Contents

Chapter 1

Introduction to Programming Languages

1.1 Introduction and Motivation

We start with the story of Donald Knuth, who invented TEX, he was trying to publish a CS textbook, and wrote some code to do so. However, one issue was that everytime he would have to modify the code he would also have to modify the documentation. So he developed *Literate programming* which involved keeping documentation along side the code and then having a separate program that would automatically generate the documentation and code. To make Literate programming more popular he wanted to publish a paper about it, Mcllroy asked Knuth to solve the following problem:

Given a text file and integer k, print the k most common words in the file (and the number of their occurrences) in decreasing frequency

Knuth wrote a large pascal program using literate programming that implemented a trie, 26-element arrays, and insertion-sort. At the end of the paper Mcllroy add a side note of a one line command using shell (which he invented): tr -cs A-Za-z '[\n]*' | sort | uniq -c | sort -rn

This displays the fundamental exploration of programming languages, which way should this problem be solved? The shell approach is more debugabble, and easier to understand. However since the pipe expects the previous argument to finish it is not parallelized unlike the pascal approach.

Sapir-Whorf hypothesis: states that the structural diversity of languages is limitless, and the language we use to some extent determines how we think. The same applies to program languages and our goal is to understand how to approach programming problems in different methods.

1.2 PL for Business

From a business standpoint we think about the costs of a language:

- Maintenance: time to write + debug code
- Efficiency at run time (CPU + memory usage and compile time)
- Cost of training and learning
- Reliability
- Portability (how easy is it to move between platforms)
- Testability
- Scalability
- Security and Privacy

These are all from a business standpoint, but we will judge languages from a developers perspective.

1.3 Judging a Language

The mantra for a successful language is *"Successful Languages Evolve"*

- Orthogonality: a language design is said to be orthogonal if one feature does not affect the choice or usage of another feature. For example, in C++ a *non-orthogonal* design is that we cannot return arrays or functions.
- Efficiency: C++ $>$ Java $\approx$ OCaml $>$ Python. OCaml is around C++ when its compiled, and around the speed of Python when it's interpreted. We can have two efficiency metrics (build versus run time), RAM usage, CPU Time, Power and Energy, Network Access, and I/O.
- Simplicity: makes learning and documentation better, but we can also make a language very simple but this can make it also very had to understand when the programs get larger.
- Safety: we would like programs to be safe, compile-time checking, we will know whether our program will work even before we have to run it. There also exist run time checks, and then undefined behaviour (which C++ uses).
- Mutability: how easy is it to add extensions to the language, and how easy is it to add new features
- Stability: we want old programs to keep working one of the biggest issues here is when new key words get introduced.

1.4 Categorization of PL

- Imperative: we focus on commands (a series of instructions that we issue to the computer and executes in the order that we tell it to. This is the most popular programming sub type (C/C++, Java, Python). This style is also the closest to machine code, however due to the ordered nature of it, it severely limits parallelism, and we can only run as fast as the CPU can interact with memory.
- Functional Languages: the focus is on functions, we chain commands via function calls. To encourage parallelism we don't allow side effects (no assignment statements). This way of thinking also provides more clarity (since it is a more mathematical). *Referential Transparency* we don't ever change the value of variable.
- Logic Languages: we don't have side effects, and no function calls. We have predicates (a statement that can either be true of false), and we connect these statements using & | !, etc. This also has a lot in common with DB query languages.

As a side note we note that I/O without side effects, so both functional and logic languages have escape hatches for this in something called a Read Eval Print Loop (REPL).

Chapter 2

OCaml and Functional Programming

2.1 Definition and Motivation

Functional programming is a programming paradigm in which we try to bind everything in pure mathematical functions style. It is a declarative type of programming style. Its main focus is on "what to solve" in contrast to an imperative style where the main focus is "how to solve". It uses expressions instead of statements

Some of the benefits of a functional style are paralleizability (so we can escape the von neumann bottleneck). Functional style also provides more clarity from a mathematical style. For example in math $i = i + 1$ is not true, but we use this all the time in an imperative style. Thus if we see two $i + j$ in our code these can evaluate to two different expressions, however in a functional style we know that the identifiers will always have the same value (*referential transparency*).

2.2 Properties of ML

ML has compile-time type checking (static type checking) it checks all the types before we run the program. OCaml has a lot type inference (now in Java & C++ using auto) (and thus the compiler will figure out the types for you). There is no memory management, there is no del or free argument and thus makes it more reliable for a run time cost. It also has support higher-order functions.

```
# let x = 37*37;;
x : int = 1369
# if x < 2000 then "a" else "b";;
- : string = a
# if x < 2000 then "a" else 27;;
- Error message for type mismatch
```

2.3 Tuples and Lists in OCaml

Tuples can have heterogeneous types (multiple different values) while lists must have the same type but can change in length. Tuples are of fixed length.

```
# 1, "a";;
-: int * string = (1, "a")
# 1, "a", 3.5;;
-: int*string*float = (1,"a",3.5)
# [1; -3; 17]
-: int list = [1; -3; 17]
```

The unit type tuple has nothing in it, and the empty list doesn't necessarily have a type of int or string or anything else so we use 'a to note that it is a list of something (a generic type)

```
# ();;
-: unit = ()
# [];;
-: 'a list = []
# 3::[4; -2; 7]
-: int list = [3; 4; -2; 7]
```

Thus when we use the empty list along side an int list, we will have type inference for the empty list to become an empty list of integers. We can use **::** to add a type to the beginning of an existing list. We can use pattern matching to break apart lists using the same pattern (::). A common idiom in OCaml is that *whatever we use to construct something we can also use to take it apart*.

2.4 Pattern Matching

- Cosntants (0, []) will only match themselves
- Identifiers (x,z) will match anything but then binds the ID to what is matched
- _ matches anything but discards the result
- P::Q matches a non-empty list where P is a pattern to match the first element, and Q is the pattern that matches the tail.
- P,Q,R matches a tuple with patterns P,Q,R
- [P; Q; R] matches a three entry list with patterns P,Q,R

As a more complicated example the following pattern will (a,b) :: c :: d: matches a list of pairs with at least two pairs. We use the *match* keyword to match patterns.

2.5 Functions

The arrow syntax is used to represent the fact that we start at a domain of a and our output is in a range of b. The < fun > syntax is supposed to represent the machine code and since that would be to messy we just output this to represent the implementation.

```
# let f x = x + 1;;
f: int->int = <fun>
# f 3;;
-: int = 4
```

We will now give an example function in OCaml that can perform the concatenation of a list and a single element

```
# let cons(x,y) = x::y;;
cons: 'a * 'a list -> 'a list = <fun>
```

This above function is the un-natural way of writing in OCaml, instead we would prefer to use currying to construct a higher-order function.

2.6 Currying

Currying is the idea that every n-argument function can be written as a series of functions that take exactly one argument chained together. We do this by constructing higher order functions. As a basic example we take the a function that requires x < y. Here we need to arguments x and y. So we create a function < which will take an argument x and return a function x < which we can now give y as an argument too to perform our less than comparison. As a more concrete example we will rewrite cons as a function ccons.

```
# let ccons x y = x::y;;
ccons: 'a -> 'a list -> 'a list = <fun>
# ccons 3 [3; -2];;
-: int list = [3; 3; -2];;
# let f = ccons "a";;
f: string list -> string list = <fun>
# f ["b"; "c"];;
-: string list = ["a"; "b"; "c"]
```

In OCaML we have that functions are left associative f x y = (f x) y but the arrow operator is right associative eg for cons we would have 'a− > ('a list− > ('a list)).

2.7 Anonymous functions

Anonymous functions are where we don't have to give a name to a function, we can use the *fun* keyword to do this.

```
# fun x -> x + 1;;
-: int -> int = <fun>
```

2.8 Shorthands

Often a lot of our functions will take the following form:

```
# fun x -> match x with
    | 0 -> -1
    | y -> y + 1
```

Since this pattern matching is so common, we can use the function which will automatically match the first argument so this anonymous function would be:

```
# function
    | 0 -> -1
    | y -> y+1
```

Here is the following short hand for currying, the top is the expanded form and the bottom is the shorthand.

```
(* Long form of currying *)
# fun x -> (fun y -> (x+y+1))
(* Short form *)
# fun x y -> x + y + 1
```

We can similarly define named functions by using the let keyword and then setting it equal to *fun*.

2.9 Pitfall of Currying

Suppose we only have access to only the $<$ operator, and we want to write a function to determine if a number is positive or negative, observe that we would have to swap the order of the arguments:

```
#let ispos  = (<) 0;;
ispos: int -> bool = <fun>
#let isneg = (<) x 0;;
isneg: int -> bool = <fun>
```

Here in this case we observe that currying may not be the best since we have to swap the order of the arguments in-order to make the function work.

Venkat's side note: Currying can cause a bit of a computational overhead, since each time we add an argument it will get treated as a separate function call it can add more computational and extra memory strain.

2.10 Recursion

In Ocaml, we need to use the *rec* keyword when we define a function to indicate that it's recursive. We motivate recursion and it's pitfalls using a simple problem: reversing an OCaml list.

```
let rec reverse l = match l with
    | [] -> []
    (* Here the @ operator indicates we are combining two lists *)
    | h::t -> (reverse t) @ h
```

This code wouldn't work...why? Because the @ operator expects both sides to be a list, this implies that h is a list, hence the type of this function is 'a list list$-$ $>$ 'a list. So instead we need to do:

```
let rec reverse l = match l with
    | [] -> []
    (* Here the @ operator indicates we are combining two lists *)
    | h::t -> (reverse t) @ [h]
```

Now we observe that the @ operator is O(n) compared to :: which is O(1). So our reverse function runs in $O(n^2)$. We can fix this by using the idea of an accumulator:

```
let reverse =
    let rec rev accum = function
        | [] -> accum
        | h::t -> rev (h::accum) t
    in rev []
```

We also write a general max function, where we will pass in an identity element and a comparison function so that it can work for any type, integers, floats, etc etc.

```
# let rec max i lt = function
    | [] -> i
    | h::t -> let m = max i lt t in
              if lt h m then m else h
val max: 'a -> ('a -> 'a -> bool) -> a' list -> 'a = <fun>
# let imax = max -99999999 (<)
```

We motivate the use of the rec keyword from a mathematical standpoint, we could not possibly define a variable x in terms of itself, this would be paradoxical. Thus we use the rec keyword to indicate this to the Ocaml compiler so that it doesn't treat it like this.

2.11 Custom Types

Types are like discriminant unions, in memory there are two pieces a type tag, and then the second part of the object is the tag value. Here is how we define the built in option type.

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

The option type is similar to the nullptr in c++, however it's nicer since we ocaml would force us to have a pattern match for the None so we would never access something when it is None (unlike c++). We could similarly define a list in the following manner:

```
type 'a mylist =
    | Empty
    | Nonempty of 'a * 'a mylist
let consmylist a l =
    Nonempty (a, l)
```

What is the difference between a C++ union and types in ocaml. In C++ a union is overlayed in memory, while in ocaml the type flag will specify which version of the type is stored. As an example of a union in c++, here if the user loads a float into an int we would get garbage, but this is not possible in ocaml.

```
// C++
union u {int a; float b;};

(* ocaml *)
type u =
    | A of int
    | B of float
```

Another structure (not nessecarily a type) is a structure, below is an example of one:

```
# type person =
    {first_name : string;
     surname : string;
     mutable age : int};;
    type person = { first_name : string; surname : string; mutable age : int; }
# let birthday p =
    p.age <- p.age + 1;;
    val birthday : person -> unit = <fun>
```

And an example of a simple custom type with some pattern matching:

```
# type colour = Red | Blue ;;
type colour = Red | Blue
# let example c =
    match c with
    | Red -> "rose"
    | Blue -> "sky"
val example : colour -> string = <fun>
```

Chapter 3

Grammars and Syntax

3.1 Defining Syntax

Syntax is form independent of meaning. Programs have two levels of syntax, we have *lexical analysis* where if we turn a series of words into a series of tokens (it will throw away all white space and new lines).

```
    int main(void) { return !getchar(); }
//  INT ID (VOID) { RETURN ! ID  ( ) ; }
//  ___ main ____ { _____ ! getchar () ; }
```

Lexical Analysis will not just provide tokens, but also some extra associated information which put together are called lexemes. For example we may have a token ”num” that gets associated with an integer value eg. 179.
Our tokenizers are greedy and will grab the first largest token it can, thus the following C++ code will not be interpreted as valid

```
    return a+++++b
```

This is because a greedy tokenizer would parse this as RETURN, ID, ++, ++, +, ID. However a parse such as RETURN, ID, ++, +, ++, ID would be valid.

Surprisingly white space and comments can still be a major issue. First we address the fact that some languages support international characters, so o can be different from o, one could be Cyrillic and we wouldn’t be able to visibly tell the difference. Another issue is a trigraph (symbols in C/C++) that translate such as a new line and thus if placed at the end of a comment can end up commenting out the next line as well.

3.2 BNF and EBNF

We use a grammar to generate a parse tree, and we valid generation of the parse tree would indicate that we have a valid parse that satisfies our grammars. The classic BNF form would have a Non-terminal on the left hand side (lhs) that could lead to set of terminals or non-terminals on the right hand side (rhs). EBNF introduces extra notation to simplify the grammar but can always be translated back into BNF. Some notations include:

1. — or
2. * 0 or more following token
3. + 1 or more
4. () groups expression
5. [] specifies range of values

We often will have a grammar written in EBNF then some pre-processing that converts it into BNF and then we have that grammar run in our parser.

3.3 Sample XML Grammar

Note that for a general program that we will operate at two levels the terminal symbols will be tokens which lead to words, but for the XML grammar we immediately just can parse the characters.

```
CharData ::= [^<&]* (* Length of 0 or more of characters that don't include < and & *)
CharRef ::= '&#'([0-9]+ | 'x'[0-9a-fA-F]+)';'
EntityRef ::= '&'Name';'
Reference ::= EntityRef | CharRef
Eq ::= S? '=' S? (* ? means optional *)
S ::= (#x20 | #x9 | #xD | #xA)+
AttValue ::= '"'([^<&'] | Reference)*'"' | "'"([^<&'] | Reference)*"'"
NameChar ::= NameStartChar | "-" | "." | [0-9]
NameStartChar ::= ":"|[A-Za-z] | "_"
Name ::= NameStartChar (NameChar)*
Attribute ::= Name Eq AttValue
Stag ::= '<' Name (S Attribute)* S? '>' (* S is a single space*)
Etag ::= '</' Name S? '>'
EmptyElemTag ::= '<' Name (S Attribute)* S? '/>'
content ::= CharData? ((element | Reference) CharData?)*
element ::= EmptyElemTag | Stag content ETag (* element is the start *)
```

Observe that the last two grammar rules are mutually recursive, we can build a regex for all of the other expressions (regular language) that are not mutually recursive.

3.4 ISO Standard for EBNF

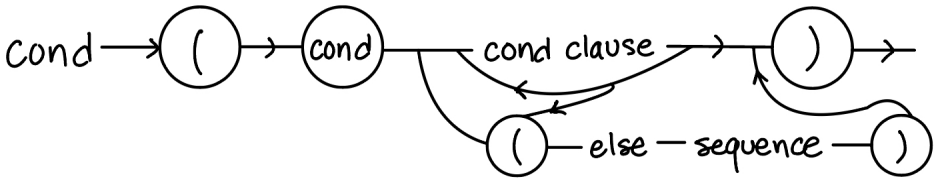
The ISO standard attempts to give a standard format for for every EBNF grammar should be written. We write a terminal as "terminal", optional items (0 or 1 instances): [option], repetition (0 or more instances) as {repetition}, and () for grouping. Alternate ways to specify $n * X$ means repeat n times, $X - Y$ is the set difference operator part of X but not Y. Then X,Y is concatenation, and X—Y is disjunction. And we use X = Y; for a rule.

We can actually define the ISO EBNF grammar in terms of itself! While this is a circular definition, some extra text just helped the authors of the ISO standard prove that their grammar also works.

```
syntax = syntax rule, {syntax rule}
syntax rule = meta id, '=', definitions list, ';';
definitions list = definition,{'|',definition};
...
```

3.5 Syntax Diagrams

Instead of writing a rule, we instead show a diagram which shows how we would parse an expression, as an example here is the cond parse in Scheme:



This can be more simple and easier to understand compared to the EBNF, however it is also an image and as a result it could cause us to do our own transcription. However, it is nice in the fact that it suggest a proper way for use to write a parser.

```
def parser_cond():
    eat("(")
    eat("cond")
    if next_two_symbols( ["(", "else"] ):
        eat("(")
        eat("else")
        parse_sequence()
        eat(")")
    else:
        while parse_cond_clause():
            continue
        eat(")")
```

3.6 What can go wrong with grammars?

Just as we can write a bad code, we can write bad grammars:
$S \to a$
$S \to b$
$T \to c$
Here if S is our start symbol we can never reach T, it is *unreachable*. Below we give an example of a grammar that generates a b followed by any number of As:
$S \to Sa$
$S \to b$
$S \to S$
$S \to aTb$
Take a look at the last two rules, they are useless, S leads to S can cause our parse tree to be much larger than necessary, and also can lead to infinite recursion, the last rule uses a non-terminal symbol T that doesn't go anywhere and thus we will never be able to construct a valid parse tree (meaning we cannot construct a list of terminal symbols).
Even if we use the above grammar after removing that last two rules, this grammar is still going to be problematic... in terms of code:

```
def S():
    return [S(); a()] || b()
```

This grammar, is correct from a theory standpoint, however this grammar will recurse forever, since it is a left recursive grammar as seen by writing the function calls we will make. We can make any left recursive grammar to be right recursive.
$S \to bT$
$T \to aT$
$T \to \epsilon$
Would be the corresponding right recursive grammar, since we take a piece of the string before we recurse (assuming that our input is of finite length.

The grammar is a formal specification of the syntax of a language. Thus, sometimes there may be syntax constraints that are not captured by the grammar. As we capture certain constraints the size of grammar may end up growing exponentially.

```
// C++ sample code
typedef int t;
t a;
```

The above code is valid c++ code since we declare a to have an identifier of t, however we would first have to parse t, to be able to know that a is a valid parse. Python tokenization is also more complicated since it has to deal with indentation.

3.7 Fixing wrong Grammars

Sometimes grammars can be too generous, and thus can allow parses that should not be valid. Here is one such toy grammar:

```
S → NP VP .
NP → N
NP → Adj NP
VP → V
VP → VP Adv
Adj → blue | green | angry
Adv → slowly | quickly | anxiously
N → dog | dogs | cat | cats
V → bark | barks | meow | meows
```

Note that this grammar is pretty good, it allows us to saw a phrase like ”blue dogs bark” and ”angry green cat meows anxiously”. However it still allows sentences that don’t make sense! Here is one two such examples: ”blue dog bark”, ”cats meows”. This has an issue to do with the pluralization of a noun and then the associated verb (singular versus plural). We can fix this we double the size of grammar, separate rules for plural versus singular.

```
S → SNP SVP .
S → PNP PVP .
SNP → SN | Adj SNP
SVP → SV | SVP Adv
PNP → PN | Adj PNP
PVP → PV | PVP Adv
Adj → blue | green | angry
Adv → slowly | quickly | anxiously
PN → dogs | cats
SN → dog | cat
PV → bark | meow
SV → barks | meows
```

Now in other languages if we had to enforce other rules such as gender, we would end up doubling our grammar rules each time. Adding more constraints to our grammar is very expensive and the same would apply to our grammar for a programming language.

3.8 Ambiguous Grammars

Ambiguous grammars allow multiple parse trees for the same input (note that this is different than allowed a wrong input that generates a parse tree). Ambiguous grammars cause trouble since they are too generous, however its not that it allows invalid sentences, it allows different interpretations of valid sentences. This would result in a case where the compiler may not do what we want it to do. (Eliminating ambiguity does not cause the grammar to grow was quickly as constraining a grammar for eliminating invalid sentences).

3.9 Ambiguity in Precedence and Associativity

We provide another example of a toy grammar that is meant for doing addition and multiplication:

```
E -> E + E
E -> E * E
E -> ID
E -> NUM
E -> (E)
```

Now we would have two parse trees for the expression $3 + 4 * a$, which makes this grammar ambiguous.

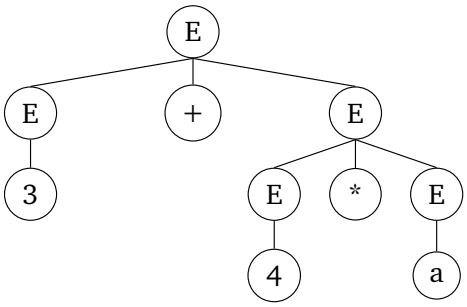


Figure 3.1: Parse tree for $3 + 4 * a$

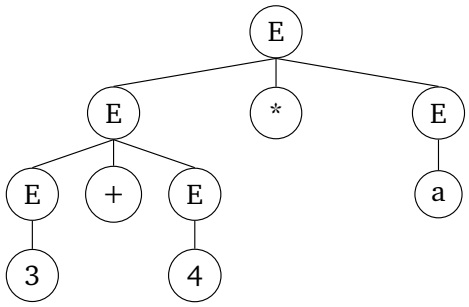


Figure 3.2: Parse tree for $3 + 4 * a$

Observe that the left parse tree is right while the right parse tree is wrong, so we need to fix the ambiguity in the grammar. We can do this by adding a new non-terminal symbol that will separate addition from multiplication.

```
# No Precedence Ambiguity
E -> T
E -> E + E
T -> T * T
T -> ID
T -> NUM
T -> (E)
```

```
# No Precedence or
↪  Associatvity Ambiguity
E -> T
E -> E + E
T -> T * F
T -> F
F -> ID
F -> NUM
F -> (E)
```

Now we have guaranteed that at our multiplication will always take precedence over addition. This is the first case of ambiguity: *precedence*. There is still one other ambiguity, with this grammar, the order of multiplication. This can be an issue when we think about overflow eg: $(a * b) * c$ could overflow while $a * (b * c)$ may not! This brings up the issue of ambiguity via *associativity* of multiplication.

3.10 The C Grammar

Here we provide a more simple version of the C grammar to illustrate one of the most common programming grammar ambiguities.

```
expr-opt:
    expr;
    epsilon
stmt:
    ;
    expr;
    break;
    continue;
    return;
    return expr;
    goto ID;
    {stmnt-list} # {} are tokens here
    while (expr) stmt # () are tokens here
    do stmt while (expr);
    switch (expr) stmt
    ID: stmt # Labeled statement
    case expr: stmt
    default stmt
    for(expr-opt; expr-opt; expr-opt) stmt
    if (expr) stmt
    if (expr) stmt else stmt
```

We first discuss the issue of why we need () with our while statement, this is because we need to be able to tell when our expression ends and starts, take the following example.

```
while i < n
    ++i;
```

We can parse this in one of two ways $(i < n)$ and $++i$; or we could have $(i < n++)$ and i; And thus we need the parenthesis for our while statement. Note: we don't need the parenthesis for the do while statement but they were kept in to keep the language consistent.

Dangling if & else problem: If if stmt else stmt, could be matched in two ways: if (if stmt else stmt) or if (if stmt) else stmt. So our current grammar is ambiguous. We need to ensure that our grammar will match the last else to the closest if, otherwise we may have multiple interpretations of what the grammar will do. We can fix this by adding another non-terminal symbol:

```
    stmt:
        if (expr) stmt else stmt1
    stmt1:
        if (expr) stmt1
        stmt
```

3.11 Parsing Programs

Software Tooling Approach: Here is how gcc compiles a c file, we follow this process.

1. foo.c is read by the c pre-processor to generate foo.i which is a file with all include statements removed and macros expanded out (no # lines)
2. foo.i is translated by cc1 to foo.s (which is assembly language)
3. foo.s is translated by the assembled into foo.o which is object code (up till this point we had text, now everything is binary)
4. foo.o has some gaps for certain functions like the sign function and the linker (ld) takes the c-library and combines them into ./aout which has no gaps.
5. The OS loader then takes this executable file and then moves it into RAM.

One upside to this approach is it makes it very easy to know which issues exist and then debug according parts or swap out parts. However, one issue is that we expose a lot of complexity to the user and then this is also the slowest approach.

IDE (Integrated Dev Environment): we want to make the process of changing a line of code and we can immediately run as fast as possible.

Static Checking: we can use a parser to see if program is correct even before we have to try and run it.

Dynamic Linking vs Static Linking: in dynamic linking we load the library and its corresponding symbols into memory at run time, where as with static linking we copy the library code into the executable at run time.

3.12 Compilers vs. Interpreters

Compiler vs. Interpreter: A compiler translates source to machine code and then runs the machine code. An Interpreter copies the source code into RAM, and then interprets the parse tree in ram and executes the instruction based on what the parse tree specifies. Compilers tend to be faster at run time, while interpreters are faster to get started, more debuggable, and more portable.

Just in Time (JIT) compiler: attempts to combine ideas of both a compiler and interpreter. We start off with an interpreter, this is given a Java.class file which represent bytecodes for the interpreter. Since the interpreter is now a simple machine (not a physical one) it can understand these byte codes and then execute each of these instructions. As this interpreter runs, it keeps track of each instruction it executes and how often. This is called a profile where the y-axis is the memory address and then the x-axis is how often the instruction gets executed (will be a histogram). The highest frequency portions are called hot spots, these hot spots are then translated into machine code just for that hot spot. We only compile instructions that get executed a lot, and the portability of bytecodes.

Chapter 4

Types:

4.1 Defining a Type

There a couple definitions that we can view types from:

1. They are a data format (int specifies a specific layout in the machine)
2. A way of characterizing what memory is used for
3. An identifier for 1 or 2
4. A set of values (set of possible floats etc.)
5. A set of values and their associated operations.
6. What you know about an expression statically (before running)

The standard mantra of OOP design, is that we cannot design a type unless we know the operations that we can do for a specific type. **Abstract vs. Exposed** types. With exposed types we know the exact memory representation of the type (helps with efficiency), with abstract types we will not know the implementation of the type but we know the operations we can perform (this is more modular). If we are able to tell the types however the speed will increase (since we help the compiler know which instructions are corresponding to the instruction). Having types also makes documentation and annotation of the reader easier (human & non-human).
Static vs Dynamic Type Checking: C, C++, Java, OCaml all perform static type checking, we will know the types even before we run. Python and scheme do dynamic type checking we may not know the types until run time. However these are at the ends of a spectrum:

- For example Python allows code to define a type and then conducts static type checking
- Java allows the instantiation of Objects, and then will check at run-time whether a method is available.

Java, Python, OCaml are all strongly typed, where as C, C++ are not strongly typed, since we can cast between types of pointers etc.

4.2 Why do we want types?

Having types allow us to ensure the correctness of operations for example $a + b$ would need to generate the correct + instruction for the computer (one for ints, floats, strings, etc). *BCPL* is a one type language (everything was an integer), this makes it simpler, but also would cause less abstraction.

4.3 Aside on floats

- We think this is a 32-bit floating point number.
- That IEEE 32-bit floating point standard is: 1 sign bit, 8 exponent bits, and 23 bits for the mantissa (f)

$\pm\, 2 \times 10^{-127} \times 1.f$	$0 < e < 255$
$\pm\, 2^{-126} \times 0.f$	$e = 0$ (tiny numbers)
$\pm\, \infty$	$e = 255, f = 0$
$\pm$ NaN(f)	$e = 255, f \neq 0$

This number will not overflow; it goes to ∞ if we multiply by a big enough number. However, underflowing by dividing by 2 constantly will underflow to 0, which doesn't give us an indication of underflow like ∞ does.
Note that -0.0 represents underflow from the negative direction.

What's NaN for? 0.0 / 0.0. and $\infty \times 0.0$ and any operation with NaN. Division by 0.0 in general just gives infinity.

One school of thought says NaN should not be allowed in the floating point type; why would a type that says it's a number have the ability to not be a number? Proponents here would advocate that NaN calculations should throw an exception.

On x86 (and probably other hardware) there's the ability to set a bit to 1 that raises exceptions when NaN or overflow calculations happen; this runs parallel to the above school of thought. Part of why this isn't popularized is merely inertia; FORTRAN was the OG language at the time and didn't have exception handling.

The reason we have such tiny numbers (2^{-126} * .0000...0001) enabled the functionality of (a = b) iff (a - b = 0). Since this is desired behavior. Subtracting numbers is quite nice. Otherwise without tiny numbers we could have a case of $a \neq b$ but $a - b = 0$

Main takeaway: Type debates will often be choosing between special values and exception handling.

4.4 Type equivalence

Are two types $T\&U$ equivalent to each other.

```
struct s {double re; im;};
struct t {double re; im;};
typedef struct s u;
```

- Name Equivalence: The above two types would be different (C uses this).
- Structural Equivalence: based on the structure the above two types would be the same

In C we use name equivalence for struct definitions, and structural equivalence for typedefs. However structural equivalence can be very difficult to tell apart, for example one place of ambiguity would be in the following struct definitions:

```
struct a {struct b*next; int v;};
struct b (struct a*next; int v;};
```

4.5 Sub-Type Equivalence

We want to know whether T is a subset of type U, which means that every type T is also a value of type U, and as a result every operation on U is also an operation on T, but T may have a *superset* of operations.
Subtypes can have issues of change of representation, eg. converting between int and float which would require machine code for the conversion. But we also may have no change of representation as well, eg in Java, changing can just be done via pointer assignment.

4.6 Ad Hoc Polymorphism

When a function can accept and return many forms, for example the $\sin(x)$ function can be a float→float or it could also be a double→double function. This is known as **overloading**. How would we know at the assembly language level which version of the function to call. This is resolved via *name mangling*, we rename the functions at the assemble level eg. $\sin\$f$ and $\sin\$d$. However this assumes that name mangling is consistent.

Coercion: we attempt to convert values from one type to another. We take the following example code:

```
float f = __;
double d = cos(f);
```

In which way do we do our conversion, do we convert the float to a double and then call cos as a double (this gives more precision) or do call cos on a float and then convert it to a double, this provides speed.

4.7 Parametric Polymorphism

A type of function that includes type variables, we motivate this using Java which has both types of poylmorphism (we write a program that should remove all short strings)

```
static void remove_short (Collection c) {
    for(Iterator i = c.iterator(); i.hasNext();) {
        if( (String)(i.next()).length() < 10 )
            i.remove();
    }
}
```

Without the string cast i.next would return an object, however an object doesn't have a length method and thus would fail. So the string cast would be a *runtime* check to see if the object is actually a string and then would continue on. Now we rewrite this with parametric polymorphism:

```
static void remove_short(Collection<String> c) {
    for(Iterator<String> i = c.iterator(); i.hasNext();) {
        if( i.next().length() < 10 )
            i.remove();
    }
}
```

This code is better since now we would only allow a collection of strings.

4.8 Templates vs Generics

Templates are used in C++, Ada while generic are used in OCaml, Java. Templates don't actually compile the code, they would wait till the template is actually used to generate the machine code. Generics compile the code exactly once to machine code, and we check the types as we go, we don't care what T and U are. This assumes that all types "smell the same", for example each object is just a pointer. Generics provide more abstraction.

4.9 Example Sub-Type

```
typedef char *T;
typedef char const *U;
char ch;
char const *p = &ch;
char *q = (char *) p;
*q = x;
```

The const in c++ indicates that we will not modify the memory location via that specific pointer, so if we define $T\ x$; and $U\ y$. Note that we can't do $x = y$ but we can do $y = x$. Thus char * is a subtype of char const *. This becomes more evident if we consider the

memory layouts that char const * and char * would access (const * has wider access since it can access read-only and read-write while char * can only point to read-write).

4.10 Sub-Types & Wildcards

```
List<String> ls = ....;
List<Object> lo = ls;
lo.add(new Thread());
String s = ls.get();
```

Observe that lines 1, 3, 4 all make sense, however line 2 is not correct, since a list of strings are not a subset of list of objects.

If S is a subtype of T does not imply that List$< S >$ is a subtype of List$< T >$.

This motivates the use of wildcards, if we want to generalize code so that it works for anything we need to use a wildcard:

```
void printall(Collection<?> c) {
    for (Object i : c)
        System.out.println(c);
}
```

In the above code if we replaced the ? with Object, we wouldn't be able to call it on a Collection<String> since it is not a subtype of Collection<Object>. Thus we need to use the wildcard.

4.11 Named Type Variables

If we wanted to write a piece of code that converts an array of one type to a collection of another type, can we use a wildcard? No, because the wildcard would allow them to be two different types without every guaranteeing that they have the same type or are subtypes of each other. Thus we would write it as:

```
void<T> convert(T [] a, Collection<T> c) {
    for (T i : a)
        c.add(i)
}

String [] test;
Collection<Object> mod;
convert(test, mod);
```

However, observe that the last three lines would not work since convert would expect the same type, even though every string is still an object, so we could write it as:

```
void<T, U> convert(T [] a, Collection<U super T> c) {
    for (T i : a)
        c.add(i)
}
```

This means that U must by a super-type of T, the alternative notation *T extends U* means that T must be a super-type of U, and this allows us to impose an constrain on the relation between T and U.
We can take any code that uses ? and now rewrite it using this method without a ?.

4.12 Generic Types at Runtime

Each object consists of a value field and then a type field, however for generics it's not enough to have a single type field, we would need a type descriptor. This can cause efficiency issues since we would need to run down the entire tree of type fields to ensure that everything matches. And this would be a large run time cost for the interpreter.

Instead Java uses *type erasure*, it well omit information about the exact type details at run time. For example if we have List $<$ String $>$ the type descriptor may just contain List, and only match that. This is good for run time, however if we access elements from a collection the compiler will insert a runtime check the ensure the proper type.

4.13 Duck Typing

Look at what an object *does*, not what it is. "If it walks like a duck and it quacks like a duck, it's a duck." Some examples are Smalltalk, Python, Ruby, JS. This is more like behaviour checking, we see if we can call a method. This can be helpful for small programs, but it's better to have static type checking for larger code.

```
class Duck:
    def swim(self):
        print("Duck swimming")

    def fly(self):
        print("Duck flying")

class Whale:
    def swim(self):
        print("Whale swimming")

for animal in [Duck(), Whale()]:
    animal.swim()
    animal.fly()
```

4.14 Identifiers and Bindings

A Name or identifier is specific per programming language. Bindings is the association between a name and value.

```
int n = 29;
int f(...) {
    &n = ...
    sizeof n;
    typeof(n) i = n
}
```

Here n is not just bound to 29, since then &n wouldn't make any sense. Thus n is bound to an integer value, an address, # of bytes in n, and it's own type.
Binding time the values that n gets bound to can happen at different times:

- Type: bound at compile time
- Address: Link Time, or for PIE at load time, if n is declared in a function we will only know the address and block entry time (when the function is called at run time).
- n-value: runtime

4.15 Scope

The scope of a name is the set of program locations where the name is visible. C++ and Java use a block structured scope indicated by { and }.

```
int f() {
    struct f {int f;} f;
    enum f {g};
    struct g {int f; } f;
    #include <f>
    #define f f
    f: goto f;
}
```

Primitive Namespaces: we have multiple namespaces (we check for conflicts within a namespace): Ordinary namespace (contains vars, functions), struct namespace, struct member namespace, enum namespaces. These all follow block structure rules. The label namespace is defined by a function scope.

Labeled Namespaces: a named collection of bindings. We bind a name to a namespace and this binding is in a parent namespace. This allows to handle naming conflicts as well if we have multiple similar function definitions in similar modules.

```
#Start of foo file:
n = 1
def f(x):
class c:
```

```
import foo
foo.f
foo.n
foo.c
```

Foo.py will create bindings for n, f, c. The caller will then be able to acess these since foo is bound the the namespace. Python namespaces work via a dictionary and we can then look these up using the dictionary at run-time. C++ & Ocaml use compile time namespaces.

4.16 Information Hiding

We want to restrict who can use which names in our namespace. The two ways to do it are:

- We label each name with a security classification. In Java there are three markers:

	Class	Package	Sub-classes	Everything
public	✓	✓	✓	✓
protected	✓	✓	✓	x
default	✓	✓	x	x
private	✓	x	x	x

- The above method, enforces and sticks to closely to the object hierarchy, instead namespaces have a specific notation. In Java these are interfaces, and in Ocaml these are called signatures. We basically provide an only what the user needs to know to utilize a a module.

```
module type Q =
    sig type 'a queue
    pop 'a queue -> 'a
end
module type M =
    struct type 'a queue = ...
    let pop q' = ....
end
```

  Now we can have a signature that can correspond to many different modules (eg. queue, priority queue) or we can have a module that can correspond to many different signatures. In the above example M is the detailed implementation, while Q is the interface where it's an API for the actual code.

Chapter 5

Java

5.1 A motivation

How do we measure the speed of a computer? We use the LINPACK benchmark which gives the flops *number of floating point operations per second*. Another benchmark can be could be how many flops we get per watt of energy is consumed, this list is called the Green 100. But why do we care? In around 40 years the super computers of today will be the more standard, and we need to ensure that our programming language can make the best use of these machines.

5.2 C/C++ Issues

There are many issues with C++ that the developers of Java wanted to improve and fix:

1. The ability to switch architectures quickly, C++ can only be compiled for a specific architecture

2. C++ is too unreliable (reliable code can be written it just requires much more effort) which comes from null-pointers and requiring the programmer to delete objects when they want (either deleting to early to forgeting to late)

3. Bloated executable, C++ generates very large executable files, this is not great for IoT settings where we want to deploy updates

4. And C++ is too complicated, the specification is closer to 1500 pages now.

C++ also has multiple inheritance which can be difficult if two parents disagree on a method call. Reference to dominance for a more information on handling these conflicts.

We note that these cons do not even include the painful multi-threading that actually exists. We provide some other issues that C++ has (most of these make the code more unreliable).

```
double stk[1000];
double *sp = &stk[1000];
#define PUSH(v) (*--sp ==(v))
#define POP() (*sp++)

swtich(op) {
    case MULT: PUSH(POP()*POP());
               break;
}
```

In the above code we have a issue with how the order of execution which based on the compiler and machine can cause the above code not to work because of the pointer arithmetic. The result of Pushing two pops is represented as:

$$*(--sp) = (*sp++) * (*sp++)$$

We can't do this in C++ since the order of evaluation of which operations get done first is not defined. So we actually don't know which pointer value changes would be made and as a result the behaviour is not necessarily consistent.

The C++ pre-processor, transforms C code to macros without macros. Some people find the use of pre-processor to be obsolete and additional clutter.

5.3 History of Java

Java was created by a company called Sun Microsystems that focused on internet of things computation. They initially were programming in C++ but saw many of the issues of C++ that were mentioned above. So they attempted to develop a new language that would fix these issues.

Sun Microsystems decided to take "help" from Xerox Park. Xerox Park had developed a language called Small-talk which had the following features:

1. The interpreter had type checking
2. Ensured that nullptrs could not be accessed and computation such as 1/0 would not happen so the hardware could not trap
3. It also had a garbage collector which would automatically clean up memory
4. Small-talk also used dynamic type checking

The first three points of the small-talk language fixed the issue of C++ being unreliable. However small-talk had a weird syntax and was not parallelized. So Sun-microsystems used small-talk as a starting base. To address the issue of not being portable across architectures, Java gets compiled into bytecode which is then executed. To reduce the complexity of the language, a smaller language C+- was created. So they invented a new language that used the syntax of C++ and much of the Small talk semantics and implementation ideas, but replaced the dynamic type checking with static type checking.

5.4 Java Basics

In Java, all variables are always initialized if we don't provide a value we initialize it to zero. Java defines primitive types which are machine independent, with only 2's complement supported. where we have:

- byte (8 bits)
- short (16 bits)
- int (32 bits)
- float (32 bits)
- long (64 bits)
- double (64 bits)

All other types in Java are called reference types (objects), where we have a pointer that will point to the objects.

Order of evaluation is always left to right in an expression, this means that if we have $a = f(x) + g(y)$, f(x) will always get called first. Unlike C++ where this is never defined. However this means that it cannot be as fast as C++, since we cannot re-order the order of execution.

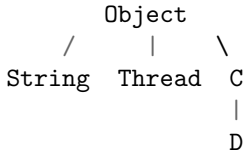
There are no pointers in Java, instead we have reference types which are implemented using pointers but we cannot access these pointers directly. Arrays are reference types, and they are always located on the heap, and functions can return arrays.

```
int guesses = new int[1000]; // size can't change later
guesses = new int[10];
int[][] twoX = .....;
```

5.5 Inheritance in Java

Everything is inherited from the base Object class. String, Thread are all inherited from Object. Java only has single inheritance which means that everything has only one parent! When we don't specify the extends keyword for a class it automatically always extends the object class.

```
class C {
...
}
class D extends C {
...
}
```



The *final* keyword indicates in java that this method can never be overwritten from the parent class.

```
class C {
    final int m() {...}
    abstract int n()
}
```

In the above code we observe that now any class that extends C will not be able to override the method m. The final keyword allows for in-lining (where we replace the call with the method itself) and thus can be an optimization, since we don't have to check the virtual file table if a method has been over-written. However, if we overuse the final keyword they we loose the entire point of using inheritance and OOP.

The *abstract* means that the parent class does not provide an implementation and requires the child class to provide an implementation. And this means that we cannot allocate an object with an abstract class. It can be thought of just providing an API.

5.6 Interfaces

Interfaces are in someways a solution to not allowing multiple parent classes. Interfaces provide an API that a class that implements it must define. We can think of a parent class as passing own a fortune while an interface is passing on a debt.

```
Interface I {
    int m();
    bool n();
    void run();
}
class C implements I,J,M {
    ...
}
```

By using an interface we can now pass an interface as an argument and thus use any argument that implements that interface can be used:

```
void p(I x) {
    x.run()
}
```

Now as long as the object implements the interface and has the run method we can call it (as an example the function p). Interfaces can have their own hierarchy. The object class structure can be a tree, but the graph for interfaces would be much more complex.

```
Interface J {
    float m();
}
```

In this case interface I and J would be conflicting interfaces and the compiler would not allow this. While interfaces avoid the idea of conflicting parents we can still have conflicting interfaces.

5.7 SIMD & MIMD

SIMD (Single Instruction, Multiple Data): This is a type of parallel computing where multiple processors perform the same operation on multiple data points simultaneously.

MIMD (Multiple Instruction, Multiple Data): This is another type of parallel computing where multiple processors execute different instructions on different data at the same time. Unlike SIMD, MIMD systems allow each processor to operate independently, potentially executing different programs or different parts of the same program. Java is part of the MIMD camp, but recent trends have shifted to SIMD.

5.8 Threading model

Java was designed with multi-threading in mind, so how does it achieve this?

Java follows the MIMD model which means multiple instruction and multiple data. Each thread has it's own instruction pointer and. If we create a new object we assume that all the CPUs will now be able to access that object.

A thread object represents a single thread of computation in the SMP environment. The default implementation of the run method in a thread class is to do nothing! Thus to do anything with a thread we need to subclass thread and then overwrite the run method.

```
Thread t = new Thread();
// This thread does nothing
class T extends Thread {
    void run() { .... }
}
Thread t = new T();
// This thread now can do what we specified in the run func
```

However threads are not usually started in this manner since they would split the object hierarchy into two separate parts.

```
Interface Runnable {
    void run()
}
Thread t = new Thread(Runnable v);
```

5.9 Thread Lifecycle

Here we talk about the lifecycle of the thread object:

1. A thread exists but has not started running, state=NEW
2. t.start() will allocate the OS resources for an actual thread, the state=RUNNABLE, meaning it maybe running or is just about to run and is just waiting
3. then a thread can sleep (state=TIMEDWAITING), wait (state=WAITING), I/O (state=BLOCKED) or execute code or yield (state=RUNNABLE)
4. Then we can exit the thread and its state=TERMINATED, however the object still exists for other threads to visit it.

Chapter 6

A Comparison Summary:

6.1 Quick Definitions

Imperative: Utilizes statements that change a program's state by directly instructing how to accomplish a task.
Functional: Focuses on evaluating expressions and using pure functions, emphasizing immutability and avoiding side effects.
Logical: Uses rules and facts to express problem-solving logic, particularly suitable for symbolic computation and AI applications.
Static Type Checking: Type analysis performed at compile-time to detect type errors before runtime.
Static Name checking: Compiler validation of identifier correctness before program execution.
Dynamic Type Checking: Type validation that occurs during program execution, allowing for runtime flexibility.
Lexical or Static Scoping: Determining variable scope based on its position in the source code's textual structure (can be nested).

6.2 C & C++

It is an *imperative* programming language. It has *static name checking*, and *lexical scoping*. It uses manual memory management (free, delete, and pointers). It is faster compared to python and java.

Specialities: portability, close to hardware programming, performance optimization.

Pros: High performance, low level control of hardware resources, many libraries and frameworks, and support for procedural, OOP, and generic programming

Cons: Manual memory management can cause it to be very unreliable. Lack of built in concurrency support which is also very confusing. Complex Syntax. Has more safety issues due to memory corruption errors and buffer overflow attacks.

6.3 Java

It is an *imperative* programming language. it has *static type checking, static scoping, static name checking*. It has automatic memory management via the *garbage collecter* and the *JVM*. Has a similar syntax to C++ but different memory model, it has static type checking unlike Python which is dynamically typed.

Specialities: The Java virtual machine and garbage collection, platform independence, and concurrency.

Pros: Java bytecode allows it it to have more platform independence. It was also built with multi threading and concurrency in mind. Enforces an OO model, with encapsulation, inheritance, and polymorphisim. It also has automatic memory management.

Cons: Performance overhead: Bytecode execution may be slower compared to native code. Verbosity: Java code can be verbose compared to dynamically-typed languages like Python. Limited low-level control: Java's memory management and execution model may not be suitable for performance-critical systems programming tasks.

6.4 Ocaml

It is a *functional* language with, *strong static type checking, type inference, static name checking, lexical scoping, automatic memory management (GC)*

Specialties: no side effects, recursion, currying, higher-order functions, pattern matching, and immutable data structs

Pros: Strong static typing and type inference promote code safety and correctness. expressive and concise syntax. Automatic memory management through GC, no memory-related

bugs. High-performance native code compiler = efficient executables. More expressive and concise than C/C++, compile-time safe than Python

Cons: Limited tooling/ecosystem compared to mainstream languages like Java or Python. Less support for concurrent and parallel programming compared to languages like Java or Rust. GC overhead may affect real-time performance in latency-sensitive applications.

6.5 Python

It is an *imperative* language with, *dynamically typed, lexical scoping, dynamic name checking, automatic MM (GC)*

Specialties: simplicity and readability, AI applications, follows the ”duck typing” philosophy (behavior $>$ type), whitespace-based syntax.

Pros: simple and readable syntax, making it easy to learn and use. Extensive STL and versatile.

Cons: slow (due to the Global Interpreter Lock (GIL), only one thread can execute Python bytecode at a time, limiting the effectiveness of multithreading for CPU-bound tasks), dynamic typing can lead to runtime errors and make code harder to refactor and maintain.

Chapter 7

Practice Problems

7.1 23 Winter

Problem #1: C has casts e.g. ”(char *) 0”. Why shouldn’t OCaml and Java has casts like this?

We note that one of the *benefits* of both Java and Ocaml is that they are strongly typed languages. This means that at compile time we can catch type errors to ensure that types correct. Having casting like this would remove all of the positive effects of being able to ensure we don’t have type errors before run time.

Problem #2: ”Colorless green ideas sleep furiously.” is a string that illustrates a particular point about English. Does a similar string illustrate the same point in Ocaml? If so, give such a string a briefly explain why.

This English sentence represents the issue of something being *syntactically correct, but semantically* having no meaning. In Ocaml we provide two such examples.

```
let f = fun x -> y + 3
let g = (Some, 3)
```

Here the function definition doesn’t make sense since y may not be defined, but it is syntactically correct. The second example returns a tuple of an option and a value, this is correct but redundant and doesn’t make sense, since if we are going to check the first part of a tuple to be not None, we should just return Some.

Problem #3a: Define an ocaml function *comcmp* such that if frag1 is the complement of frag2 then it returns COMP, if frag1 is shorter than frag2 but is complementary to an initial prefix of frag2 returns LEFT SHORT. If frag2 is shorter than frag1 but is cimplementary to an initial prefix of frag1 returns RIGHT SHORT. Otherwise it returns NOT COMP

```
let getcomp = function
    | A -> T
```

```
    | C -> G
    | G -> C
    | T -> C

let rec compcmp frag1 frag2 = match (frag1, frag2) with
    | [],[] -> COMP
    | [], _ -> LEFT_SHORT
    | _ , [] -> RIGHT_SHORT
    | f1h::f1t, f2h::f2t -> if getcomp f1h = f2h then
                                compcmp f1t f2t
                            else
                                NOT_COMP
```

Problem #3b: Modify the Ocaml function to work on lists of any types

```
let rec gcompcmp cmp frag1 frag2 = match (frag1, frag2) with
    | [],[] -> COMP
    | [], _ -> LEFT_SHORT
    | _ , [] -> RIGHT_SHORT
    | f1h::f1t, f2h::f2t -> if cmp (f1h f2h) then
                                gcompcmp f1t f2t
                            else
                                NOT_COMP
```

To make this happen we pass in a custom comparison operation that will determine if two elements of the list are complementary.

Problem #3c: What are the types of compcmp and gcompcmp?

compcmp: fragment → fragmentcomptest
gcompcmp: ('a → 'b → bool) → a' list → b' list → comptest

Problem #4: Consider the following Ocaml code:

```
type nucleotide = A | C | G | T
type fragment = nucleotide list
type acceptor = fragment -> fragment option
type matcher = fragment -> acceptor -> fragment option

let rec match_star matcher frag accept = match (accept frag) with
    | None ->
        matcher frag
            ( fun frag1 ->
                if frag == frag1 then None
                else match_start matcher frag1 accept)
    | ok -> ok
```

Problem #4a: This code uses 'None' but not 'Some'. How can it get away with that?

This is because the patter matching is exhaustive in this case still, we observe that ok will get matched to (Some x) since we have already handled the None case.

Problem #4b: In the above code we use '==' (i.e. address comparison) instead of '=' (structural equality). What are the advantages (if any) and disadvantages of using '==' instead of '=' in this program.

One benefit of using physical equality is that it will be faster (only address comparison) compared to doing structural equality. It is okay to do it in this case because we are comparing frag1 to itself and thus the only time where both address comparison and structural equality result in the same thing is if they are the same variable.

Problem #4c: Suppose we defined a matcher to now taken in an acceptor and then return another acceptor, why is this okay?

The matcher essentially takes as arguments an acceptor and a fragment and gives back a fragment option. This can be thought of as taking an acceptor and giving back a function that takes s fragment and gives a fragment option (which is an acceptor). We would have to change the ordering of matcher to be type matcher = acceptor $\rightarrow$ fragment $\rightarrow$ fragment option.

Problem #4d: Rewrite the function match star in terms of this new matcher type

```
let rec match_star matcher frag accept = match (accept frag) with
| None ->
    matcher
        ( fun frag1 ->
            if frag == frag1 then None
            else match_start matcher accept frag) frag
| ok -> ok
```

Problem #5: Suppose we provide a new definition of float in Ocaml below:

```
type sign  = | Minus | Plus
type exp = | Normal of int | Tiny | Special
type frac = int
type flooat = sign * exp * frac
```

Suppose we define all the addition, multiplication and division functions for these new type of flooat. Compare and contrast for programs could use 'float' (IEEE standard) versus 'flooat'. Give pros and cons of how well two types work in practice.

We first start with the pros of flooat, this type has more transperency on how we implement a flooat value and thus can benefit the developer. This is because we have split the parts of the flooat into three different values it makes it easier to access these individually. Furthermore we utilize two integers so we are not as constrained by the number of bits so we can have a more express more float values.

Cons of flooat: we consume more space than a standard float. We will have more undefined behaviour at the edge cases since we have restrictions on the size of an int. Furthermore flooat is not a primitive type anymore and this will enforce more restrictions in terms of the functionality.

Problem #6: Explain how in Java, three thread objects could exists even though there are no threads there (i.e as far as the OS is concerned)

The thread objects could still exist if we have three threads that are in the terminated state since we can still visit the ”thread graveyard” even though they are not actively running.

Problem #7a: In class it was said that we can take an OCaml function definition let f a b c d = WHATEVER d and simplify it by writing let f a b c = WHATEVEVER. When this works, does f’s type change?

No, f’s type does not change due to currying f would still be treated as a nested function, so it would not change.

Problem #7b: Does this idea work even if ’f’ has only one argument?

Yes, it would still work since we can still curry a singular argument take the following example:

```
let f x = g x
let f = g
```

This above example would still work.

Problem #7c: Give an example where this idea does not work even though whatever is parenthesized expression This idea would not work if we used the recursive keyword and defined f in terms of itself.