phone of int
;;

let rec sum_phones list =
    match list with
        | [] -> 0
        | (Phone i)::tail ->
            i + (sum_phones tail)
        | _::tail ->
            sum_phones tail
;;
```

Listing 60: Sum Phones

Option type! This is used to indicate presence or absence of something. This is often used as a return value, in case of errors.

```
type 'a option = None | Some of 'a;;
```

Listing 61: Option

- Instead of throwing errors, we can return None or Some.
- Exceptions crash the program, while None propagates the error.
- Many builtin operators have alternatives: either return Some 'a / None, or return 'a / throw a Not_found exception.

Note that lists are also algebraic types.

```
type 'a mylist =
     | Empty
     | Cons of ('a * 'a mylist)
;;
```

Listing 62: List Type

We can see that we can also have recursive algebraic types. This is also useful for trees:

```
type mytree =
    | Leaf
    | Node of 'a * strtree * strtree
;;
```

Listing 63: Tree Data Structure

Anonymous records can also be used to name objects when hard to understand.

```
type mytree =
    | Leaf
    | Node of { data : 'a;
                left : mytree;
                right : mytree}
;;
```

Listing 64: Tree Data Structure with Record

Another interesting type: Result. We can use multiple type parameters.

```
type ('a, 'b) result = Ok of 'a | Error of 'b;;
```

Listing 65: Exception

Chapter 6

Context Free Grammars

Definition 22 (Context Free Grammar). A context free grammar is a 4 tuple (V, Σ, R, S), where

1. V is a finite set called the variables,
2. Σ is a finite set disjoint from V, called the terminals,
3. R is a finite set of rules, with each rule being a pair of a variable and a string of variables and terminals,
4. $S \in V$ is the start variable.

If u, v and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar, we say that uAv **yields** uwv, written as

$$uAv \implies uwv$$

We say that u **derives** v, written as $u \implies v$, if $u = v$ or if a sequence $u_1, u_2, \ldots u_k$ exists for $k \geq 0$ and

$$u \implies u_1 \implies u_2 \implies \ldots u_k \implies v$$

The language of the grammar is $\{w \in \Sigma^* | S \implies w\}$. The ϵ terminal can also be used as an empty string.

Example (Equal ABs).

1. $X \to aXb$
2. $X \to \epsilon$

Example (Derive aabb).

$$\begin{aligned} X &\Rightarrow_1 aXb \\ &\Rightarrow_1 aaXbb \\ &\Rightarrow_2 aabb \end{aligned}$$

Example (PlusTimes).

1. $A \rightarrow A + A$
2. $A \rightarrow A \cdot A$
3. $A \rightarrow N$
4. $N \rightarrow DN$
5. $N \rightarrow D$
6. $D \rightarrow 0|1|2|3|\ldots|9$

Example (Deriving PlusTimes).

$$\begin{aligned} A &\Rightarrow_1 A + A \\ &\Rightarrow_2 A * A + A \\ &\Rightarrow_3 N * A + A \\ &\Rightarrow_3 N * N + A \\ &\Rightarrow_3 N * N + N \\ &\Rightarrow_4 DN * N + N \\ &\Rightarrow_5 DD * N + N \\ &\Rightarrow_4 DD * DN + N \\ &\Rightarrow_5 DD * DD + N \\ &\Rightarrow_5 DD * DD + D \\ &\Rightarrow_5 12 * 34 + 5 \end{aligned}$$

Context-Free Grammars can also be represented by Parse Trees, which are graphical representations of a string being derived from a CFG. This is typically how the strings are represented in code.

Definition 23 (Leftmost Derivation). A derivation of a string w in G is a leftmost derivation if at every step the leftmost remaining variable is the one replaced.

Definition 24 (Rightmost Derivation)**.** A derivation of a string w in G is a rightmost derivation if at every step the rightmost remaining variable is the one replaced.

Definition 25 (Ambiguity)**.** A string w is derived ambiguously in context free grammar G if it has two or more different leftmost derivations. A grammar G is ambiguous if it generates some string ambiguously.

Example (Converting to Non Ambiguous)**.** Convert the following Grammar to a non ambiguous grammar, where the precedence is $\sim, \cap, \cup$.

$$S \to S \cap S$$

$$S \to S \cup S$$

$$S \to \sim S$$

$$S \to T$$

$$T \to true|false|a|b|c$$

We can convert this by yielding all $\cup$ first, then $\cap$, and then $\sim$, so we get

$$S \to S \cup T | T$$

$$T \to T \cap A | A$$

$$A \to \sim A | B$$

$$B \to true|false|a|b|c$$

Theorem 6 (Proving Ambiguity)**.** We can prove a Grammar is ambiguous by using only leftmost derivations and deriving the same string via two different derivations.

Definition 26 (Chomsky Normal Form)**.** A context free grammar is in Chomsky normal form if every rule is of the form

$$A \to BC$$

$$A \to a$$

where a is any terminal and A, B, C are any variables, except B, C cannot be the start variable. In addition, we permit the rule $S \to \epsilon$, where S is the start variable.

Theorem 7. Any context free language is generated by a context free grammar in Chomsky normal form.

6.1 CFGs and Parsing

Definition 27 (Operator Precedence)**.** When creating CFGs to parse programming languages, often associate production rules for different levels of operator precedence. High precedence will be lower down in the CFG / parse tree.

Example (Unary Math)**.** Let the CFG be the following:

$$A \to A + A | A - A | M$$

$$M \to M * M | M/M | T$$

$$T \to N | U$$

$$U \to -N$$

$$N \to number$$

If we parse $5 + 12/ - 3 - 7$ using leftmost derivation, we get

$$(5 + (12/(-3)) - 7)$$

Definition 28 (Cycle)**.** A CFG has a cycle if we can derive some string s which contains the non terminal symbol A from A. For example: $A \to B \to A$

Example (Programming)**.** Consider the following CFG.

$$A \to (B)$$

$$A \to id | number$$

$$B \to BA$$

$$B \to A$$

where id is any identifier like x, $+$, foo, $*$. All of the following can be derived from this CFG:

```
hello
(doit)
(+ 1 2 3)
(define (double x) (* x 2))
(let ((a 7) (b 9)) (+ a (* 10 b)))
```

Listing 66: Some Parsing CFG

There is a distinction between parse trees and abstract syntax trees. Parse trees show all characters and how they would derive from a CFG. ASTs eliminate some raw characters. For example, *Add* and *Mul* would be nodes in an AST, while they would be symbols or leaves in a CFG.

Chapter 7

Lexing and Parsing

How do we process code to compile it?

- Get input: For example, `5 + 10*4 + 7*(3+2)`
- Lex input to tokens: `[Int 5; Plus; Int 10; Times; Int 4; Plus;`...
- Parse tokens to an expression tree:
  `Add(Const(5), Add(Mul(Const(10), Const(4)),` ...
- Evaluate tree: `80`

7.1 Lexing

- Raw input is just a bunch of characters
- Lexing is done to ease processing later on
- Group characters into **tokens**

```
type token = Plus | Times | OParen | CParen | Int of int;;
```

Listing 67: Token

- More extensive arithmetic will include Subtraction, Division, Floating Point
- Full programming languages have variable identifiers, keywords like let/in and for/do
- Typically spaces do not get tokenized
- Some tokens can consist of multiple characters

```
let lex_string string =
    let len = String.length string in
    let rec lex pos =
        if pos >= len then
            []
        else
            match string [pos] with
            |' ' | '\t' | '\n' -> lex (pos + 1)
            | '+' -> Plus :: (lex (pos + 1))
            | '*' -> Times :: (lex (pos + 1))
            | '(' -> OParen :: (lex (pos + 1))
            | ')' -> CParen :: (lex (pos + 1))
            | d when is_digit d ->
                let stop = ref pos in
                while !stop < len && is_digit string.[!stop] do
                    incr stop;
                done;
                let numstr = String.sub string pos (!stop - pos)
                let num = int_of_string numstr in
                Int(num) :: (lex !stop)
            | _ ->
                let msg = sprintf "lex error at char \%d" pos
                ↪  string.[pos] in
                failwith msg
    in
    lex 0
;;
```

Listing 68: Example Lexing

- Might benefit from **regular expression** matching
- Often toolchains for lexers use regular expressions

7.2 Parsing

- CFGs allow formal specification of syntax
- Derive strings from CFGs
- Parsers do the inverse: Given a string, see if it is allowed by the grammar
- Inverse problem is harder
- Top Down versus Bottom Up

- Leftmost derivation versus Rightmost derivation
- Deterministic based on finite lookahead, or nondeterministic
- Recursive descent, Pushdown automata, Parsing tables
- LL(k): Left to right parsing, leftmost derivation, k-token look-ahead
- LALR(k): Left to right parsing, rightmost derivation, k-token lookahead

Note. Note that parsers cannot necessarily detect if all strings are valid in all grammars. Some classes of grammars such as the LL(k) class can be fully decided by a recursive descent parser. The LL(k) class excludes all ambiguous grammars and all left-recursive grammars.

Goal of parsers: Go from tokens to abstract syntax tree.

```
type expr =
    | Add of expr * expr
    | Mul of expr * expr
    | Const of int
;;
```

Listing 69: Abstract Syntax Tree

Definition 29 (Recursive Descent Parser)**.** A parser which handles parsing by using a series of functions (typically one function per nonterminal symbol), which are possibly recursive. These functions look for non terminal symbols and attempts to consume them according to production rules.

Some properties of recursive descent parsers include:

- Top down parsing: constructs upper parts of Parse Tree before going lower.
- Non deterministic: involves backtracking
- Lookahead: looks forward 1 or more tokens in input to proceed

Consider the following CFG:

$$A \to A + A$$

$$A \to M$$

$$M \to M * M$$

$$M \to N$$

$$N \to \text{number}$$

$$N \to (A)$$

An example string can be $5 + 10 * 4 + 7 * (3 + 2)$. We can think of the following structure:

```
let parse_tokens tokens =

    (* A parse_A: addition only *)
    let rec parse_A toks =
        ...

    (* M parse_M: multiplication *)
    and parse_M toks =
        ...

    (* N parse_N: self evaluating tokens like Int and Paren *)
    and parse_N toks =
        ...
    in

    (* Main code for parse_tokens, end helper functions *)
    let (expr, rest) = parse_A tokens in
        ...
    ;;
```

Listing 70: Parser for CFG

For example, the highest precedence level can be seen in the following:

```
let rec parse_N toks =
    match toks with
        | [] ->
            raise (ParseError {msg = "expected expression";
            ↪  toks=toks})
        | Int n :: tail ->
            (Const(n), tail)
        | OParen :: tail ->
            begin
                let (expr, rest) = parse_A tail in
                match rest with
                    | CParen :: tail -> (expr, tail)
                    | _ -> raise (ParseError {msg = "unclosed
                    ↪  paren"; toks=rest})
            end
        | _ ->
            raise (ParseError {msg="syntax err"; toks=toks})
```

Listing 71: Highest Precedence

```
and parse_M toks =
    let (lexpr, rest) = parse_N toks in
    match rest with
    | Times :: tail ->
        let (rexpr, rest) = parse_M tail in
        (Mul(lexpr,rexpr), rest)
    | _ -> (lexpr, rest)
```

Listing 72: Second Precedence

- Parsing is a search process in which each function tries to consume tokens with some failures allowing backtracking
- Gets tricky to understand with parenthesis involved which circle back to the top of parsing functions
- We can visualize this with a function call stack
- This is where debugging helps

Note. We can trace functions using `##trace funcname;;` in the OCaml REPL, shows calls with params and return values.

7.3 Evaluation

Evaluation is actually quite simple, we just recurse down the tree and evaluate each subexpression. For example,

```
let rec evaluate expr =
    match expr with
        | Const i -> i
        | Add(lexpr, rexpr) ->
            let lans = evaluate lexpr in
            let rans = evaluate rexpr in
                lans + rans
        | Mul(lexpr, rexpr) ->
            let lans = evaluate lexpr in
            let rans = evaluate rexpr in
                lans * rans
;;
```

Listing 73: Evaluation

Note. OCaml has a "Threading operator". For example, if we have `let ans = f3 ( f2 ( f1 x )) in ...` this can be rewritten as `let ans = x \|> f1 \|> f2 \|> f3`

7.4 Advanced Processing

Consider the expression `10 - 2 - 3` when parsing. Note that if we used our previous parser, we would evaluate $2 - 3$ first, and then $10 - (x)$.

Definition 30 (Right and Left Associative)**.** If we have an arbitrary expression $a \sim b \sim c$, the operator $\sim$ is **left associative** if the expression is interpreted as $(a \sim b) \sim c$, and **right associative** if it is interpreted as $a \sim (b \sim c)$.

Note that addition and multiplication are both left and right associative. However, subtraction is left associative, while our parser is right associative.

```
(* Compare this with the parse_M recursive parser *)
let parse_Sub toks =
    let (lexpr, rest) = parse_N toks in
    let rec iter lexpr toks =
        match toks with
            | Minus :: rest ->
                let (rexpr, rest) = parse_N rest in
                    iter (Sub(lexpr, rexpr)) rest
            | _ -> (lexpr, toks)
    in
    iter lexpr rest
```

Listing 74: Left associative Parser Step

- Right associative parsers recurse deeply to the right to generate right hand expression
- Left associative iterates consuming subtraction operations in a (tail recursive) loop
- Left associative creates left-heavy tree by combining right and left expressions in a Sub and then passing it forward in the iteration to be the left branch.

7.5 Advanced Lexing

- Often, lexing takes too much space
- Typically implemented as a lexing buffer or iterator
- Contains `next()` to see the next token.
- Lexer generators improve over handwritten lexers
- **Lex** is a classic tool to generate lexers: describes how characters translate to tokens

7.6 Advanced Parsing

- **Yacc**: Yet another compiler compiler
- Parser generator
- Input grammar / token types to Yacc to create a parser
- Most high level languages actually use a **compiler** on top of this to optimize and convert to assembly.

Chapter 8

Lambda Calculus

Definition 31 (Untyped Lambda Calculus)**.** Untyped Lambda calculus can be described by the following CFG:

$$T \to x \text{ (Variable name)}$$

$$T \to \lambda x.T \text{ (Abstraction)}$$

$$T \to TT \text{ (Application)}$$

Example (Lambda Terms)**.**

$$\lambda x.\lambda y.xy$$

$$x\,y\,z \text{ (Same as } (x\,y)\,z\text{: left derivation)}$$

Note. Application associates left. Though the CFG is ambiguous, we assume all applications are Left Associative (see example above).

Definition 32 (Bound Variable)**.** A variable x is bound when it occurs within the body of an abstraction: eg

$$\lambda x \ldots x \ldots$$

The abstraction is known as the **binder** of x.

Definition 33 (Free Variable)**.** A variable that is not bound.

Definition 34 (Combinator)**.** A Lambda term with no free variables.

Example (Identity Function)**.** The identity function is a combinator.

$$\lambda x.x$$

Definition 35 (Alpha Conversion)**.** Terms are equal up to renaming of bound variables. For example,

$$\lambda x.x \equiv \lambda y.y$$

$$\lambda x.\lambda y.xy \equiv \lambda q.\lambda r.qr$$

This is known as Alpha-Conversion. Alpha conversion can be performed by:

- Traverse the AST tree of the lambda term CFG
- When encountering an abstract node, replace bound variable x with v_i
- Each later appearance of x is substituted with **fresh variable** v_i.
- Increment counter i.

Definition 36 (Beta Reduction)**.** Beta reduction is as following: Apply the function by **substituting** bound variables with their parameter in the abstraction body.

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

Substitution:

$$(\lambda x.t_1)t_2 \implies [x \mapsto t_2]t_1$$

"substitute t_2 for x in t_1"

Definition 37 (Normal Form)**.** When Beta reduction is no longer possible it is referred to as **Normal Form** or **Beta Normal Form**.

Example.

$$\lambda x.(x((\lambda y.y)a)) \implies \lambda x.(xa)$$

Example.

$$((\lambda x.x)a)((\lambda x.x)b) \implies ab$$

Definition 38 (Lazy Evaluation)**.** Also known as Call by Name. Reduce Leftmost / Outermost Application first. Abstractions that are not applied do not reduce. Substitute entire argument first into abstractions.

Definition 39 (Eager / Strict Evaluation)**.** Also known as Call by Value. Evaluate argument to applications first by reducing them to their normal form. Then perform substitution within Abstractions. Abstractions that are not applied do not reduce. Reduce argument first before subbing into abstractions.

Example (Lazy Evaluation)**.**

$$\begin{aligned}(\lambda z.z)((\lambda y.y)x) &\implies (\lambda y.y)x \\ &\implies x\end{aligned}$$

Example (Eager Evaluation)**.**

$$\begin{aligned}(\lambda z.z)((\lambda y.y)x) &\implies (\lambda z.z)x \\ &\implies x\end{aligned}$$

Theorem 8 (Church-Rossner)**.** Reductions can be done in any order and will always reach the same normal form. Normal forms are unique under full beta reduction. (Caveats include "up to alpha conversion", and "termination").

Haskell uses lazy evaluation, while oCaml uses eager evaluation.

Definition 40 (Booleans)**.** The definition of true in lambda calculus is

$$\lambda x.\lambda y.x$$

False is

$$\lambda x.\lambda y.y$$

Boolean values are if/then/else expressions. See following:

Example (If Statement)**.**

$$\begin{aligned}\text{if true then a else b} &\implies \text{True} a\, b \\ &\implies (\lambda x.\lambda y.x)ab \\ &\implies a\end{aligned}$$

Note we get b if False is placed instead of True.

Definition 41 (Church Numeral)**.** We define

$$0 = \lambda f.\lambda x.x$$

$$1 = \lambda f.\lambda x.fx$$
$$2 = \lambda f.\lambda x.f(fx)$$
$$n + 1 = \lambda f.\lambda x.f(nfx)$$

Definition 42 (IsZero Function)**.** This function tests if a value is 0 or not:

$$\lambda z.((z(\lambda y.\text{False}))\text{True})$$

Definition 43 (Addition)**.** This function adds two numbers:

$$\lambda f.\lambda x.(\text{M } f(\text{N } fx))$$

Definition 44 (Omega Combinator)**.**

$$(\lambda x.xx)(\lambda x.xx)$$

Note that by attempting to reduce this, we encounter an infinite loop. Therefore, we can create loops.

Definition 45 (Fixed Point or Y Combinator)**.**

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

This allows the parameter f to continue replicating, which is known as "recursion". This makes Lambda Calculus Turing complete.

Note. Consider the following
$$(\lambda x.\lambda y.xy)y$$

Note that the OUTSIDE y is different from the INSIDE y. This is fixed through alpha reduction.

Chapter 9

Operational Semantics

Formal Semantics

- Attempts to describe with some mathematical rigor the meaning of programming language statements
- Comes with several flavors
- Useful to quickly describe to humans small features of languages
- Used by some proofs about properties of languages

Definition 46 (Operational Semantics)**.** OpSem is one flavor of Formal Semantics, which relates syntax of a language to behavior of an abstract machine. This **describes meanings through how things execute**.

On the other hand, denotational semantics describes meanings through mathematical constructs, and axiomatic semantics describes meanings through promises.

Definition 47 (Rules of Inference)**.** The way we notate OpSem is the following:

$$\frac{H_1 \dots H_n}{C}$$

This means "If the conditions $H_1 \dots H_n$, or the hypotheses are true, then the condition C, or the conclusion is true. If there are no hypotheses, then the conclusion is called an axiom and automatically holds.

Example (Sum)**.**

$$\frac{e_1 \implies n_1 \;\; e_2 \implies n_2 \;\; n_3 \text{ is } n_1 + n_2}{e_1 + e_2 \implies n_3}$$

Note. The statement n_3 is n_1+n_2 is in the meta language, which is used to describe the target language.

Example (Grammar Correspondence)**.** The following Grammar:

$$E \to n|E+E$$

can be seen as the following OpSem:

$$\frac{}{n \implies n}$$

$$\frac{e_1 \implies n_1 \quad e_2 \implies n_2 \quad n_3 \text{ is } n_1+n_2}{e_1+e_2 \implies n_3}$$

However, if we have variable names and identifiers, then we need some kind of environment A to store variables and their values. For example, the following Grammar:

$$E \to x|n|E+E|\text{let x} = E \text{ in } E$$

If our E is x, then we have

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

This states that "If in the environment, $x=v$, then we can arrive at the conclusion that x evaluates to with that environment".

Now suppose our $E = \text{let x} = e_1 \text{ in } e_2$. Then we get

$$\frac{A;e_1 \implies v_1 \quad A,x:v_1;e_2 \implies v_2}{A;\ \text{let } x = e_1 \text{ in } e_2 \implies v_2}$$

Using these, we can derive proofs. For example, in this Grammar, if we want to prove that let $x = 3$ in $x+4$ is valid and evaluates to 7, we can do the following. We want to show

$$\frac{}{A;\text{let } x = 3 \text{ in } x+4 \implies 7}$$

We then get

$$\frac{\frac{}{A;3 \implies 3} \quad \frac{\frac{A,x:3(x)=3}{A,x:3;x \implies 3} \quad \frac{}{A,x:3;4 \implies 4} \quad 7 \text{ is } 3+4}{A,x:3;x+4 \implies 7}}{A;\text{let } x = 3 \text{ in } x+4 \implies 7}$$

Definition 48 (Lambda Calculus in OpSem)**.**

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

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

$$\frac{A;e_1 \implies e'}{A;(e_1e_2) \implies (e'e_2)}$$

$$\frac{A; e_2 \implies e'}{A; (e_1 e_2) \implies (e_1 e')}$$

$$\frac{A; e_2 \implies e' \quad A, x : e'; e_1 \implies e''}{A; ((\lambda x.e_1) e_2) \implies e''}$$

Chapter 10

Rust

10.1 History

Best language best language.

- Started as a side project by Graydon Hoare while working at Mozilla
- Compiler initially written in OCaml, but became self-hosting
- Rust foundation was created and Mozilla took on the language
- Rust is the most loved programming language since 2016 according to Stack Overflow.

```rust
use std::io;
// imports another "crate" (package) which is used to handle input

use std::str::FromStr;
// used to get at the i32::from_str() method to convert a string to
// int, one of two paths shown below the other being the obtuse
// instr.parse::<i32>()

const VERBOSE: bool = true;
// module level variables, all caps to avoid compiler warnings

// collatz function: notice parameter and return types are
// required. i32 is a 32-bit signed integer (e.g. positive or
// negative).
fn collatz(start: i32, maxsteps: i32) -> (i32, i32) {
    let mut cur = start; // by default vars are immutable but
    let mut step = 0; // 'mut' keyword adjusts this
    if VERBOSE {
        // no ( ) around conditions
        println!("start: {start}"); // format substituion in prinln!() mcaro,
         ↪  most of the time...
        println!("Step  Current");
        println!("{step:3} {cur:5}");
    }
    while cur != 1 && step < maxsteps {
        step += 1;
        if cur % 2 == 0 {
            cur = cur / 2;
        } else {
            cur = cur * 3 + 1;
        }
        if VERBOSE {
            // using 'if(VERBOSE)' will cause the compiler to complain
            println!("{step:3} {cur:5}");
        }
    }
    (cur, step) // last value sans as semicolon (;) is the function return value
                // return (cur,step);                  // alternative explicit
                 ↪  return
}
```

Listing 75: Collatz in Rust

10.2 Stuff

- Comments: `// comment`
- Expressions: `x + 1`, `a && b`
- Statements: `println!("hi");`, statement lines end with semicolons
- Assignment via `let x = expr;` or `let mut x = expr;`
- Basic IO: `println!()`, `io::stdin.read_line()`
- Function declarations: `fn funcname(param1: type1) -> RetType`
- Conditionals: `if cond { ... } else { ... }`, also `match` conditions
- Loops: `while cond { ... }` and `for iter {}`
- Rust has tuples and variant types
- Great standard library

10.3 Philosophy

- Rust's main directive: avoid memory bugs at all costs
- Provides great mechanisms for error handling but forces programs to contend with errors
- If failing, fail predictably
- While OCaml and Python are "memory safe", they rely on using a garbage collector
- Rust provides memory safety without a garbage collector, which takes up less overhead
- Aims for zero cost abstractions
- Forces concepts such as ownership and lifetimes

Rust borrows a lot from many languages.

- It borrows syntax and generics from C++
- Defaults to immutable data with explicit labels for mutability, and has rich pattern matching and variant types like OCaml
- Has useful array like data structures and all else prefers iterators, map, and reduce like python
- Makes use of code annotations and uses method dispatch like Java
- Has powerful macro creations such as Lisp

10.4 Ownership of Memory

The main point of ownership is each value or memory block in Rust has an owner. There can only be one owner at a time, and when the owner goes out of scope, the value will be dropped (de-allocated).

For example, consider the following code.

```
fn print_str() {
    let i = 5;
    let s = "hello";
    let h = String::new("there");

    println!("{i} {h} {s}");
}
```

Listing 76: Rust Print Str

At the end of this, *h*, which is initially heap allocated, goes out of scope and will be dropped. Meanwhile, in C we must add an additional `free()` at the end to make sure memory does not leak. In Java, this pattern is safe as *h* will get garbage collected after it goes out of scope.

```
fn show_append() {
    let s = String::from("two");
    let t = String::from("three");
    let st = append2(s, t); // append2() gains ownership of s,t
    let ts = append2(t, s); // ownership lost and compiler forbids
    ↪  reuse
    println!("{st}", st);
    println!("{ts}", ts);
}

fn append2(x: String, y: String) -> String {
    let mut z = String::new();
    z.push_str(&x);
    z.push_str(&y);
    return z;
} // x,y dropped here and deallocated
```

Listing 77: Rust Bad 1

We see that this code breaks as we reuse a String, which can only have one owner. One fix is to **clone** the data. Some types implement Copy, which allows them to be easily copied from place to place. However, String does not implement Copy, and so we cannot Copy it into append2.

```
    let st = append2(s.clone(), t.clone());
    let ts = append2(t.clone(), s.clone());
```

Listing 78: Rust Clone

Here both the strings are cloned, or their data contents are duplicated. However, this is inefficient. Rust also allows us to use references, which makes this efficient.

```
fn show_append() {
    let s = String::from("two");
    let t = String::from("three");
    let st = append2(&s, &t);
    let ts = append2(&t, &s);
    println!("{st}", st);
    println!("{ts}", ts);
}

fn append2(x: &String, y: &String) -> String {
    let mut z = String::new();
    z.push_str(&x);
    z.push_str(&y);
    return z;
}
```

Listing 79: An Append Function

Here, append2 “borrows” ownership of s and t, and then returns it at the end of the function. We can also have mutable references.

```
fn add_some(vec: &mut Vec<i32>) {
    for i in 1..=3 {
        vec.push(i);
    }
}

fn main() {
    let mut v = vec![10, 11];
    add_some(&mut v);
    add_some(&mut v);
    println!("{:?}", v);
}
```

Listing 80: Mutable Ref

Programs can only have 1 mutable reference to an object at a time, OR as many immutable refs as desired. Note in the previous example we used the Vec, which is Rust's data structure similar to ArrayList or python lists.

10.5 Slices

Rust has **slices**, which consist of a borrowed portion of a data structure with a length. For example `&[i32]` is a slice of i32, which is equivalent to an array of i32s. This extends to strings, allowing us to create string slices with `&str`, equivalent to a char array.

```rust
fn main() {
    let a = "hello world";
    let b = String::from("hello world");

    let mut c = "goodbye mut";
    let mut d = String::from("goodbye mut");

    for (i, ch) in c.chars().enumerate() {
        println!("c[{i}]: {ch}");
    }

    for (i, ch) in d.chars().enumerate() {
        println!("d[{i}]: {ch}");
    }

    let cs1: &str = &c[2..11];
    let ds1: &str = &d[2..11];

    let clen = c.len();
    let dlen = d.len();

    d.push_str(" again");
    d.replace_range(0..0, "And ");
```

Listing 81: String and str

10.6 Struct and Enum

We can create a product type with

```rust
struct Omelet {
    cook_time: f32,
    is_cooked: bool,
    ingredients: String,
}
```

Listing 82: Rust Struct

And we can create sum types with

```rust
enum Breakfast {
    None,
    Meager(String),
    Hearty(Omelet),
    Misc(u32, String),
}
```

Listing 83: Rust Enum

We can match on enums, and use structs.

```rust
fn break_count(br: &Breakfast) -> u32 {
    match br {
        Breakfast::None => 0,
        Breakfast::Misc(count, _) => *count,
        _ => 1
    }
}

fn main() {
    let dog_br = Breakfast::Meager(String::from("kibble"));
    let my_br = Breakfast::Hearty(Omelet { cook_time: 5.00,
                                  is_cooked: true,
                                  ingredients: String::from("egg") });
}
```

Listing 84: Match

10.7 Traits and Impl

- Rust is not object oriented, but it can feel that way
- Rust favors an approach similar to C++, define a struct and associated functions using `impl`
- Rust supports syntactic sugar around impl such as method invocation
- Several impl can exist for a struct

```rust
struct Omelet {
    ...;
}

impl Omelet {
    fn new(ingr: &str) -> Omelet {
        ...
    }

    fn cook(&mut self, time: f32) {
        ...
    }
}
```

Listing 85: Rust Impl

```rust
fn main() {
    let omlet: Omelet = Omelet::new("peppers");
    omelet.cook(0.35);
}
```

Listing 86: Using Methods

Remark. A small aside: Rust's "methods" are mostly just functions. They aren't exactly Proper Methods, as they don't use subclassing and dynamic dispatch.

- Traits are a way to indicate that "data with operations XYZ can eb used here"
- For example, the ability to compare data, the ability to copy data

```
struct Omelet {
    ...
}

trait Updateable {
    fn update(&mut self);
}

impl Updateable for Omelet {
    fn update(&mut self) {
        self.cook(0.25);
    }
}
```

Listing 87: Example Trait

- There are many existing traits, such as Iterator, Display
- One important trait is the Copy / Clone triat, seen before

```
impl Display for Omelet {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result {
        write!(f, "Omelet (ingredients: {})", self.ingredients)
    }
}

fn main() {
    let om: Omelet = Omelet::new();
    println!("{}", om); // Can now print due to Display
}
```

Listing 88: Display Trait

```
impl Updateable for i32 {
    fn update(&mut self) {
        *self = *self + 1;
    }
}
```

Listing 89: New Trait for Existing Type

We cannot implement existing traits for existing data type implementations. However, we can create a wrapper type to implement the trait for us.

With traits, we can now specify them in generic code.

```
fn show_it<T: Display>(thing: T) {
    println!("The thing is {}", thing);
}

fn show_it2(thing: impl Display) {
    println!("The thing is {}", thing);
}

fn show_it3<T>(thing: T)
where T: Display
{
    println!("The thing is {}", thing);
}
```

Listing 90: Generic with Trait

10.8 Lifetimes

What do we do in this scenario?

```rust
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Listing 91: Rust Funky

We don't know whether the reference being returned refers to an x or a y. We don't know the scopes of x and y, and whether the reference we return is valid. Therefore, we must annotate this with explicit **lifetimes**.

- **'a** is a lifetime
- `x: &'a str` indicates that x's referenced data has a lifetime **at least as long as 'a**.

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

Listing 92: Rust Funky Fix

The return value must live at least as long as the parameters.