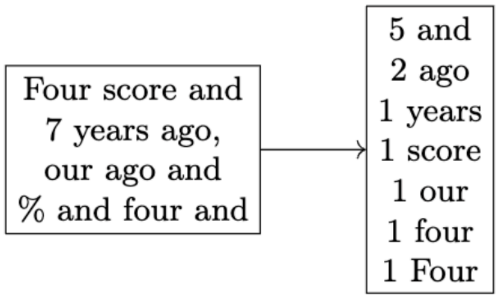
Lecture 1

Sub-Note 1: Brief Intro

Problem: Write a program that takes input from a finite, arbitrary byte stream and outputs a ***sorted concordance count*** (e.g. frequency count) from largest to smallest of all the words. We define a word using the regex: **[a-zA-Z]+**. Additionally, words are bounded by whitespace.

Example

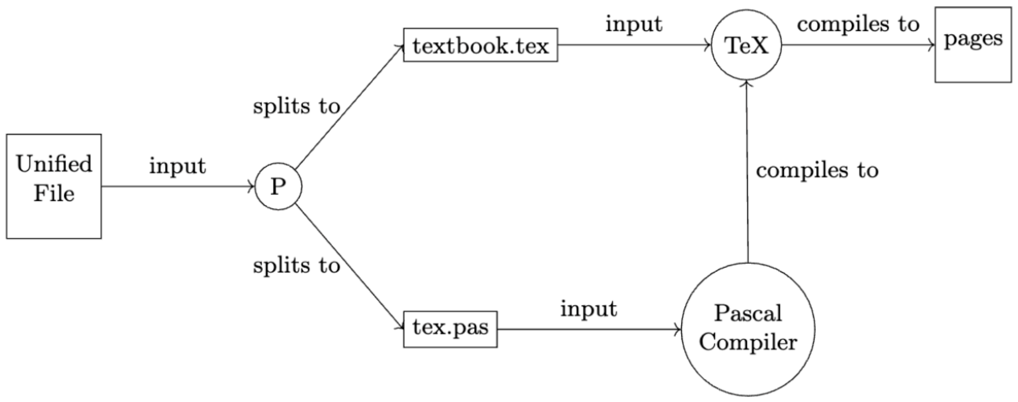


Solution 1 (Knuth): Knuth developed a program based on the concept of literate programming to solve the problem above, using ***hash tries*** (trie with hash table at each node)

Wrote Tex since the lead type press at the time couldn't handle code and equations nicely

Developed "***literate programming***": Rather than having 2 distinct worlds A and B (e.g. source code and documentation), combine them into a single system AB where you can see both the source code and documentation

- Created this because he wanted to ***keep documentation in sync with code***
- New approach followed the diagram below



Solution 2 (McIlroy): Wrote a sh program using the tr command

```
tr -cs 'A-Za-z' [\n]* | sort | uniq -c | sort -rn
```

- squeezes new lines (-s option) and applies them to non-alphabetic chars (-c option)
- Feeds this as input to the sort command using a pipe (invented by McIllroy)
- Output how many times the word was repeated (-c option in uniq)
- Sort in reverse numerical order

Comparison: If you want to solve the example problem with a single processor thousands of times, use Knuth's approach (faster, more "checked" by compilers) over McIllroy's
- Although McIllroy's approach is technically "parallel" (e.g. uniq and sort can run in parallel), the sort command must wait for ALL input to be read
- McIllroy's approach is ***more concise, clear***, and better for ***ad-hoc situations***

Sub-Note 2: Language Design Issues (e.g. Language Wars)

Often disagreement about which languages/tools should be used for a specific project
- e.g. Microsoft Azure vs GCP
- Often the wars are between "neighbors" (e.g. languages with similar behavior like C vs C++, OCaml vs Haskell)

Guiding Question: How do we resolve disagreements?

1. Inertia
- What do people already know?
    - e.g. If someone knows Python, they gravitate towards that

2. Efficiency (Runtime)
- Power (watts), energy (joules), time (s), space (bytes/bits)
    - With respect to the application's POV (keep these numbers ***small***)
- Utilization
    - With respect to the device's POV (keep this number ***large***)
- Time is split into real time and CPU time (and an algorithm could be better in one aspect and worse in the other)
- Network access (e.g. if your application can execute without talking to a server, have an advantage over your competitors)

3. Ecosystem Size
- Available libraries, tools

4. Syntax Issues
- Simplicity vs readability vs compatibility/interoperability vs documentation/learnability
- Ex. (2 + 3) * 4 (C) vs 2 3 + 4 * (Forth)
    - Although the second option has ***simpler*** syntax, we prefer the first since it's ***easier to read***
- Ex. C has a library called Cosmopolitan which allows executables to run unchanged on most platforms ⇒ ***super compatible***
    - Every executable contains several copies of your program, each of which is designed for a different OS/platform

Design Issue #1: Orthogonality

- Different components of a system should be independent
    - i.e. when writing a program, using a feature from one part of a language ***ideally shouldn't*** affect any other choices you want to make
- Ex. Imagine the following ***templated function*** in C:

```
t f() {
    t x = …
    return t;
}
```

- Want to be able to choose any type for t, but that's not allowed in C ⇒ can't return array types (e.g. typedef int t[100])
    - Not feasible for a function to return an array of size 10 GBs every time it is called if the array is super large
    - Instead, C makes you pass in pointers
    - Example of non-orthogonality – ***C traded orthogonality for efficiency***

Design Issue #2: Safety

- How safe is the language you're using in the event that bugs arise in your code?
    - Ex. Rust vs C++ ⇒ Rust enforces memory/type safety at compile-time unlike C++
    - Ex. Take the code

```
char *p = nullptr; return *p;
```

        - Example of undefined behavior in C (another example is you can't shift by a negative number)
            - On some platforms, returns data at location 0
            - On SEASNet, throws a seg fault as the hardware uses virtual memory + the low-order virtual address is a page marked invalid
- We prefer languages that ***limit undefined behavior***

Design Issue #3: Abstraction

- Ideally want a language to be able to build an abstraction for lower-level data
- Examples of abstraction include functions, classes, and packages for modules

---

Lecture 2

Sub-Note 1: Functional Programming

1. Why?

Put simply, there is a need to come up with a way to write computer software that is better than what we have currently (e.g. current languages work BUT are a real pain – they're slow, unreliable, and we generally want something cleaner)

1. ***Clarity – Want programs to be easier to write and maintain***
   - FP movement was started by John Backus, who created Fortran (ancestor of most current languages like C and Java)
     - In these languages, you ***write everything as a recipe for a computer to follow***; programs are a series of commands to be executed
       - Even includes loops/functions (shorthand for a series of cmds)
     - **PROBLEM**: Every solution must then be a long, thin, string of commands
     - An example of a clarity issue:

```
int b  = g(z);
int a = f(b) + 1;
…
int z = f(b) + 1;
```

In languages like C/Fortran, ***a is NOT necessarily equal to z*** – could be possible that the value of b has changed between when we first called it and when we called it after

   - Above example shows ***most languages lack referential transparency*** – idea that it should be obvious what every identifier (i.e. name) refers to
     - In other words, ***if you write the same expression 2x, you should get the same value*** regardless
     - ***Makes programs easy to read***

2. ***Performance*** – Want our programs to be ***more easily optimizable***
   - In other words, we ***want the compiler to do the heavy lifting*** to make our programs run faster
   - Suppose we have the following C program:

```
int x, y, z;                    movl $12, x
int f()                         movl $12, %rsi
{                               call f
        x = 12;                 movl %rax, y
        y = f(x);               movl %z, %rax
        return x + z;           addlq $12, %rax
}                               ret
```

Ideally, we want the machine code to look something like the right – BUT, a ***C compiler can't generate it***. Specifically, it is possible that f has changed x. More specifically, ***after calling f(), it has to reload x since it doesn't know if f() has changed the value of x***. This results in extra machine-level code which could slow down performance.

- To make this more clear, consider the following example:

```
return f(x) + f(x);                        movl x, %ax
                                           call f
                                           addl %rax, %rax
                                           ret
```

Again, because f() could have a side-effect and could yield a different value for each argument, we can't do the above addl instruction

Solution: ***Functional programming*** attacks the clarity problem by ***giving you referential transparency*** (names always stand for the same thing) and ***better performance*** (compiler can generate better code for reasonably common things like the above).

Sub-Note 2: Comparing Imperative/Functional/Logic

Imperative languages still rule in most application areas, but the ***lessons we learn from functional programming extend across languages***. Languages are not purely imperative, or purely functional – for instance, C++ has functions which are a major part of functional programming. You could limit yourself to a functional style in C++, but it's so awkward that people rarely do it, so it's better to learn the functional way of thinking using a FP language.

Case in Point: Google used a functional programming idea called MAP-REDUCE in its early days to speed up its query processing internally
- Essentially split a big problem into smaller, more generic problems

1. **Imperative**
    - **Basic Unit**: Command/Statement ⇒ S1, S2, …
    - **Glue**: Semicolons are used to sequence statements/pieces of your program
    - **Relies on**: Variables with state, which are modified via assignment

2. **Functional**
    - **Basic Unit**: Functions ⇒ F1, F2, …
    - **Glue**: F1 (F2(x), F3(y, w), w) ⇒ function calls
        - Function calls have ****some* sequencing*** – you have to call F2 and wait for it to return before calling F1
        - However, ***F2 and F3 can still run in parallel***
    - **Relies on**:
        - Functional evaluation providing a partial order on computation
            - Know F2 has to finish before F1 starts but don't know that it needs to finish before F3 starts, so partial order
        - ***Referential transparency*** (w in F1 and F3 is the same w)
        - ***Functional forms*** (e.g. higher-order functions) – functions that take other functions as arguments or return other functions as results

- **Give up on**:
    - ***Assignment statements*** (no variables with state – variables have values but they don't change)
    - ***Side-effects*** (when a statement affects the machine state, e.g. by assignment, I/O, …)

Functional Form Example(s)

- We can treat the sigma operator $\sum(0 \leq i < 10)\, f(i)$ as a function with 3 arguments: $\sum(0, 9, f)$
- Additionally, C has a function called qsort that takes a comparison function as the last argument; this comparison function is called on each element in the array

3. **Logic (WIP)**

Sub-Note 3: Functional Programming Clarifications

Question: Are there cases where F2 and F3 can't be computed in parallel?

- In a ***FP language***, generally ***NO*** (at least if your hardware isn't limited)
- In C/C++/Fortran, generally ***YES*** – there are ***often constraints on order of evaluation***
    - In Java, you have to evaluate all code from left to right (F2 must finish before F3)
    - In C, you have to finish one before starting the other
    - The answer is NO if we know ***F2 and F3 are completely independent***

Question: Imperative vs functional trade-offs with ***out-of-order processors***

- Imperative languages often benefit from out-of-order execution since they express computation in a straightforward, linear manner which allows processors to more easily exploit ILP and analyze dependencies
- FP languages emphasize immutability and statelessness. These features ***may lead to more intermediate data and memory allocations***, but modern runtime systems and compilers mitigate this through lazy evaluation and JIT compilation
- One advantage is that FP languages offer a simplified synchronization model in parallel programming (don't need to worry about synchronization as much) – since data is immutable, there's ***no risk of race conditions caused by shared mutable state***
    - However, the thread executing F2 must finish before a thread executes F1

Question: Why OCaml over SML?

- Mostly a "cultural" thing – OCaml is more widely used in industry, which may mean it has more practical hacks than SML does
- SML deals with floating point better since need to explicitly convert in OCaml (3.0 + 4 works in SML)

Question: Some HW problems show functions taking an array and appending to it; why isn't this considered a side-effect?

- Recall OCaml is primarily immutable – the array being appended to is completely separate; the original array isn't being modified, so there's no side-effect

Question: If there are no side effects, doesn't the machine state (e.g. registers) change as functions are evaluated?

- Technically yes, but these changes aren't directly visible (e.g. users can't see the content of registers and there is no way to look in memory). There is a difference between the state of the underlying machine and state visible to the user(s).

Sub-Note 4: Introduction to OCaml

Basic OCaml Properties

1. **Compile-time (static) type checking** – ***types are checked before the program starts executing***
    - Similar to Java, C/C++; unlike Python, sh
    - We want this because we ***want reliability*** – we don't want a program to fail because of type errors (impossible in a running program)
    - This is why OCaml is more of a production language than a scripting one
2. **Don't need to always worry/write down types**
    - Similar to JS, Python; unlike traditional C/C++
    - C/C++ actually took a page from OCaml's book and introduced the ***auto*** keyword
3. **Less of a need to worry about storage management**
    - No concept of "free" or "del" operations to delete objects; instead the ***system automatically does it for you with a garbage collector***
        - Garbage collector just frees objects when they aren't needed
    - Similar to JS, Python, Java; unlike C/C++
    - We want this because we ***want reliability and convenience***, and we are willing to trade performance for this
4. **Good support for higher-order functions**

OCaml Syntax

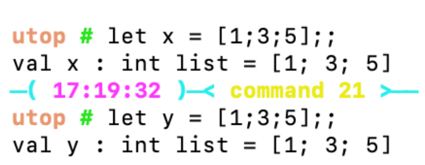
Let's say we have the expression:

"a" = if n > 0 then "a" else 29;

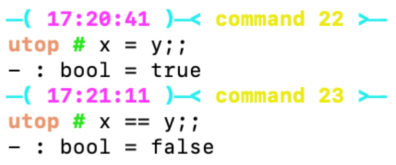
Intuitively, this should evaluate to true, but it ***throws an error in OCaml*** because the then and else parts ***MUST be the same type***. Recall OCaml does ***type checking during compile-time***, so if it fails, it won't execute your code even if it is reasonable (if we executed the above in Python, it would evaluate to true).

- Type-checking is a double-edged sword – good for catching stupid type-related mistakes

NOTE: ***= compares the values*** while ***== compares whether they are the same object***



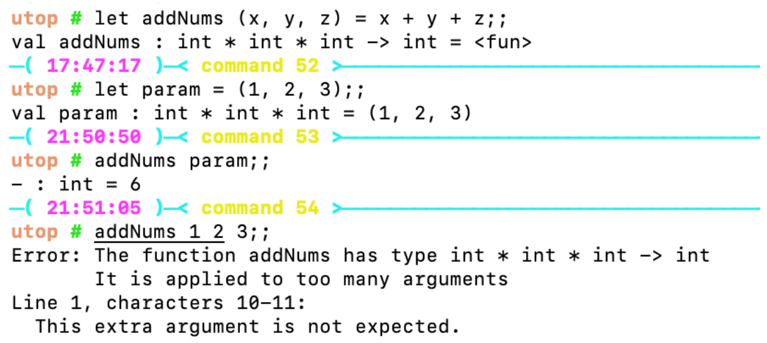
```
utop # let x = [1;3;5];;
val x : int list = [1; 3; 5]
─( 17:19:32 )─< command 21 >──
utop # let y = [1;3;5];;
val y : int list = [1; 3; 5]
```



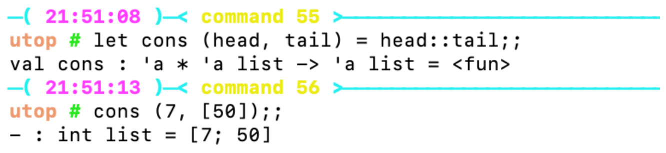
```
─( 17:20:41 )─< command 22 >─
utop # x = y;;
- : bool = true
─( 17:21:11 )─< command 23 >─
utop # x == y;;
- : bool = false
```

Aside on Functions:

- Every function in OCaml ***takes a single argument*** and ***returns a single result***
    - Technically an exception but not really – :: is a built-in function, can think of it as taking 2 arguments – an element and a list
- Can pass multiple arguments to a function, but it's written differently

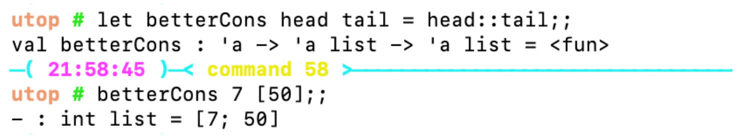


```
utop # let addNums (x, y, z) = x + y + z;;
val addNums : int * int * int -> int = <fun>
─( 17:47:17 )─< command 52 >──────────────────────────────
utop # let param = (1, 2, 3);;
val param : int * int * int = (1, 2, 3)
─( 21:50:50 )─< command 53 >──────────────────────────────
utop # addNums param;;
- : int = 6
─( 21:51:05 )─< command 54 >──────────────────────────────
utop # addNums 1 2 3;;
Error: The function addNums has type int * int * int -> int
       It is applied to too many arguments
Line 1, characters 10-11:
  This extra argument is not expected.
```



```
─( 21:51:08 )─< command 55 >──────────────────────
utop # let cons (head, tail) = head::tail;;
val cons : 'a * 'a list -> 'a list = <fun>
─( 21:51:13 )─< command 56 >──────────────────────
utop # cons (7, [50]);;
- : int list = [7; 50]
```

- In the above examples, we ***pass the arguments as a tuple*** and it returns a list
- This is ***NOT ideal*** – the function needs both arguments before it can execute, which is sort of an unnecessary restriction
- The better way to define cons is shown below:



```
utop # let betterCons head tail = head::tail;;
val betterCons : 'a -> 'a list -> 'a list = <fun>
─( 21:58:45 )─< command 58 >──────────────────────────
utop # betterCons 7 [50];;
- : int list = [7; 50]
```

- Still ***achieve the same result***
- betterCons has ***implied parentheses around head***, ***pass in a value of type ' a*** and it returns a function from list to list
- ***betterCons is more flexible*** – don't have to call the intermediate resulting function immediately, but it is more confusing
- "***currying***": idea that you have one argument and one result (functions with more arguments are simulated with higher-order functions)
- betterCons is shorthand for ***betterCons2 = fun head -> (fun tail -> head::tail)***

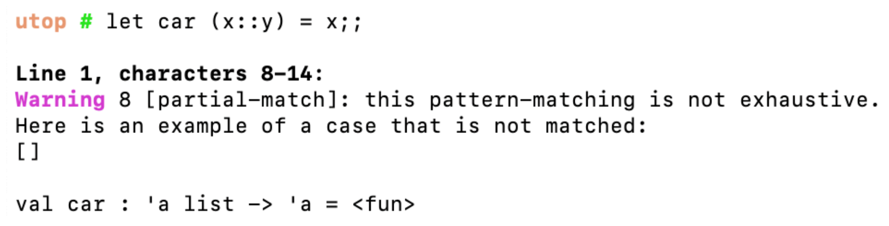
Aside on Currying:

- Currying is useful because it ***makes partial application natural***
    - Ex. Take the following curried function $let\ add\ x\ y = x + y$

        - We can “fix” some arguments then later apply the rest; e.g. create an incrementer function $let\ increment = add\ 1$
        - Through this, we retain access to both add and $increment$
- The ***OCaml compiler is also optimized for currying*** instead of passing in a tuple
- However, for developers unfamiliar with currying, its syntax/behavior can be ***confusing***

Function calls are ***left-associative***, meaning implied parentheses/arrows go from ***left to right***
- f g h is equivalent to writing (f g) h
- Differs from function types, which have ***right-associative arrows***
- ‘a -> ‘a list -> ‘a list is equivalent to writing ‘a -> (‘a list -> ‘a list)

Let’s say we want to grab the first element of a list:



```
utop # let car (x::y) = x;;

Line 1, characters 8-14:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]

val car : 'a list -> 'a = <fun>
```

- Returns a warning since there are some lists that the pattern (x::y) won’t match ⇒ e.g. the empty list
- If we ignore the warning and ***call car [];;*** we get a ***runtime exception***

Lecture 3

Sub-Note 1: Pattern Matching

Question: Relative to languages like C/C++, how much overhead (in memory and time) does a recursive function call in a functional language like OCaml have?
- Generally speaking, not much more
    - ***Closures*** (e.g. functions that return other functions) are an exception; they can introduce additional overhead because they may need to allocate memory on the heap to store captured variables
- The ***real costs*** come from OCaml’s ***garbage collection*** system, not the function calls
    - In C/C++, programmers ***manually*** manage memory
    - In OCaml, memory management is ***automated*** via a garbage collector, which adds runtime overhead
    - OCaml ***trades performance for safety and convenience***

Patterns
- Used to take apart data structures; similar to grep from Linux

```
match E with
  | <Pattern 1> -> <Expression 1> (* first vertical bar is optional)
  | <Pattern 2> -> <Expression 2>
  | ...
```

| *<Pattern n> -> <Expression n>*

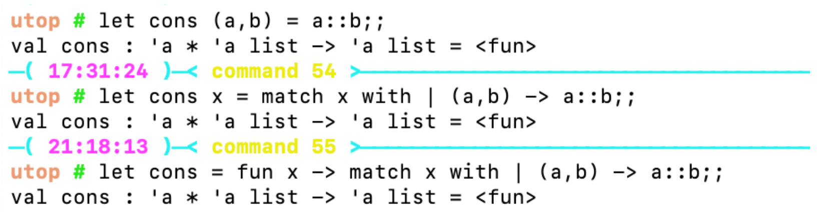
In the above, the expression E is evaluated to produce a value. Patterns are checked in order, from 1 to n, to see if they match that value. For the first one that does, its corresponding expression is evaluated and becomes the result of the match expression.

- If ***no pattern*** matches, the program raises a ***runtime error***

Basic Patterns	**Values Matches**
1. Constants (0, “a”)	Itself
2. Identifier (var1)	Any value, but it binds the identifier to that value
3. _	Any value, but it discards it
4. (P)	Whatever the pattern P matches
5. P1,P2,...Pn	A n-tuple st. the first component matches P1, …
6. [P1;P2;...Pn]	A list of length n st. each element matches the corresponding pattern
7. P1::P2	A list of length > 0 st. P1 matches the head and P2 matches the tail

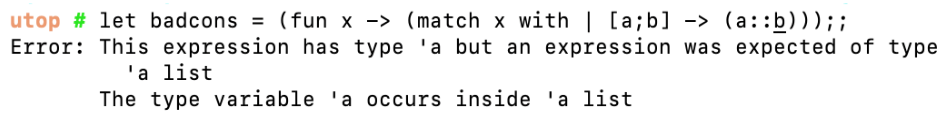
Between 6 and 7, most prefer the latter

- ***[P1;P2;P3] is equivalent to P1::P2::P3::[]*** since ***:: is right-associative***
    - If P1 matches int, so must P2 and P3
- Essentially P1::(P2::(P3::[])))
- Likewise, ***[P1] is equivalent to P1::[]***



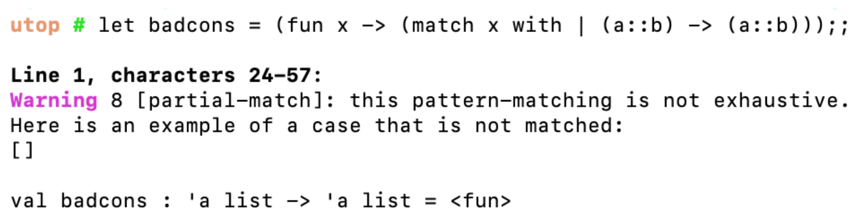
```
utop # let cons (a,b) = a::b;;
val cons : 'a * 'a list -> 'a list = <fun>
─( 17:31:24 )─< command 54 >──────────────────────────────
utop # let cons x = match x with | (a,b) -> a::b;;
val cons : 'a * 'a list -> 'a list = <fun>
─( 21:18:13 )─< command 55 >──────────────────────────────
utop # let cons = fun x -> match x with | (a,b) -> a::b;;
val cons : 'a * 'a list -> 'a list = <fun>
```

- The above functions are equivalent, so they ***should generate the same machine code***
- Don’t need a fall-through case since the pattern (a,b) can match any 2-tuple as long as b is a list and a is a value of the same type



```
utop # let badcons = (fun x -> (match x with | [a;b] -> (a::b)));;
Error: This expression has type 'a but an expression was expected of type
         'a list
       The type variable 'a occurs inside 'a list
```

- What is wrong with the above declaration? ***Lists are homogeneous***, so a and b must be the same type, but the :: means b must be a list of type a



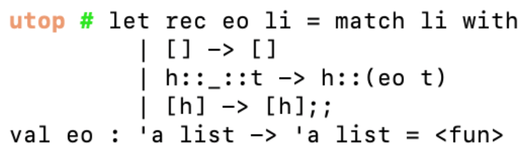
```
utop # let badcons = (fun x -> (match x with | (a::b) -> (a::b)));;

Line 1, characters 24-57:
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]

val badcons : 'a list -> 'a list = <fun>
```

- The above also isn't correct because even though it works on non-empty lists (e.g. badcons(23::[15::19])), it doesn't work on the empty list []
    - This ***seems wrong at first glance***, but because of the parentheses, 23::[15::19] is evaluated first and becomes a list
- In general, match expressions are often more simple for tuples than lists

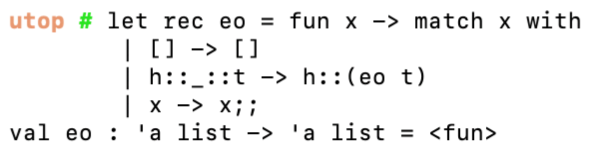
Sub-Note 2: Recursion

Example 1: Given an input list, write a function to return a list containing every other element

- If the input is [1;2;3], return [1;3]
- If the input is [1;2;3;4], return [1;3]



```
utop # let rec eo li = match li with
         | [] -> []
         | h::_::t -> h::(eo t)
         | [h] -> [h];;
val eo : 'a list -> 'a list = <fun>
```



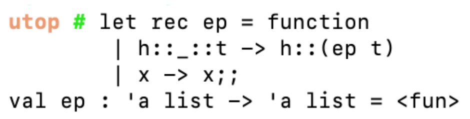
```
utop # let rec eo = fun x -> match x with
              | [] -> []
              | h::_::t -> h::(eo t)
              | x -> x;;
val eo : 'a list -> 'a list = <fun>
```

- ***h::v::t*** matches a ***list of length 2 or more*** where the first element is h, the second element is v, and the tail is t
    - Since we don't care about the second element, we can use the _ character
- Define a ***recursive*** function by using the ***rec keyword***

Comments

- The general rule in OCaml is you ***can't define an identifier using itself*** (e.g. for the syntax let ID = EXPR, we can't reuse ID in EXPR) ***unless you use the rec keyword***
    - n = n + 1 or even n = n + 0 (even though we don't modify n) is ***forbidden***
- In the example on the right, we don't need the first pattern – the last pattern (e.g. x -> x) handles it for us already

We could similarly define a function ep with the same purpose:



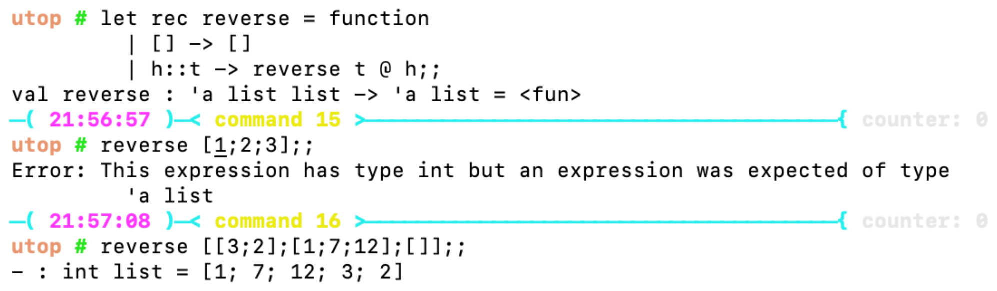
```
utop # let rec ep = function
         | h::_::t -> h::(ep t)
         | x -> x;;
val ep : 'a list -> 'a list = <fun>
```

- ***function is short for fun x -> match x***
- x -> x is a fall-through case

Example 2: Given an input list, write a function to return the list but reversed

- If the input is [1;2;3], return [3;2;1]
- If the input is [[1;3];[2;5;8];[]], return [[];[2;5;8];[1;3]]

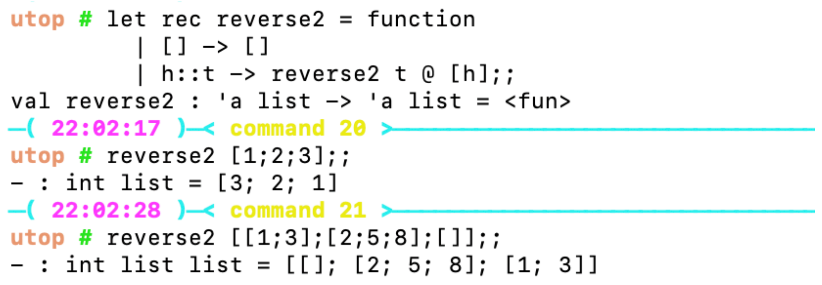
Incorrect Solution



```
utop # let rec reverse = function
         | [] -> []
         | h::t -> reverse t @ h;;
val reverse : 'a list list -> 'a list = <fun>
─( 21:56:57 )─< command 15 >──────────────────────────────────────────{ counter: 0
utop # reverse [1;2;3];;
Error: This expression has type int but an expression was expected of type
         'a list
─( 21:57:08 )─< command 16 >──────────────────────────────────────────{ counter: 0
utop # reverse [[3;2];[1;7;12];[]];;
- : int list = [1; 7; 12; 3; 2]
```

- @ is used to concatenate 2 lists – it takes 2 lists as arguments, so ***h must be a list***. Therefore, (reverse t) must return a list of lists, as ***this makes h a list*** when extracted as the head
- This is ***WRONG***, and we ***know immediately from the function type signature***
- **IMP**: Type checker allows us to ***discover bugs in programs before we run them***
    - ***OCaml > Python in this aspect*** – in Python, you would need to actually run the program to find the bugs

Correct (But Inefficient) Solution



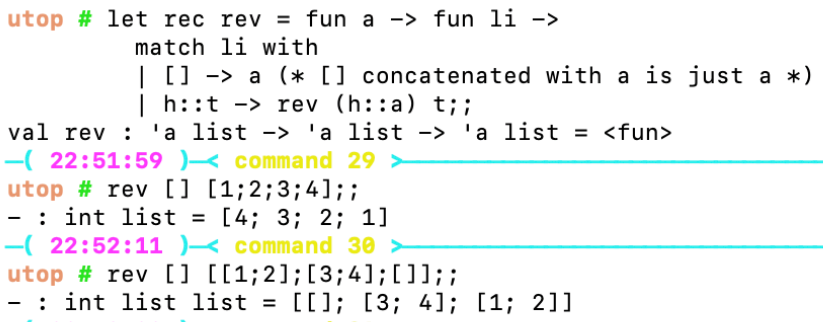
```
utop # let rec reverse2 = function
         | [] -> []
         | h::t -> reverse2 t @ [h];;
val reverse2 : 'a list -> 'a list = <fun>
─( 22:02:17 )─< command 20 >──────────────────────────────
utop # reverse2 [1;2;3];;
- : int list = [3; 2; 1]
─( 22:02:28 )─< command 21 >──────────────────────────────
utop # reverse2 [[1;3];[2;5;8];[]];;
- : int list list = [[]; [2; 5; 8]; [1; 3]]
```

Question: Is there a way to efficiently add an element to the end of a list?

- No – in OCaml, ***lists are singly linked lists***; accessing the head is fast since it's the first element, but ***accessing the last element requires traversing the entire list***
    - To add to the back, we necessarily need to find the last element
- OCaml encourages efficient programming by ***disincentivizing the above***

- Because of this, the above function works, but is ***slow***
    - Suppose t contained a million elements – the call to reverse2 would need to recursively process all elements of t and then concatenate h to it
    - Concatenation is expensive since it needs to copy all elements of the first list (e.g. reverse2 t) to create a new list
- In other words, the ***cost of A@B is O(length of A)***
- However, the ***cost of A::B is O(1)*** – only need to create one new node in memory
- The cost of reverse2 A is therefore ***O(length of A)^2 – can we do better?***

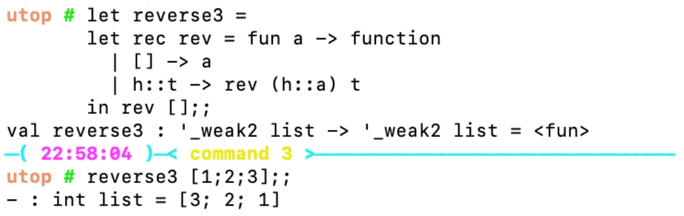
Optimal Solution



```
utop # let rec rev = fun a -> fun li ->
         match li with
         | [] -> a (* [] concatenated with a is just a *)
         | h::t -> rev (h::a) t;;
val rev : 'a list -> 'a list -> 'a list = <fun>
─( 22:51:59 )─< command 29 >──────────────────────────
utop # rev [] [1;2;3;4];;
- : int list = [4; 3; 2; 1]
─( 22:52:11 )─< command 30 >──────────────────────────
utop # rev [] [[1;2];[3;4];[]];;
- : int list list = [[]; [3; 4]; [1; 2]]
```

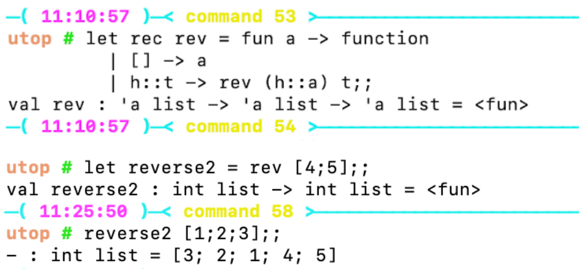
- Computes the reverse of li by using an ***accumulator a*** (***separate list*** that collects the reversed elements as we traverse through the input list)
- ***h::t -> rev (h::a) t*** ⇒ prepend the current element h to a, then recursively process t (***a is built in the opposite order we're traversing through***, so from right -> left)

More clean example(s) of the above:



```
utop # let reverse3 =
       let rec rev = fun a -> function
         | [] -> a
         | h::t -> rev (h::a) t
       in rev [];;
val reverse3 : '_weak2 list -> '_weak2 list = <fun>
─( 22:58:04 )─< command 3 >──────────────────────
utop # reverse3 [1;2;3];;
- : int list = [3; 2; 1]
```

- rev is defined as a ***local function*** using the ***in keyword***, meaning the ***scope of rev is restricted to reverse3***
    - ***Calling rev*** on the next statement will produce an ***unbound value error***
- reverse3 is pre-configured to ***always use the empty list as the initial accumulator***
- ***Better for encapsulation*** because it makes the code more self-contained



```
─( 11:10:57 )─< command 53 >──────────────────
utop # let rec rev = fun a -> function
         | [] -> a
         | h::t -> rev (h::a) t;;
val rev : 'a list -> 'a list -> 'a list = <fun>
─( 11:10:57 )─< command 54 >──────────────────

utop # let reverse2 = rev [4;5];;
val reverse2 : int list -> int list = <fun>
─( 11:25:50 )─< command 58 >──────────────────
utop # reverse2 [1;2;3];;
- : int list = [3; 2; 1; 4; 5]
```

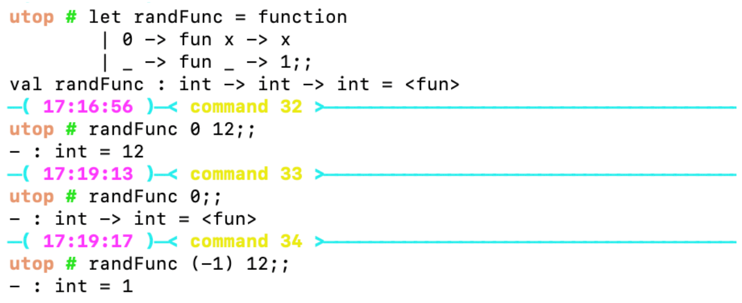
- Here, rev is a ***top-level function*** – globally-defined, accessible anywhere
- rev is a ***curried function*** – takes an accumulator list ('a list) and returns a function from a list to another list ('a list -> 'a list)
- In this implementation, ***we can call rev with any initial accumulator we want***

Aside on Functions

1. **Functions can be anonymous**
    - fun x -> x + 1 is a function that gives its argument a name, but the ***function itself doesn't have a name***
    - Similar to how C++ doesn't assign the intermediary value (n + 3) a name in (n + 3) * 12

    - OCaml applies this attitude to functions
- Using nameless functions allows you to ***write more compact code***

2. **Functions often return other functions**



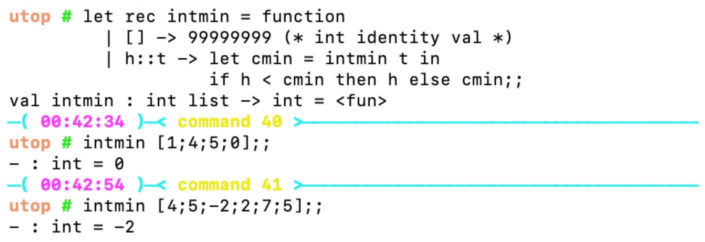
```
utop # let randFunc = function
        | 0 -> fun x -> x
        | _ -> fun _ -> 1;;
val randFunc : int -> int -> int = <fun>
─( 17:16:56 )─< command 32 >──────────────────────────────────────
utop # randFunc 0 12;;
- : int = 12
─( 17:19:13 )─< command 33 >──────────────────────────────────────
utop # randFunc 0;;
- : int -> int = <fun>
─( 17:19:17 )─< command 34 >──────────────────────────────────────
utop # randFunc (-1) 12;;
- : int = 1
```

- randFunc returns a function – if we pass in 0, we get the identity function; otherwise, we get a function that always returns 1

3. **fun vs function**
- ***fun accepts arguments explicitly*** and processes them in the function body
    - Functions defined with fun are ***automatically curried***
    - Ex. fun x y z -> E resolves to fun x -> fun y -> fun z -> E
- ***function is designed for pattern-matching*** and only takes one implicit argument (can't handle multiple arguments)

Example 3: Given an input list, write a function to return the minimum element
- If the input is [5;1;2], return 1
- If the input is [], return the identity value (for int, this is 9999999)
- Then rewrite the function to be generic

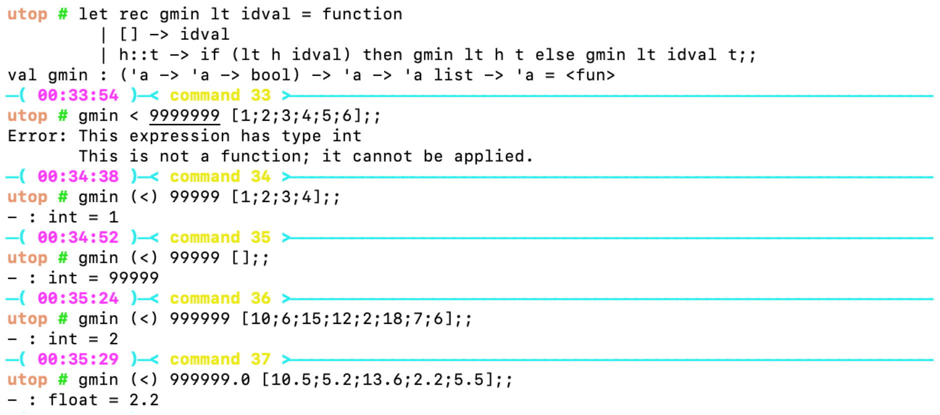
Naive Solution (Int)



```
utop # let rec intmin = function
         | [] -> 99999999 (* int identity val *)
         | h::t -> let cmin = intmin t in
                   if h < cmin then h else cmin;;
val intmin : int list -> int = <fun>
─( 00:42:34 )─< command 40 >──────────────────────────────────
utop # intmin [1;4;5;0];;
- : int = 0
─( 00:42:54 )─< command 41 >──────────────────────────────────
utop # intmin [4;5;-2;2;7;5];;
- : int = -2
```

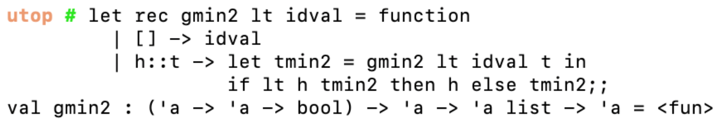
- Problem: Throws a ***type error*** if we pass in lists of floats, strings, ...

Optimal Solution (Generic)



```
utop # let rec gmin lt idval = function
         | [] -> idval
         | h::t -> if (lt h idval) then gmin lt h t else gmin lt idval t;;
val gmin : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a = <fun>
─( 00:33:54 )─< command 33 >──────────────────────────────────────────────────
utop # gmin < 9999999 [1;2;3;4;5;6];;
Error: This expression has type int
       This is not a function; it cannot be applied.
─( 00:34:38 )─< command 34 >──────────────────────────────────────────────────
utop # gmin (<) 99999 [1;2;3;4];;
- : int = 1
─( 00:34:52 )─< command 35 >──────────────────────────────────────────────────
utop # gmin (<) 99999 [];;
- : int = 99999
─( 00:35:24 )─< command 36 >──────────────────────────────────────────────────
utop # gmin (<) 999999 [10;6;15;12;2;18;7;6];;
- : int = 2
─( 00:35:29 )─< command 37 >──────────────────────────────────────────────────
utop # gmin (<) 999999.0 [10.5;5.2;13.6;2.2;5.5];;
- : float = 2.2
```

- ***gmin takes a comparison function*** (e.g. < operator) and returns a function with type signature ‘a -> ‘a list -> ‘a
    - Passing an identity value to this function returns a new function with type ‘a list -> ‘a, which, when called with a list, returns the minimum element
- To use < as a function rather than operator, ***wrap it in parentheses*** since operators are treated as infix by default
- The ***comparison function lt is binary*** (as seen in the type signature), so we ***call it as (lt h idval) and not (h lt idval) to pass arguments in the correct order***
- gmin also works for 2D lists as long as idval is a list



```
utop # let rec gmin2 lt idval = function
         | [] -> idval
         | h::t -> let tmin2 = gmin2 lt idval t in
                   if lt h tmin2 then h else tmin2;;
val gmin2 : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a = <fun>
```

Sub-Note 3: Types

1. **Defining Own Types**
    - ***Constructors must start with capital letters***
    - We ***can define new discriminated union types (aka variants)*** as follows:

type mydutype = | Foo | Bar of int | Baz of int * string

    - ***Each value has an associated tag/constructor*** (e.g. Foo, Bar) that is immutable and can’t be overwritten, so it can be relied upon
    - Can use a constructor to create new values
        - Ex. let baz1 = Baz (1,”hello”) or let foo1 = Foo

    - Can be contrasted with unions in C/C++:
        - ***The tag Baz of int * string is essentially struct { int i; char * s; } baz;***
        - In a union, all members share the same memory location (start at offset 0)
            - In the following declaration:

union mydutype { int bar; struct { int i; char * s; } baz; }; mydutype u;

    - ***&u = &u.bar = &u.baz.i***. So, when we store a value in one union member, it overwrites the memory of the others, making them ***invalid***
    - Therefore up to the programmer to never access the wrong member (otherwise will get garbage) ⇒ ***very error-prone*** (runtime error)

- In OCaml, we use pattern matching to safely examine/deconstruct values:

```
match myval with
   | Foo -> 1
   | Bar n -> n + 7
   | Baz (n,_) -> n - 3
```

Tag and associated data are separate. The only way to access the data is through pattern matching, so we can never access invalid memory

Classic Example: Mimicking null pointers

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

The above is a type definition using ***generic types***. ***option is a user-defined type constructor – think of "None" as a null pointer*** (has no associated value) and ***"Some" as a piece of storage with an associated value of type ‘a***. We can then create values like below:

```
let x = Some "abc"
let y = None
```

NOTE: OCaml actually ***uses the above idea (options) to avoid common run-time errors (e.g. dereferencing null pointers)!***

We can even define more complex types:

```
type ‘a priorityQ = | Empty | Nonempty of (int * ‘a * ‘a priorityQ)
type ‘a list = | [] | :: of ‘a * ‘a list
type ‘a mylist = | EmptyList | Cons of ‘a * ‘a mylist
```

For priorityQ, the integer represents the "priority" of the item, followed by the value itself, followed by the tail. For list and mylist, the general meaning of list and mylist is the same.

Lecture 4

Sub-Note 1: Syntax (Form) vs Semantics (Meaning)

Sidenote: There's a deep connection between how we write software for computers and how people speak and write in natural languages (e.g. English). Both PL and natural languages require syntax to organize ideas into structured and meaningful communication. This connection made CS more "respectable" by showing it draws from broader linguistic and logical principles.

1. Syntax: Can be split into 2 distinct parts

- **Spec**: Defines how the program is supposed to behave without providing actual code
    - Focuses on ***expected behavior***, not implementation details
    - Ex. Specify input/output but omit the steps to achieve the end result
    - Higher-level of abstraction than writing the code
- **Implementation**: Refers to the ***actual code***, i.e. the ***steps and logic used to achieve the specification***

With syntax specifications, we write down what we want in a ***formal way***. We avoid using natural languages (e.g. English) and approximations (i.e. "hand-waving"); ***specs should be just as formal as a C++ program.*** In contrast, ML is informal.

Sentence Example 1: "Colorless green ideas sleep furiously" - Chomsky

- ***Syntactically correct BUT its semantics are nonsensical***
- What is the C++ equivalent?
    - Writing a function that adheres to the grammar but crashes during runtime (e.g. dividing by 0)

Sentence Example 2: "Ireland has leprechauns galore" - Eggert

- ***Makes sense semantically but syntactically incorrect*** – why?
    - "galore" doesn't fit the traditional sentence structure – is it an adjective? adverb?
    - "has leprechauns galore" is a verb phrase, making "galore" out-of-context
- What is the C++ equivalent?
    - ***There is none*** – it is impossible to write a program that makes sense semantically but violates syntax because the ***compiler would reject it***

Sentence Example 3: "Time flies"

- ***Makes sense both syntactically and semantically*** (form and meaning match), ***but it's ambiguous*** because it has multiple valid interpretations:
    1. Time passes quickly – declarative sentence (Noun Verb)
    2. Time how fast the flies are flying – imperative sentence (Verb Noun)
- Rely on context to resolve ambiguity
- This is a classic example of ambiguity

Ambiguity exists in programming – the same program can mean different things to different compilers. ***Programmers don't want ambiguity*** – it ***makes code less reliable*** since we don't know what it's going to do. Take the following C++ program:

```
int a, b;
...
return a+++++b;;
```

The last line of code is ambiguous because it can be parsed in multiple valid ways:

```
(a++)++) + b
(a++) + (++b)
a + (++(++b))
```

How do we handle/resolve ambiguity? One approach is to execute the code and see the output. A better approach is to have a specification for the language that explicitly defines how to handle such cases.

Sub-Note 2: Criteria for Good Syntax

What does it mean for a programming language to have ***well-designed syntax***?

1. Inertia: Want syntax that people are used to
- Ex. Consider the following math expression $(a + b) \cdot c$,
    - C writes (a + b) * c
    - Lisp writes (* (+ a b) c)
    - PostScript writes a b + c * (reverse Polish notation)
- We ***prefer C's syntax*** as it resembles traditional mathematical notation, so people are more used to it

2. Simplicity
- In the example above, we ***prefer Lisp*** because we don't need to worry about operator precedence and associativity (just look at the parentheses)

3. Unambiguous
- In the above example, all three language syntaxes are unambiguous, so there is no reason to prefer one over the other

4. Readable
- Correlates with inertia – if a user already knows a language, it's syntax is more readable
- Languages that allow names/identifiers tend to be easier to read

5. Writable
- Writable syntax seeks to minimize the # of keystrokes, ChatGPT prompts, overall effort required to write a program/idea
- Readability and writability often conflict – in APL, a program simulating the Game of Life might look like the following:

```
++/ιN,...
```

This syntax is ***writable (very compact) but not readable***. ***In commonly-used software***, where code is read more often than it's written, always ***prioritize readability***.

6. Redundant

- Redundancy in syntax provides additional checks to catch errors and reduces the likelihood of undetected bugs, ***improves reliability***
    - Ex. A language where you ***need to write every line of source code 2x*** would be ***impractical, but very reliable***
    - Ideally, we want programs with bugs to not compile
- Ex. In early versions of Fortran, you didn't need to declare variables explicitly; instead:
    - Identifiers starting with letters i-n were assumed to be ints
    - All other identifiers were assumed to be floats
    - **Problem**: Although this made programs ***more writable*** (e.g. we can write "i = 3" instead of "int i = 3")***, it was bad for redundancy*** – typos would go unnoticed, making it ***hard to debug***
        - Ex. If you misspelled some identifiers (e.g. using a "b" instead of "i"), the program is still valid even with the bug(s)

Sub-Note 3: Grammar

Grammar: Syntax specification
- Formal grammar is a formal specification of the syntax rules allowed in a PL
- ***To define grammar***, we ***need lower-level notions (e.g. tokens)*** which the grammar assumes to be built-in

Token: ***Lowest-level syntactic component*** in most programming languages
- Represents an ***individual unit*** of a language (e.g. ***keyword, operator***)
- In ***English***, ***words are*** treated as ***tokens*** – can take any distinct word and look it up
- ***Built from sequences of characters***
    - So, ***what defines a character***?
        - Naive Definition: A single keystroke
        - This definition does overlook some edge cases:
            - Pressing the SHIFT key doesn't produce a character
            - Many character are more complex than a single keystroke (e.g. the character $1/10$ (U+2152 in Unicode)

The ***main takeaway*** is that:
- A ***program is a sequence of bytes*** that ***represents a sequence of chars***
    - Not every byte sequence forms a valid char sequence
- This ***character sequence in turn represents a sequence of tokens***
    - However, the mapping from char sequences to token sequences isn't as generic and easy as mapping programs to char sequences

So, a token is simply a valid sequence of characters, but ***not every character sequence is a valid token sequence***. For instance:

$$char\ foo[] = "hello!';$$

The ***program won't compile*** since foo is an ***unterminated string*** (no ending quotation).

Sub-Note 4: Grammar Case Studies and BNF/EBNF

Case Study 1: Internet Standard RFC-5322

Emails have message headers, which often follow the format of RFC-5322. If an email doesn't conform to this standard, it likely won't be delivered.

A key element in headers is the messageID, which is used to uniquely identify an email. This allows systems to detect and discard copies. For example, a messageID might look like:

$$< eggert\$fsf933"abc"@cs.ucla.edu >$$

Important Aside: The grammar that governs the structure of the messageID uses Extended Backus-Naur Form (ENBF), an extension of ***Backus-Naur Form***. ***BNF is commonly used to define rules in context-free grammars***. It involves:

1. ***LHS*** (Nonterminal symbol), essentially a ***placeholder***
2. ***RHS*** (Finite sequence of symbols that can either be terminal or nonterminal)
    - ***Terminals*** are ***concrete values*** (e.g. strings)
    - ***Nonterminals*** are ***symbols that require further definition***

For example, given

$$messageID = "Message\text{-}ID:"\ msg\text{-}id\ CRLF$$

The LHS is the nonterminal placeholder $messageID$, and the RHS consists of $"Message\text{-}ID:"$ (terminal), $msg\text{-}id$ (nonterminal), and $CRLF$.

Case Study 1 (Continued):

$$msg\text{-}id = \text{"<" dot-atom-text "@" id-right ">"}$$

Here, $eggert\$fsf933"abc"$ matches to ***dot-atom-text***, and $cs.ucla.edu$ matches to ***id-right***. From just these two rules, it seems like we have a valid msg-id, but we must continue ***recursively checking the rules for the nonterminals id-right and dot-atom-text***:

$$id\text{-}right = \text{dot-atom-text / no-fold-literal}$$

The / is an example of a ***meta-notation*** in EBNF – not a terminal or nonterminal symbol, but instead a ***shorthand for BNFs***. In this particular grammar, / ***means the logical or***. We could have equivalently expressed id-right in BNF as:

$$id\text{-}right = \text{dot-atom-text}$$

$$id\text{-}right = \text{no-fold-literal}$$

Continuing,

$$no\text{-}fold\text{-}literal = "[" \text{ *dtext } "]"$$

OR, in BNF,

$$sdtext =$$
$$sdtext = dtext\ sdtext$$
$$no\text{-}fold\text{-}literal = "["\ sdtext\ "]"$$

The * is a meta-notation that says to repeat the symbol following it 0+ times (very similar to the regex *). Also, note that in the above, ***sdtext is a "made-up local variable"***. It is supposed to represent the equivalent of *dtext. dtext can be further defined:

$$dtext = \%d33\text{-}90\ /\ \%d94\text{-}126$$

dtext must be a single character within the ASCII range 33-90 or 94-126. We could equivalently write dtext as a combination of $dtext = \%d33$, then $dtext = \%d34$… in BNF. Finally, we have the the rules for dot-atom-text and atext:

$$dot\text{-}atom\text{-}text = 1{*}atext{*}("."1{*}atext)$$
$$atext = ALPHA\ /\ DIGIT\ /\ "!"\ ?\ "\#"$$

$1*$ is equivalent to *, but the symbol must appear at least once. $()$ is meta-notation that the * immediately preceding it applies to everything inside the $()$.

Observations

This ***grammar is significant because it can easily be converted into regex***. We can then feed the regex into a simple grep command, which is ***very fast***, much more than a C program:

$$\#!\ /bin/sh$$
$$pat = "\ldots"$$
$$grep\ "\$pat"\ file$$

However, ***not all grammar can easily be translated into regex***. Take the following:

$$s = "("\ s\ ")"$$
$$s = "a"$$

This ***recursive grammar can't be represented by regex*** because ***regex lacks the ability to count*** (e.g., track the number of opening or closing parentheses). BNF and EBNF, on the other hand, can support counting and recursion.

Likewise, in XML 1.1, it's impossible to write a regex to match XML data. The reason for this is that elements can recursively contain other elements inside it. Fundamentally, ***regex can't handle nested recursion***.

Lecture 5

ISO Standard for EBNF

Rules are written like $LHS = RHS;$

Terminal Symbol:	Literal values ⇒ $"terminal"$ OR $'terminal'$
Option	Repeat 0 or 1 time ⇒ $[option]$
Repetition:	Repeat 0 or more times ⇒ $\{repetition\}$ OR $*repetition$
Grouping:	$(grouping)$
Or Operator:	|
Exception Operator:	$-$ ($A - B$ means an instance of A that is NOT an instance of B)
Concatenation Operator:	, (A, B means an instance of A concatenated with an instance of B)

Example: In Prolog, identifiers can consist of the following characters:

$$\_,\ a\text{-}z,\ A\text{-}Z,\ 0\text{-}9$$

However, identifiers can't start with a digit, and they can't consist of a single underscore. ***In ISO EBNF***, this is equivalent to:

$ids = all\_ids - "\_"$; ⇒ the "–" indicates an ***exception***; exclude the single underscore
$all\_ids = \_A\text{-}Za\text{-}z,\ \{\_A\text{-}Za\text{-}z0\text{-}9\}$;

Case Study 1: ISO EBNF for EBNF itself

ISO EBNF can be used to describe the syntax of EBNF itself:

$$syntax = syntax\_rule,\ \{syntax\_rule\} ;$$
$$syntax\_rule = meta\_id,\ \textbf{'=',}\ definitions\_list,\ \textbf{';'} ;$$
$$definitions\_list = defn,\ \{'|',\ defn\} ;$$
$$defn = term,\ \{',',\ term\} ;$$

syntax represents the start symbol. From the above, a ***valid grammar consists of one or more syntax_rule's***.

Note that the rule $syntax\_rule$ is ***self-referential*** – it ***describes the structure of an EBNF rule using EBNF itself***. $meta\_id$ is "syntax_rule", followed by "=" separating the LHS and RHS, and $definitions\_list$ representing the RHS. This seems problematic BUT is ***good for self-verification*** and ensures a ***precise/uniform specification***. In fact, if we drew a comparison to C, it would be something like:

$$int\ x\ =\ x;$$

This is ***undefined behavior*** – x is declared but hasn't been initialized. The compiler tries to access the value of X on the RHS, but x doesn't have a value. However,

$$void\ *p\ =\ \&p;$$

is allowed. When p is declared, memory is allocated for it immediately. The compiler only needs the address of p, not the value.

Sub-Note 2: Common Pitfalls when Parsing Grammars

Tokenization is the process of ***breaking input into individual tokens*** while ***parsing*** is the process of ***interpreting these tokens***.

Common Pitfalls during Tokenization

1. **Confusion with Characters**
   - Ex. In C/C++, a program with the lines $int\ microsoft\ =\ 27;$ and $int\ microsoft\ =\ 28;$ ***could compile – how?*** In the second declaration, the "o" could be the cyrillic o, making them ***separate variables***
2. **Reserved Words**
   - Ex. $int\ float\ =\ 19;$ ⇒ isn't allowed in C since ***float is a reserved word***
     - Reserved words look like identifiers but can't be used as them
     - However, if we changed $float$ to $Float$, the code would execute
3. **Case-Sensitivity**
   - Case sensitivity varies between languages (e.g. C vs domain names)
   - Ex. In domain names, case is ignored – both https://ucla.edu and HTTPS://UCLA.edu resolve to the same website
4. **Greedy Tokenizers**
   - Tokenizers capture the longest possible token starting at the current character
     - Ex. For $int\ integral\ =\ 27$, capture $integral$ instead of int
     - Ex. $if\ (a\ <=\ b)$ ⇒ < is a valid token, but the tokenizer captures <=
     - ***Greedy is fast***
   - For a more concrete example, consider the following C code:

$$a\ =\ b{+}{+}{+}{+}{+}c$$

This ***code won't compile*** – because tokenizers are greedy, C tokenizes the expression as $a\ =\ (b{+}{+}){+}{+}){+}\ c)$. You would ***need parentheses to make it valid*** (e.g. $a\ =\ (b{+}{+}){+}({+}{+}c)$)

5. **Long Tokens**
    - Most IDEs aren't well-equipped to handle long strings (some characters won't be visible). In C, there are 2 main solutions:
        1. $static\ char\ const\ msg[]\ =$ "...... " "abc" "....... ";
            - ***Separate the long token into several consecutive smaller ones*** (without any operators in between them)
        2. $static\ char\ const\ msg[]\ =$ "\\............... \\
           .... \\
            - Use a \\ ***followed by an actual newline to extend a token*** across multiple lines in the IDE
            - Tricky to distinguish between the 2 uses if your source code actually contains "\\"
            - This approach could also lead to tricky bugs:

```
print("a");
// Which $%@! pushed to the main branch?? \
print("b");
print("c");
```

In the program above, we expect "abc", but only "ac" is printed. Even in the assembly, there are only 2 calls to print. This is because in C, \ followed by an actual newline means the comment continues onto the next line.

Pitfalls during Grammar

Here, we assume the issues related to tokenization (i.e. greedy tokenization, reserved words) has been addressed, and instead only focus on context-free grammars.

Grammar Introduction
- Here are the ***basic components of context-free grammars*** (e.g. BNF):
    1. Finite set of tokens/terminals
        - Could be the empty set
    2. Finite (and nonempty) set of nonterminals disjoint from the above
    3. A start symbol (must be a nonterminal)
    4. A rule set (must be nonempty)
        - At least one rule must have the start symbol as the LHS
- The basic definition for rules is $LHS\ (nonterminal)\ {-}{>}\ RHS\ (sequence\ of\ symbols)$
- **IMP**: You ***can have blind alleys in context-free grammars***

Problem 1: Useless Rules

Case Study 1: Unused Rules

- For simplicity, S will always be the start symbol (for future case studies as well)
- Capital letters denote nonterminals, lowercase letters denote terminals

$$S \text{ −> } a$$
$$T \text{ −> } b$$

Unused rules are those where the ***nonterminal on the LHS can't be reached from the start symbol***. Here, a is the only valid string since the second rule is never used. The analog in C:

$$int\ main()\ \{\ return\ 10;\ \}$$
$$int\ T()\ \{\ return\ 20;\ \}\ //\ unused\ function$$

Case Study 2: Blind Alley

$$S \text{ −> } a$$
$$S \text{ −> } bT$$
$$T \text{ −> } cT$$

A ***blind alley*** occurs when a ***grammar rule leads to infinite recursion***. Here, expanding $S \text{ −> } bT$ yields $bT \text{ −> } bcT \text{ −> } bccT$, a process that continues indefinitely.

Case Study 3: Duplicate Rule

$S \text{ −> }$
$S \text{ −> } T$
$T \text{ −> } aT$
$T \text{ −> } Ta$
$T \text{ −> } a$

While the grammar is valid, both $T \text{ −> } aT$ and $T \text{ −> } Ta$ recursively generate sequences of a. One rule is sufficient, so for example, $T \text{ −> } Ta$ could be removed.

Problem 2: Adding Constraints to Grammar

Sometimes we want to ***implement syntactic constraints***, or rules that restrict which expressions are considered valid. This is ***challenging to do in BNF and other context-free grammars***. Consider the following grammar:

$$\begin{array}{ll} S \text{ −> } NP\,VP & VP \text{ −> } v \\ NP \text{ −> } n & VP \text{ −> } VP\,adv \\ NP \text{ −> } adj\,NP & \end{array}$$

Examples of valid sentences include:
- $Dogs\ bark$
- $Maxwell\ meows\ loudly$

However, ***this grammar is too simple*** – it ***incorrectly matches invalid sentences*** like:

$$Maxwell\ meow\ loudly\ (verb\ is\ plural,\ subject\ is\ singular)$$

To enforce subject-verb agreement, ***we need to complicate the grammar*** to handle these exceptions. This added complexity comes with a ***tradeoff*** – adding constraints increases the size of the grammar, making it ***harder to read, maintain, and scale***.

$S \text{ -> } SNP\ SVP$ $SNP \text{ -> } adj\ SNP$
$S \text{ -> } PNP\ PVP$ $PNP \text{ -> } pn$
$SNP \text{ -> } sn$ $PNP \text{ -> } adj\ PNP$
… $(continue\ for\ VP)$

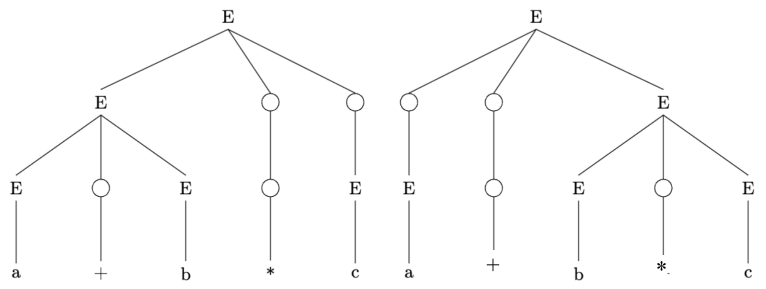
When faced with the above problem, ***PL will often keep the simpler grammar and document the constraint(s) using natural language*** (e.g. verb and subject must be in agreement).

Problem 3: Ambiguity

The ***biggest hassle*** when writing/validating context-free grammar is ***avoiding ambiguity***:

$E \text{ -> } E + E$ $E \text{ -> } '('\ E\ ')'$
$E \text{ -> } E * E$ $E \text{ -> } id$

The expression $a + b * c$ is correctly validated, but the ***grammar itself is ambiguous***. This type of grammar leads to ***abstract parse trees*** – result from ambiguous grammars and depend on a smart compiler to resolve ambiguity (although "incorrect", ***much easier to read***). We ***can create multiple valid parse trees*** for the expression, even though only one respects OOO:



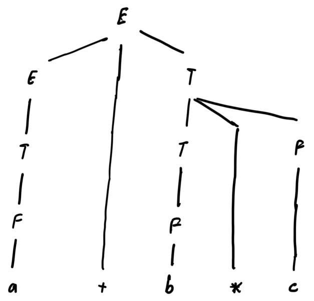
The right tree respects operator precedence (* before +). How do we eliminate ambiguity? We could keep the simpler grammar and document the OOO in English. The better approach is to ***complicate the grammar to remove parses we don't want***:

$E -> E + T$ $\quad$ $T -> F$

$E -> T$ $\quad$ $F -> id$

$T -> T * F$ $\quad$ $F -> '(' E ')'$

By making the grammar more complex, we've ***ensured both operator precedence and associativity*** are respected. This grammar produces ***concrete parse trees***, which result from ***unambiguous grammars*** but are ***more complicated*** and ***memory-intensive***, as they ***require more symbols*** to generate the same parse tree. Only one parse tree is now valid:



Case Study 2: C's Grammar

$stmt$:

';' ⇒ generates no machine code (does nothing)

$expr$ ';' ⇒ evaluate the expression and discard the result

$break$ ';' | $continue$ ';'

$return\ (expr)_{opt}$ ';' ⇒ return an optional expression

- This isn't correct – the grammar doesn't allow for valid returns like $return\ f(x) + 1;$
- Removing the parentheses, i.e. $stmt$: $return\ expr_{opt}$ ';' solves the problem

$goto\ ID$ ';' ⇒ jump to a labeled statement

$while\ (expr)\ stmt$

- Don't need a semicolon since it belongs to the $stmt$, NOT the $while$
- Here, the parentheses are required to avoid ambiguity; for example, $while\ 3 + 4$
  - Could be interpreted as $while\ 7$....
  - Could also be interpreted as $while\ 3$ then $+ 4$

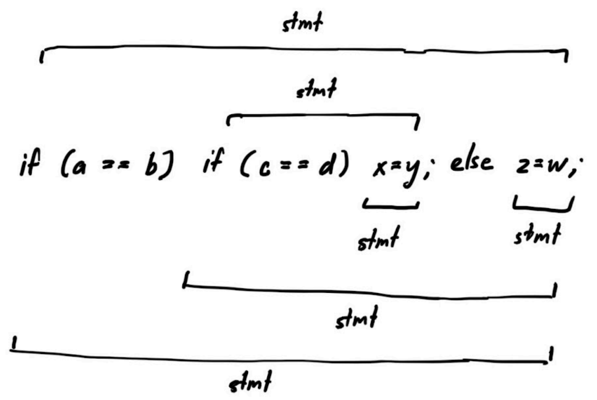
$do\ stmt\ while\ (expr)$ ';' ⇒ '; ' must belong to the do-while loop, so no ambiguity

$switch\ (expr)\ stmt$

$if\ (expr)\ stmt$

$if\ (expr)\ stmt\ else\ stmt$

There is ***still ambiguity***; for example: $if\ (a == b)\ if\ (c == d)\ x = y;\ else\ z = w;$



The grammar allows 2 valid parsing, but only one interpretation is actually correct (the bottom one). This problem, known as the ***dangling else***, can be addressed by ***complicating the grammar*** through introducing a new nonterminal:

$stmt$: stm | if $(expr)$ $stmt$

stm: ';' | $expr$ ';' | $break$ ';' | $continue$ ';' | ... (except the if statement in stmt)

Lecture 6

Sub-Note 1: Problems with Parsing

There are three main challenges related to parsing:

1. Recursion in Parsers
- Grammars are typically recursive

2. Handling Disjunction (Fancy term for "or")
- Nonterminals often have multiple differing grammar rules: Ex. $S -> T \mid S + T$

3. Handling Concatenation
- Ex. $S -> A, B$

We ***can use matchers and acceptors to address disjunction and concatenation***:
- **Matcher**: Part of a compiler's parser, matches a portion of the input string
    - Matcher addresses concatenation since matcher matches a and acceptor looks at the b that follows
    - Matcher solves disjunction because it can match different initial prefixes
- **Acceptor**: Validates the suffix of the matched string

$V\,0$: $input\ sequence\ -> bool$

Basic Matcher: Outputs "yes" if the input has valid syntax or "no" otherwise
- In other words, a matcher checks if the input token sequence is in the language

$V\,0.1$: $grammar\ -> matcher$

- We want to be able to generate a matcher function from a grammar

How do we handle disjunction? Since there are multiple possibilities for a match we ***need backtracking***, and more specifically, an ***acceptor***

$V\ 0.2$: $matcher$: $frag\ {-}{>}\ acceptor\ {-}{>}\ bool$

- Function that tells the matcher whether the unmatched suffix is acceptable
- Now, a matcher is a function that takes a token sequence (i.e. fragment) and acceptor

$V\ 0.3$: $matcher$: $acceptor\ {-}{>}\ frag\ {-}{>}\ bool$

In Eggert's proposed version, the arguments are flipped

- Although this order seems counterintuitive, it makes composing functions easier

$V\ 0.4$: $matcher$: $acceptor\ {-}{>}\ frag\ {-}{>}\ frag\ option$

- Don't match the entire input, just an ***initial prefix*** ⇒ makes the matcher ***more flexible***
- Matcher takes in an acceptor which takes in a fragment; the matcher then parses an initial prefix of the fragment, and if it finds an acceptable suffix, it returns it
- Since $frag\ {-}{>}\ frag\ option$ is just an acceptor,
    - In other words, a matcher takes in an acceptor and returns a ***"pickier" acceptor***

Sub-Note 2: Simple Acceptors and Matcher

Acceptors: Functions from $frag\ {-}{>}\ frag\ option$

An acceptor that matches nothing (no input works):

$let\ accept\_none\ =\ fun\ frag\ {-}{>}\ None$ OR

$let\ accept\_none\ \_\ =\ None$

An acceptor that matches everything (all inputs work):

$let\ accept\_all\ frag\ =\ Some\ frag$

An acceptor that only matches non-empty fragments (only [] doesn't work):

$let\ accept\_nonempty\ =\ function\ |\ []\ \ {-}{>}\ None\ |\ frag\ \ {-}{>}\ \ Some\ frag$

Matchers: Must also verify that the acceptor likes the unmatched suffix

A matcher that only matches the empty string (passes entire input to acc):

$let\ match\_empty\ acc\ frag\ =\ acc\ frag$ OR

$let\ match\_empty\ =\ (fun\ acc\ {-}{>}\ (fun\ frag\ {-}{>}\ (acc\ frag)))$

- Note this simplifies to $let\ match\_empty\ =\ fun\ acc\ {-}{>}\ acc$ (identity function)

A matcher that matches nothing (no input works):

$let\ match\_nothing\ =\ fun\ acc\ {-}{>}\ fun\ frag\ {-}{>}\ None$ OR

$let\ match\_nothing\ acc\ frag\ =\ None$ OR

$let\ match\_nothing\ \_\ \_\ =\ None$

A matcher that always succeeds:
$let\ match\_anything\ \_\ frag\ =\ Some\ frag$

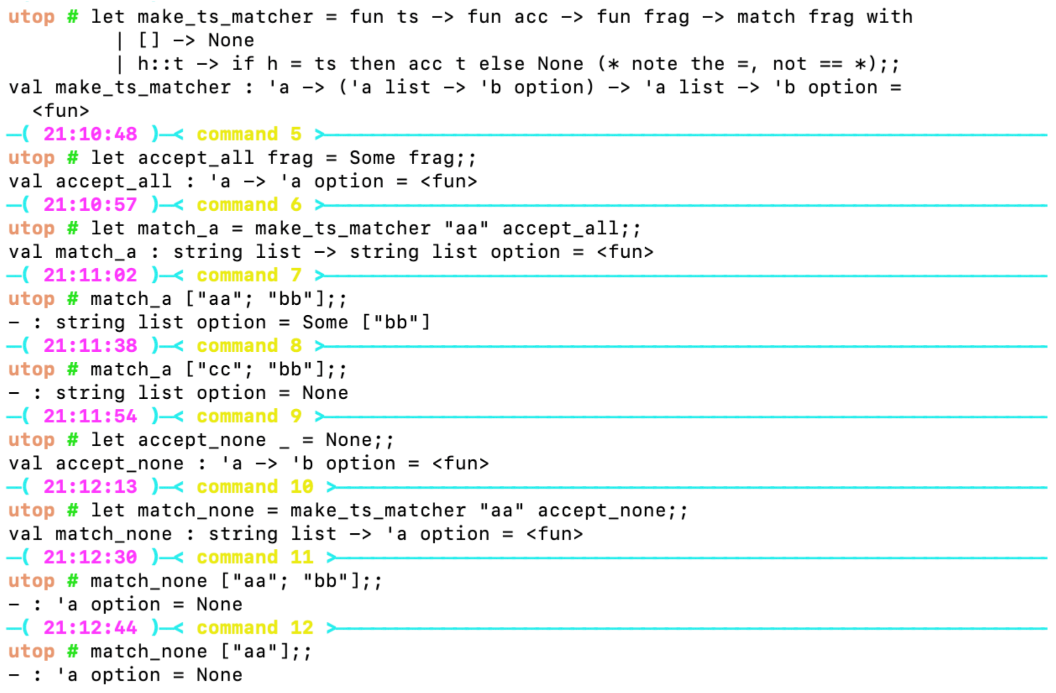
- $match\_empty$ ***succeeds only the acceptor likes it*** BUT $match\_anything$ ***succeeds regardless*** ⇒ the latter isn't really a matcher

Sub-Note 3: Disjunction

Suppose we have the following grammar: $S\ -> \ A\ |\ B$. Assuming we already have a matcher for A and B, is it possible to build a matcher for S?

Our target function is a function $make\_matcher$: $symbol\ -> \ matcher$ that takes in a piece of the grammar (e.g. a symbol like $T\ "abc"$) and returns a matcher

First, let's start small and make a matcher for a single terminal symbol:
$make\_ts\_matcher$: $'t\ -> \ ('nt, 't)$ matcher



```
utop # let make_ts_matcher = fun ts -> fun acc -> fun frag -> match frag with
        | [] -> None
        | h::t -> if h = ts then acc t else None (* note the =, not == *);;
val make_ts_matcher : 'a -> ('a list -> 'b option) -> 'a list -> 'b option =
  <fun>
─( 21:10:48 )─< command 5 >──────────────────────────────────────────────────
utop # let accept_all frag = Some frag;;
val accept_all : 'a -> 'a option = <fun>
─( 21:10:57 )─< command 6 >──────────────────────────────────────────────────
utop # let match_a = make_ts_matcher "aa" accept_all;;
val match_a : string list -> string list option = <fun>
─( 21:11:02 )─< command 7 >──────────────────────────────────────────────────
utop # match_a ["aa"; "bb"];;
- : string list option = Some ["bb"]
─( 21:11:38 )─< command 8 >──────────────────────────────────────────────────
utop # match_a ["cc"; "bb"];;
- : string list option = None
─( 21:11:54 )─< command 9 >──────────────────────────────────────────────────
utop # let accept_none _ = None;;
val accept_none : 'a -> 'b option = <fun>
─( 21:12:13 )─< command 10 >─────────────────────────────────────────────────
utop # let match_none = make_ts_matcher "aa" accept_none;;
val match_none : string list -> 'a option = <fun>
─( 21:12:30 )─< command 11 >─────────────────────────────────────────────────
utop # match_none ["aa"; "bb"];;
- : 'a option = None
─( 21:12:44 )─< command 12 >─────────────────────────────────────────────────
utop # match_none ["aa"];;
- : 'a option = None
```

- In the above, the matcher generated by $make\_ts\_matcher$ ***checks if the first element is "aa"***. If it is, the matcher passes the remaining list to the acceptor. If the acceptor approves the tail, the matcher returns Some with the accepted suffix

This can be simplified to:
$let\ make\_ts\_matcher\ =\ fun\ ts\ -> \ fun\ acc\ -> \ function$
$|\ []\ -> \ None \quad (*\ function\ works\ since\ we\ immediately\ turn\ around\ and\ use\ frag\ *)$
$|\ h{::}\ t\ -> \ if\ h\ =\ ts\ then\ acc\ t\ else\ None$

and ***further into***:

```
let make_ts_matcher ts acc = function
  | [] -> None     (* function works since we immediately turn around and use frag *)
  | h::t -> if h = ts then acc t else None
```

However, ***we can't simplify*** `h::t -> if h = ts then acc t else None` **to** `ts::t -> acc t`

- Every variable mentioned is considered a newly defined local variable, SO the `ts` in the match is different from the input `ts`

NOW, we can return to writing `make_dsj_matcher`, `make_matcher`:

```
let make_matcher = function
  | [] -> match_empty
  | T x -> make_ts_matcher x
  | Or (a, b) -> make_dsj_matcher (a, b);;  (* match anything that a or b matches *)

let make_dsj_matcher (a, b) =
  let ma = make_matcher a (* create a separate matcher for each ts *)
  and mb = make_matcher b in
    fun acc -> fun frag -> match (ma frag acc) with
   (*  try ma on the fragment and if the suffix is accepted, it works  *)
      | Some x -> Some x
      | None -> match (mb frag acc) with (* try with mb if ma failed *)
         | Some x -> Some x
         | None -> None;;
```

We can further compress `make_dsj_matcher`. Specifically, we can simplify the innermost `match` statement to just `mb frag acc`. We can also simplify `Some x -> Some x` to just `x -> x` and move it below the `None` match. Altogether,

```
let make_dsj_matcher (a, b) =
  let ma = make_ts_matcher a (* create a separate matcher for each ts *)
  and mb = make_ts_matcher b in
    fun acc -> fun frag -> match (ma frag acc) with
   (*  try ma on the fragment and if the suffix is accepted, it works  *)
      | None -> mb frag acc (* try with mb if ma failed *)
      | x -> x
```

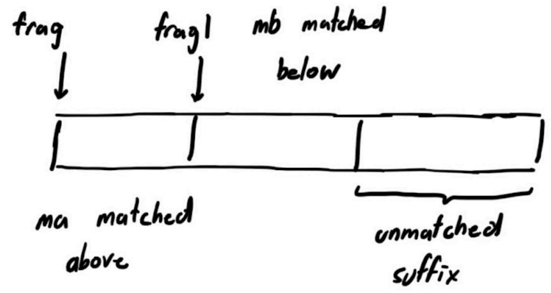
What if we have more than 2 symbols? We can include the following function:

```
let rec make_or_matcher = function
  | []  -> match_empty
 | h::t -> let hm = make_matcher h (* head matcher *)
```

```
            $and\ tm\ =\ make\_or\_matcher\ t\ (*\ tail\ matcher\ *)$
  $(*\ plug\ in\ the\ function\ from\ above\ *)$
```

Sub-Note 4: Concatenation

Suppose we have the following grammar: $S\ -> \ A, B$. How can we build a matcher for S?



The basic idea is that ma matches an initial prefix of $frag$, then mb matches an initial prefix of that, leaving an unmatched suffix

- In other words, we call ma with a fancy acceptor (it will work on the suffix that includes the part mb matched and the unmatched suffix, which I believe is frag1)

```
$let\ rec\ make\_concat\_matcher\ (a, b)\ =$
  $let\ ma\ =\ make\_matcher\ a$
  $and\ mb\ =\ make\_matcher\ b$
  $in\ fun\ acc\ ->\ fun\ frag\ ->$
  $ma\ (fun\ frag1\ ->\ mb\ acc\ frag1)$
```

We can further simplify the last 2 lines into:

```
$in\ fun\ acc\ ->$
$ma\ (mb\ acc)$
```

What if we have more than 2 symbols? We can include the following functions:

```
$let\ append\_matchers\ matcher1\ matcher2\ acc\ frag\ =$
  $matcher1\ frag\ (fun\ frag1\ \ ->\ \ matcher2\ frag1\ accept)$

$let\ make\_appended\_matchers\ ls\ =\ (*\ ls\ is\ the\ list\ of\ patterns\ *)$
  $let\ rec\ mams\ =\ function$
    | [] $->\ match\_empty$
    $h{::}\ t\ ->\ append\_matchers\ (make\_matcher\ h)\ (mams\ t)$
  $in\ mams\ ls$
```

Overall, the combination of make_dsj_matcher and make_concat_matcher does backtracking.

Sub-Note 5: Translation Stages in Compilers (GCC)

Modern compilers use advanced techniques to optimize and handle edge cases – such as avoiding infinite recursion from a badly-defined grammar – but the basic idea is the same: given a grammar, generate a parser.

Approach 1: The model below outlines the standard stages used to ***build low-level applications***, like Firefox and Chrome. ***If you want high-performance code, use this approach.*** Imagine the following program:

```
#include <stdio.h>\n int main() { return !getchar(); }\n
```

Stage 0: Tokenization

- Process of ***converting source code*** (i.e. series of bytes) into a ***sequence of tokens***, which can be inputted into a parser
    - Often eliminates comments/whitespace
- **Limitations**: Can't determine context-sensitive details (e.g. whether a string like fi is a misspelling of the keyword if or an actual identifier)
    - Only handles simple pattern matching (e.g. detecting invalid characters)
- **Result**: After tokenization, the parser receives a series of terminal symbols

Tokens vs Lexemes

- **Token**: Category representing a ***group of chars with a collective meaning*** ⇒ Ex. id
    - Often represented by numeric codes ⇒ Ex. id is token #2, int is token #27
    - For the parser, tokens with the same numeric code are treated identically, even if they have different names
- **Lexeme**: ***Actual sequence of characters*** forming an instance of a token
    - Ex. if the token is id, the lexeme could be $count,\ main$

Stage 1: Parsing

- Analyze the sequence of tokens and check if they're ***syntactically valid***
- ***If yes, a parse tree is generated***, where the root is a nonterminal symbol like "$program$" and each child represents a parsed element of the language

Stage 2: Semantic Analysis

- Validates the parse tree to ensure the program is ***semantically correct***
- Will ***perform static checking*** ⇒ includes both type checking and verifying that every identifier is declared before its used
- Outputs an ***attributed parse tree*** – parse tree where each node carries additional information (e.g. for an identifier node, this can include the identifier name and type)

Stage 3: Code Generation

- ***Convert the attributed parse tree into assembly language***
- We can view the assembly code by running gcc -$S\ foo.c \Rightarrow$ creates $foo.s$
- Assembly code can't directly be executed since it's not machine code

Stage 4: Assembly
- Takes in $foo.s$ (text file containing assembly instructions) and generates $foo.o$ (binary file containing machine instructions)
  - Some parts of $foo.o$ are left blank to be filled in later by the linker

Stage 5: Linker
- Takes in $foo.o$ and links it with necessary libraries (e.g. using files like $libc.so$), filling in any unresolved references (e.g. library function calls)
- Generates an executable foo

Stage 6: Loader
- Part of the OS kernel in Linux; loads the executable into RAM and executes it

Approach 2: Integrated Development Environment (IDE)
- IDEs incorporate a ***compiler that operates entirely in main memory***, translating source code strings into machine code and immediately executing them
- ***Good for rapid development*** since editing, compiling, and running occur ***within a single program***
- However, if a failure occurs, ***both the program and IDE can crash***

Approach 3: Hybrid Environment (Pioneered by Java, OCaml, JavaScript)
- ***Not as dangerous as an IDE but still has its efficiency***
- Interpreter compiles source code into an intermediate representation (e.g. bytecode) and executes it in a controlled environment
  - Ex. If we add 2 integer, the interpreter checks for overflow (if there is, abort)
- Often ***combined with a JIT compiler***
  - Used to enhance the performance of interpreted languages
  - Whereas traditional compilation converts the entire program into machine code before execution, a ***JIT compiler does it during execution***
  - ***Profiles the program***, compiling the most frequently-used sections into machine code for faster execution
  - **Advantages**: Optimizes for the specific CPU and reduces unnecessary work
  - **Disadvantages**: Uses more memory and has runtime compilation overhead

---

Lecture 7

Sub-Note 1: Types

A type can mean different things to different computer scientists:

Definition 1: A predefined or user-defined data structure (e.g. int, struct)

- **Counterexample**: In C, $enum\ \{A, B, C\}$ is considered a type, but doesn't match the traditional definition of a data structure

Definition 2: Set of things that the compiler knows about a value

- **Value**: Result from evaluating an expression; for instance, if n is an integer, evaluating $n + 1$ yields a value of type int
    - This isn't really correct – the actual range of values for $n + 1$ should be $[INT\_MIN + 1, INT\_MAX]$, but the compiler uses $[INT\_MIN, INT\_MAX]$
    - Overflow can technically create INT_MIN, but that's ***undefined behavior*** in C

As a further counterexample, consider the following code:

```
for (int i = INT_MAX − 1; i ≤ INT_MAX; i++)
    print(i);
```

Because of overflow, this code results in undefined behavior (***program is unpredictable*** and potentially dangerous). For a more reflective counterexample, consider the following code written in Python:

```
def f(x, y):
  return x < y
```

Here, the types of x and y are ***NOT known at compile-time*** and are checked dynamically. If the types differ at runtime, the interpreter raises an error. By definition, Python doesn't agree with our second definition.

Definition 3: Set of values

Definition 4 (BEST): Set of values and associated operations

- Object-oriented languages use this definition (including Python and C++)
- In C++, we have the concept of a "class" which encompasses both sets

Primitive vs Constructed Types

- **Primitive**: Basic, built-in types (e.g. int, char)
    - ***Don't always mean the same thing in different languages***
    - Ex. $ints$ in C/C++ only need to store numbers [-$32767, 32767$], but macros (e.g. INT_MAX) may evaluate to different values on different implementations
    - Ex. int could also mean a mathematical integer (no limit on size) ⇒ this allows us to NOT worry about overflow
    - Ex. In JavaScript, $ints$ are a subset of $floats$

Case Study 1: JavaScript Floats

Represents floats using the IEEE 754 floating-point format. For a 32-bit representation:

- **Sign Bit** (MSB) ⇒ 0 for positive, 1 for negative
- **Exponent** (Next 8 bits)
- **Mantissa** (Next 23 bits)

The value of a floating-point is calculated differently based on the exponent field:

$(-1)^s * 2^{e-127} * 1.(mantissa)$ ⇒ ***Normalized*** values (when $e \neq 0$ and $e \neq 255$)
$(-1)^s * 2^{-126} \cdot 0.(mantissa)$ ⇒ ***Denormalized*** values (when $e = 0$)
$(-1)^s * Infinity$ ⇒ When $e = 255$ and $f = 0$
NaN ⇒ When $e = 255$ and $f \neq 0$

Denormalized values are needed to represent ***"tiny numbers", including 0***. More importantly, this prevents abrupt underflow to 0. Take the following hypothetical:

```
$float\ x, y;$
$if\ (x \neq y)$
        $print(x - y);$
```

Ideally, we never want the $print()$ to return 0, but with just the formula for normalized values, small numbers would underflow to 0. For denormalized numbers, when $f = 0$, we can create both $+0$ (000...000) and -0 (100...000), which we couldn't do with just the definition for normalized values.

We separate other cases ($e = 255$) because of the following scenario:

1. When $f = 0$, we get $infinity$
    - In earlier computing systems, if overflow occurred, an exception would be thrown. This was later revised to improve robustness. Now, we can either:
        1. **Trap**: Built-in to the hardware, ***async event*** causing the program to abort
        2. Keep going and ***return infinity*** (***default*** case)
    - We can obtain infinity by multiplying the max representable float by 2
    - We can obtain infinity by dividing $1/0$ (this would crash for integer arithmetic)
2. When $f \neq 0$, we get NaN
    - What should $inf - inf$ evaluate to? $0/0$? We return NaN in these cases

Important: $x == y$ and $memcmp(\&x, \&y, sizeof(x))$ aren't always equal:

1. If $x = +0$ and $y =$ -0, the former is true (numerically equal) BUT the latter is false (bit representations differ)
2. The opposite is true for NaN's – NaN always evaluates to false no matter what it's compared to (e.g. $NaN \neq x$ for all x; $NaN \neq NaN$ numerically)

Sub-Note 2: What are types used for?

1. **Annotations**
    - Types act as “comments” for the programmer and “execution hints” for the compiler (e.g. a float variable should be stored in a float register for best efficiency)
    - Can affect execution (e.g. casting a $float$ to int) ⇒ $float\ x;\ (int)\ x;$
2. **Inference**
    - Types allow us to infer further properties of a program
    - Ex. $int\ i\ = ...;\ float\ f\ = ...;\ i\ *\ f$ ⇒ What is the type of $i\ *\ f$? In C, it’s inferred as float
3. **Checking**
    - Helps programmers catch mistakes and prevent minor errors from snowballing
    - Two types:
        - **Static**: Checks ***before*** execution, guaranteed no type errors at runtime
            - Type is “bound” to the variable
            - ***Faster*** execution and more ***reliable***
        - **Dynamic**: Checks ***during*** execution
            - Type is “bound” to the value
            - More ***flexible*** and forgiving (some programs may run using dynamic type checking but not static)
        - Aren’t mutually exclusive ⇒ for example, Java is mostly statically-typed but performs dynamic type checks during casting
    - **Strongly Typed**: Not clearly defined, but more or less “can’t cheat typing”
            - OCaml is statically strong while Python is dynamically strong
            - C/C++ is weakly typed – any pointer type can be converted to another pointer type through casting

Type Equivalence
- Given 2 types T and U, how do we know if they’re the same type (i.e. is $T\ =\ U$)?
- There are 2 types of equivalence:

1. **Name Equivalence**: Equivalent if they have the ***same name***

For example, imagine the following structs:
$struct\ s\ \{\ int\ val;\ \ struct\ s^*\ \ next;\ \ \};$
$struct\ t\ \{\ int\ val;\ \ struct\ t^*\ \ next;\ \ \};$

Because ***C/C++ uses name equivalence for classes and structs***, these represent 2 distinct types because they have different names (even though they have the same underlying representation). This can be seen below:

Case 1: $s\ v\ =\ ...;$ $t\ w\ =\ ...;$ $v\ =\ w;$ ⇒ Won’t compile since v and w are different types
Case 2: $s\ v\ =\ ...;$ $s\ w\ =\ ...;$ $v\ =\ w;$ ⇒ Would compile

2. **Structural Equivalence**: Equivalent if their ***underlying representations are identical*** (have same data structure internally)

$typedef\ int\ s;$
$typedef\ int\ t;$
$s\ x\ = ...;$
$t\ w\ = ...;$
$x\ =\ w;$ ⇒ ***s and t are aliases of int***

Except for classes and structs, ***C/C++ uses structural equivalence***, so the above is allowed. Similarly, Java uses name equivalence for classes and structural equivalence for scalars (e.g. integers, booleans, characters).

Flavors of Types

- There are 2 flavors of types:

1. **Abstract Type**
   - ***Know the name and operations*** of the type, but the internal ***implementation is hidden***
   - Encapsulate their implementation, providing flexibility and modularity
   - Work best with name equivalence
2. **Exposed Type**
   - ***Implementation is visible*** to the compiler, allowing for ***optimization and more efficient code***
   - Users are tightly bound to the current implementation
   - $float$ is generally considered an exposed type since we know its representation follows the IEEE-754 standard

Subtype Polymorphism

- Given 2 types T and U, how do we know if T is a subtype of U?
- Subtypes either ***have the same or more operations*** than their super types
- A type T is a subtype of U if it ***can be used anywhere*** U ***is expected***:

$type\ day\ =\ Sunday\ |\ Monday\ |\ Tuesday\ ...\ |\ Saturday$
$type\ weekday\ =\ Monday\ |\ ...\ Friday$

$weekday$ is a subtype of day since every $weekday$ is a day. So, if we have a function that takes in a parameter of type day, it can accept a $weekday$ too.

Sub-Note 3: Polymorphism

Purpose: Some languages use ***dynamic type checking***, making polymorphism irrelevant. This allows functions to accept any argument type, deferring type validation to runtime

- While some appreciate this ***flexibility***, others find it leads to ***undetected errors***
- ***Polymorphism balances this flexibility with static type safety***, enabling functions to handle multiple types while preventing runtime type errors

For example, the function $f(x)$ could have different interpretations based on the type of x or the return type of $f()$. There are 2 main categories:

1. **Ad-Hoc Polymorphism (Disorganized)**
   - **Overloading**: A ***single function name*** refers to ***multiple implementations***
     - Compiler selects the appropriate implementation based on the argument types
     - Ex. Calling $cos(x)$ invokes either $cosf()$ for floats or $cosd()$ for doubles at the machine level (hidden from the user), though you always use $cos()$
     - Under the hood, ***compilers create separate definitions with unique names***, fixing each reference to use the correct name according to the involved types
   - **Coercion**: ***Implicit type conversion*** done by the compiler
     - Good since you ***don't have to write obvious type conversions explicitly***
     - Bad since it ***can perform not-so-obvious conversions unexpectedly***
     - Ex. If x is of type $float$ and i is of type int, $i * x$ triggers an automatic conversion of i to $float$, effectively computing $x * ((float)\ i)$

To illustrate the harms of coercion, take the following program:
$unsigned\ int\ x = -1$
$if\ (x < 0)\ print("ouch")$ // never prints "ouch" for any input x

Since uid_t ***is unsigned, -***1 ***is coerced*** and becomes the max unsigned value. 0 is also converted to unsigned for the comparison since we can't directly compare signed with unsigned.

Additionally, imagine the following scenario:
$int\ f(float,\ int)$
$int\ f(int,\ float)$
$f(3, 7)$

What implementation does $f(3, 7)$ resolve to? Do we coerce the first or second argument? In C/C++,the ***compiler will likely crash if there is ambiguity*** similar to the above.

2. **Parametric Polymorphism (Organized)**
   - You ***write code/functions that don't assume a specific type***, allowing you to use that function with a multitude of types
     - Ex. In OCaml, a list can be defined as $'a\ list$ where $'a$ is a type parameter ⇒ we don't know the type of $'a$ until the list is instantiated
   - Referred to as ***generic programming*** in object-oriented languages

Case Study 1: Traditional Java

In Java, the type $Object$ is the root of the class hierarchy ($Thread,\ String$ are subclasses). Any object can therefore be assigned to a variable of type $Object$:

```
Object o = "Hello!";
```

However, if you want to use o as a specific type, such as a $String$, you need to cast it:
$String\ s = (String)\ o;$ ⇒ this casting introduces a runtime check to ensure that o is actually a string. If it isn't, a $ClassCastException$ is thrown.

Example 1: Iterating Over a Collection of Strings

Suppose you have a collection of strings and want to remove all strings of length 1:

```
Collection c = {...} // Bunch of objects
for (Iterator i = c.iterator(); i.hasNext(); ) {
    String temp = (String) i.next(); // returns the curr element and shifts iterator forward
    if (temp.length() == 1) i.remove(); }
```

The code above works, but ***lacks compile-time safety***. Additionally, we ***must explicitly cast each object*** since the $next()$ method returns $Object$ references, and the compiler doesn't automatically know their actual types.

Solution: Modern Java uses ***generic types***

Through generics, we can ***specify the type*** of elements in a collection ***at compile-time***. The compiler only needs to verify that only $String$ objects are being added to the collection, which ***eliminates the need for casting and enhances the type safety of the code***:

```
Collection<String> c = {...};
for (Iterator<String>  i = c.iterator(); i.hasNext(); )
    if (i.next().length() == 1) i.remove(); // i.next() refers to a string
```

Lecture 8

Sub-Note 1: Generics vs Templates

- Generics are used in higher-level languages (e.g. OCaml, Java) while templates are used in lower-level languages (e.g. C++) that care more about efficiency
- In ***templates***, the compiler generates ***separate code for each type used*** (multiple versions of the same function) ***before runtime***
    - Static checking at ***type instantiation***
    - Could lead to code bloating but ***can optimize better for a specific type***
    - Don't retain type information at runtime ⇒ makes debugging harder
- Generics implement type erasure (generic type information is removed at runtime)
    - Static checking at ***definition***
    - ***Cleaner but less flexible/efficient***
    - Must generate code that is general enough to work on different types

Sub-Note 2: Extends vs Super

Example 1

$List{<}String{>}\ \ ls = \ldots;$
$List{<}Object{>}\ \ lo = ls;\ //\ Compile\ error$
$lo.add(new\ Thread());$
$String\ s = ls.get(0);\ //\ result\ should\ be\ of\ type\ string\ since\ ls\ is\ of\ type\ string$

Recall that in Java, $Object$ ***is the root of all types.*** In addition, Java is ***reference-based*** – ls and lo point to the same list (lo isn't declared with the keyword new). So, strings are a subtype of objects. HOWEVER, in Java, a $List{<}String{>}$ ***is NOT a subtype of*** $List{<}Object{>}$

In general, ***if*** X ***is a subtype of*** Y***, every operation on instances of*** Y ***should work on*** X***.*** Java prevents $List{<}Object{>}\ lo = ls$; to maintain type safety:

- lo and ls point to the same list (since Java is reference-based, and lo isn't declared with the new keyword)
- Since lo is a $List{<}Object{>}$, we can add a $Thread$ object into lo, but that modifies ls as well ⇒ $ls.get(0)$ then returns a non-string, ***violating type safety***

Example 2:

What does $List\ {<}?\ extends\ Orange\ {>}$ ***mean?***

- Can hold elements of ***any type that is a subclass of*** $Orange$ (e.g. $BloodOrange$)
- The list is a producer; we can ***read elements as*** $Orange$ BUT NOT $BloodOrange$
- We also ***can't add new elements*** because the ***exact subtype is unknown***

$List\ {<}?\ extends\ Orange\ {>}\ oranges = new\ ArrayList\ {<}{>}\ ();$
$oranges.add(new\ Orange());\ //compilation\ error$

The reason we can't do the above is that the list could be pointing to a subtype (e.g. $List{<}$ $BloodOrange{>}$); adding a general $Orange$ would violate type safety

What does $List\ {<}?\ super\ BloodOrange\ {>}$ ***mean?***

- Can hold elements of type $BloodOrange$ ***or any of its supertypes*** (e.g. $Orange,\ Fruit$)
- The list is a consumer; you can ***add*** $BloodOrange$ ***instances*** to it
- We ***can't add supertypes*** (e.g. $Orange$) since the list could be typed to $BloodOrange$
- Can only safely read elements as $Object$

PECS (Producer Extends, Consumer Super)

- If you are only ***reading items*** from a generic collection, it is a producer and you should ***use the "extends" keyword***
- If you are only ***adding items***, it is a consumer and you should ***use "super"***

- If you are doing ***both, don't use either***

Aside on C/C++ Pointers:

A `char *` is a pointer to a char. However, contrary to popular belief, a `const char */char const *` is NOT a pointer to a character that can't be modified. Imagine the following:

```
char c = 'A';
char const * p = &c;
print(* p) // prints 'A'
char c = 'B';
print(* p) // prints 'B'
```

In actuality, `char const *` means a ***pointer to a character that can't be modified through that pointer***. If the actual character data isn't constant, it can still be changed through other means.

Example 3: Wildcard

Consider a function that prints every element in a list:

```
void printList(List<Object> & l) {
        for (Object o: l) // Syntactic sugar to create an iterator on a list
        System.out.println(o);
}
```

Although we can pass in a `List<Object>`, we can't pass in a `List<String>`, since it's not a subtype of a `List<Object>`. The ***? wildcard*** allows the function ***accept a list of any type***:

```
void printList(List<?> & l) {
        // String s: l would result in a compile-time error
        for (Object o: l) // works because everything is at least an object
        System.out.println(o);
}
```

Example 4: Bounded Wildcard

```
public void printShapes(Collection <? > shapes) {
        for (Shape s: shapes)
                printShape(s); // assume for now this just prints a shape
}
```

This is a compile-time error since `?` represents an unknown type – the compiler can't guarantee that all elements in `Shape` are `Shape` objects. To address this, we can use ***bounded wildcards***:

```
public void printShapes(Collection <? extends Shape > shapes) {
```

Now, we can pass in a $Collection$<$Rectangle$>, but not $Collection$<$String$>, since $String$ is not a subtype of $Shape$. Note every type is $super$ and $extends$ to itself. However, there is still useful code that you can't write. Imagine we want to copy an array into a collection:

```
void copy(Object[] arr, Collection<Object> co) {
        for (Object o: arr)
                co.add(o);
}
```

This only works if arr and co hold instances of $Object$; it won't work if we want to copy an array of strings into a collection of strings. To generalize, we can try the following:

```
void copy(? [] arr, Collection<?> co) {
```

However, this is ***too generous*** – if this worked, we could copy an array of strings into a collection of threads. Instead, we ***use <T>*** to tell Java that we want the ? to be the same:

```
<T> void cvt(T [] arr, Collection <T> co) {
        for (T o: arr)
                co.add(o);
}
```

Now, we can copy from an array of strings into a collection of strings, BUT we ***can't copy an array of rectangles into a collection of shapes*** (which makes sense since $Shape$ is the supertype of $Rectangle$). So, we ***use the*** $super$ ***keyword***:

```
<T> void cvt(T [] arr, Collection <? super T> co) {
        for (T o: arr)
                co.add(o);
}
```

Alternatively, we could have used <T, U>, and replaced ? with U, but if we only see a type variable once, it is convention to use ?.

Implementation Notes: How are generics implemented in Java?

1. Using Generics with Type Erasure
    - In Java, ***every object is stored as a reference*** (or pointer) to its underlying value, with the language enforcing a strong type system
    - Although you may write code like $List$<$String$> or $List$<$Integer$>, because of ***type erasure***, the ***runtime only sees one raw type descriptor*** – just $List$

- **Recap**: Compiler uses the generic type information to enforce type safety at compile-time, but then removes the specific type parameters – so ***at runtime, we only know a list exists, we don't know the type of its elements***

2. Alternative and Simpler Approach to Generics: **Duck Typing**
    - An ***object's suitability is determined by the methods it supports rather than by its explicit type*** – if an object "quacks" and "waddles" (i.e. has methods like quack() and waddle() that work correctly), we treat it as a duck
    - Instead of checking an object's declared type, just call the methods you want – if they succeed, the object meets our needs (e.g. is considered a duck)
    - **Advantages**: ***Good in exploratory or rapid-prototyping scenarios*** (i.e. ML experiments) that can be rerun in the event of failures
    - **Drawbacks**: ***Less safe*** – lacks the compile-time assurances of static type checking, meaning errors (or even security vulnerabilities) might only be detected at runtime, which is less ideal for production code

Sub-Note 3: Brief Origin of Java

Java was designed to be a simple, portable, and reliable language:
- ***Only supports single inheritance*** – stark contrast with C++, where a class can inherit from multiple superclasses
- Has ***automatic garbage collection*** and ***no pointers*** (although references exist, they have less power than pointers and are safer)
- ***Primitive types (e.g. byte, int, float) are portable*** — in C/C++, the size of these types was machine-dependent, but they ***always have the same size in Java***

Bytecode Execution: Java is designed to compile into bytecode, a platform-independent intermediate language, rather than machine code
- This means Java code can be executed on any machine with a JVM ⇒ portable

HotJava Browser: Written in Java, faster and more reliable than Mosaic (dominant browser at the time) since it could quickly load and execute applets over the Internet

Sub-Note 4: Java Multithreading

Multithreading ***promotes efficient and safe management of concurrent processes accessing shared resources***. As a result, it works great in solving the consumer/producer, dining philosophers, and reader/writer problems:

Generally speaking, we prefer to do multithreading in Java:

1. Memory Management
- Java has ***automatic garbage collection***, reducing the risk of memory leaks and simplifying memory management
- C++ requires ***manual*** memory management, offering more control but increasing bugs

2. Concurrency Support

- Java provides a comprehensive concurrency API, including high-level constructs like thread pools and concurrent collections which make multithreading easier
- C++ offers lower-level threading mechanisms, granting fine-grained control but often needs more complex code to handle synchronization

3. Safety and Error Handling

- Java has strict runtime checks and exception handling to reduce undefined behavior
- C++ places the onus on the programmer, which can lead to buffer overflows

Lecture 9

Sub-Note 1: Comparing Java and C++ (2 Object-Oriented Languages)

1. Design Philosophy

- Java offers a ***higher level of abstraction*** above the machine
    - Allows Java code to be ***portable across platforms***
    - ***More reliable than C++*** (partly because it was initially developed for devices like toasters, and C++ was prone to crashes)
- However, this comes ***at the expense of performance***
- Java provides ***built-in support for concurrency*** (which C/C++ don't really)

2. Bytecodes

- ***Intermediate representation*** Java programs are compiled into
- Executed by the ***JVM***, an ***abstract stack-oriented machine*** that interprets the bytecode (Gives you portability and reliability at the cost of performance)
    - A ***JIT compiler*** is used by the JVM to ***recover this performance loss***:
        - Part of the runtime; ***translates frequently-used bytecodes into machine-code***
- Java gives you ***portability***, ***small executables***, and ***some runtime efficiency*** with the JIT compiler (albeit not as much as C/C++)

3. Single Inheritance

- ***Java supports single inheritance*** which was ***added for simplicity and performance***
- ***A single class can only inherit from one superclass***
- **Syntax**: $class\ C\ extends\ D$
- All classes inherit from the $Object$ class (root node in Java's type tree) – if a class doesn't explicitly extend another class, it implicitly extends $Object$

Case Study 1: Arrays

Java: Arrays are initialized following the syntax $int[]\ a = new\ int[27];$

- All arrays inherit from the $Object$ class. So, they ***reside on the heap*** and are managed by Java's ***automatic garbage collection system***
    - Introduces ***performance overhead*** (arrays are slower in Java vs C++)
    - But, ***more reliable*** (Eliminates errors from manual memory management)
- The ***type of an array*** is ***determined by the data it holds***, not its length
- Doesn't require the size of an array to be known at compile-time; however, ***once an array is created, its size is fixed***
- Unlike C++, Java allows methods to return arrays

Important: Java compiler uses escape analysis to optimize memory usage

- ***If an array is confined to a method and doesn't escape its scope***, the ***JIT compiler may allocate it on the stack*** rather than the heap ⇒ improves performance

Important: An array of objects in Java is actually an ***array of pointers to those objects***

C++: You can declare an array inside a function ⇒ $int\ f()\ \{int\ a[27]...\}$

- a is allocated on the stack and only exists within the scope of the function f
- Can't return an array in a function (***undefined behavior***)

Case Study 2: Abstract Classes

Overriding: A subclass ***provides an implementation for a method already defined*** in the superclass. However, it should align with the original intent of the superclass's method, ensuring the subclass remains a valid subtype.

Exception: An ***abstract class can't be instantiated directly***. They ***MAY contain abstract methods*** – methods declared without an implementation – that ***must be implemented by any concrete subclass***.

Ex. $abstract\ class\ List\ \{$
$int\ l;$
$abstract\ void\ append(Object\ o);$ // abstract method
$int\ length()\ \{\ return\ l;\ \};$ // concrete method
$\}$

Any concrete subclass of $List$ must implement the $append()$ method. If one doesn't provide its own $length()$ method, it just ***inherits the one from*** $List$.

Question: Why declare a class abstract if it doesn't have abstract methods?

- Useful when we want to explicitly prevent a class from being instantiated directly
    - **Ex**. When the class serves as a base for other classes but shouldn't be used to create objects on its own (e.g., a generic $Vehicle$ class)

Attempting to create an object of an abstract class, such as $new\ List()$, throws a ***compile-time error*** since no constructor exists for an abstract class – otherwise, we could call the $append()$ method, but it doesn't have an implementation. However, you ***can create an instance of a concrete subclass***, like so: $List\ l = new\ LinkedList();$

Case Study 3: Interfaces

Interfaces ***define method prototypes without providing implementations***. For example,

- $interface\ X\ \{int\ length();\ void\ append(Object\ o);\}$
- $interface\ Y\ extends\ X(int\ foobaz();\}$

For a class to implement an interface, you need the $implement$ ***keyword***:
$abstract\ class\ List\ implements\ X\ \{...\}$

A class ***can only extend one superclass,*** but it can ***implement multiple interfaces***. In other words, an abstract class is basically an interface combined with a concrete class.

Java also has ***final classes and methods***, which are the ***exact opposite***. Declaring a method as final prevents subclasses from overriding it, and declaring a class as final prevents other classes from extending it.

Question: Why declare methods/classes final?

- One purpose is ***efficiency***
  - Ex. $final\ int\ f(int\ a,\ int\ b)\ \{return\ a\ *\ a\ +\ b;\}$
  - A ***Java compiler can inline the method*** – rather than generating bytecodes to do function calls, it generate bytecodes to do a multiply and add
    - Allows a JIT compiler to be more efficient
  - Not possible without using the $final$ keyword since the object may have been subclassed, and it's possible the subclass redefined f()
- Another purpose is ***trust***
  - $final$ ensures the intended behavior of methods or classes remains unchanged, preventing unintended modifications by subclasses

Sub-Note 2: Object Class

Includes the following methods:

$public\ Object()$ ⇒ ***constructor***

- $Object\ o\ =\ new\ Object()$ creates a new object without specific properties
- Useful as a ***sentinel value*** in data structures (e.g. ***dummy node in LL***)

$public\ boolean\ equals(Object\ obj)$

- $o.equals(o1)$ ⇒ checks if o and $o1$ are equal
- If one isn't defined, it ***defaults to reference/address equality*** (i.e. == operator)
- ***Useful if we want "semantic equality"*** – gives the designer the freedom to decide what equality actually means

$public\ final\ Class\ getClass()$

- ***Returns a*** $Class$ ***obj representing the runtime class*** of the object on which it's called
- $class$ ***is a keyword*** used to define new types while $Class$ ***is an identifier***
- Ex. $List\ l\ =\ new\ LinkedList();\ l.getClass()$ ⇒ returns $LinkedList$
- ***Useful for debugging*** since ***we can ask an object what class it is***
- **Efficiency**: Since objects are internally represented as pointers to pieces of storage (with type and value fields), $getClass()$ ***is essentially a simple pointer dereference***
- **Trust**: Java doesn't allow you to override $getClass()$, so you always get ***reliable results – an object can never "lie" about what type it is***

IMP: The actual signature of $getClass()$ uses ***generics***

- $public\ final\ Class\ <?\ extends\ X>\ getClass()$ where X is the type of o
- Ex. If o is of type $List$, $o.getClass()$ returns $Class<?\ extends\ List>$

$public\ int\ hashCode()$
- Should be consistent with $equals()$ ⇒ ***if*** $equals()$ ***returns true, both objects should have the same hash code***
- Typically defaults to hashing the object's internal address

$public\ String\ toString()$
- ***Returns a string representation*** of the object; can be used for debugging

$protected\ Object\ clone()\ throws\ CloneNotSupportedException$
- Creates and returns a copy of the original object
- Default implementation makes a ***shallow copy*** – copies the object's fields, but if a field is a reference to another object, it ***copies the reference/pointer***
- Only works for classes that implement the $Cloneable$ interface

Sidenote: $protected$ means the ***method isn't meant for ordinary users to invoke*** – instead, you should use copy constructors or factory methods
Important: If a method can throw an exception, its API must declare it
- Informs callers and helps Java do static checking

Sidenote: The $Cloneable$ interface was originally intended as a mixin to indicate that a class supports cloning BUT it doesn't actually declare the $clone()$ method itself
- Can't call $clone()$ on a generic object – a class must implement $Cloneable$ and override $clone()$ to make it available

$protected\ void\ finalize()\ throws\ Throwable$
- Can throw any exception/error
- Hook into the garbage collector – called just before the object is about to be reclaimed
- $finalize()$ should only be used for cleanup of non-Java resources (e.g. closing files) since the ***JVM doesn't guarantee it's ever called on any object***

Sub-Note 3: Multithreading

Java implements threads using the $Thread$ class:
- ***Each*** $Thread$ ***represents a distinct thread of execution*** with its own call stack, program counter, and runtime environment
- By default, the $run()$ method in $Thread$ does nothing; you define a thread's behavior by overriding the $run()$ method

There are ***2 main approaches for defining a thread***:
1. ***Extend the*** $Thread$ ***class*** - create a subclass of $Thread$ and override $run()$
    - $class\ MyThread\ extends\ Thread\ \{...\}$ ⇒ ***too restrictive*** since it ties the class to the $Thread$ hierarchy

2. ***Implement the*** $Runnable$ ***interface*** – define a class that implements $Runnable$ by providing an implementation for $run()$

$$interface\ Runnable\ \{void\ run();\}$$
$$class\ C\ implements\ Runnable\ \{...\ void\ run()\ \{do\ things\}...\}$$
$C\ obj\ =\ new\ C();$ // ***runnable object (code that wants to run)***
$$Thread\ t\ =\ new\ Thread(obj);$$

$Thread$ has a constructor that takes any runnable object and invokes the $run()$ method of it. This ***decouples the thread logic from*** $Thread$, allowing our class to extend another class.

Thread Life Cycle

1. When you create a thread using the new keyword, the thread enters the NEW state
    - The thread is an object that represents a task that will eventually execute
2. Invoking the $start()$ method (e.g. $t.start()$) ***allocates necessary OS resources*** like a virtual CPU, instruction pointer
    - The object is now in the $RUNNABLE$ state (***doesn't mean it's running***, rather that the thread is ready to execute whenever the scheduler allocates CPU time)
3. The thread executes the code in its $run()$ method, still remaining in $RUNNABLE$
4. When $run()$ completes, the thread has no more code to execute, and it enters the $TERMINATED$ state
    - ***While the*** $Thread$ ***object still exists, it can't be restarted***

Common Methods

- $yield()$ ⇒ thread is still in a $RUNNABLE$ state but gives other threads a chance to execute if the system is short on available CPU resources
- $sleep()$ ⇒ thread is put into $TIMED\_WAITING$ state; after sleep duration expires, it becomes $RUNNABLE$ again
- $wait()$ ⇒ thread is put into $WAITING$ state; becomes $RUNNABLE$ again after some event occurs
- I/O ⇒ threads become $BLOCKED$ since they need to wait for data to come in
- ***The OS focuses its scheduling on threads in the*** $RUNNABLE$ ***state***

Imagine the following code:

```
public static void main(String[] args) {
        ... t1.start(); t2.start();
        System.out.println(1/0); // should throw an exception
}
```

The program doesn't stop – we see the exception thrown in the main method but $t1$ and $t2$ keep executing. If $t1$ had a division by 0 exception, $t2$ would still execute – threads are independent.

Race Conditions / Collisions

- Program's behavior depends on the sequence of threads / order of execution
- A classic race condition is as follows:
    - Thread A writes to variable v at the same time thread B reads from it
    - As a result, thread B reads garbage

Solution 1 (Simple): Synchronized Methods

- Ex. $synchronized\ int\ next()\ \{return\ v{+}{+};\}$
- ***Only one thread can execute the method at a time***
- Under the hood, when a thread calls a synchronized method, it first ***acquires the monitor lock*** associated with the object the method belongs to
    - ***No other thread can execute ANY synchronized method on that same object*** until the lock is released

Can cause thread contention, when multiple threads compete for the same lock

- Problematic since blocked threads still hold memory for their stack and state
- The ***real overhead is with context-switching*** – when the OS saves the state of one thread so it can load the state of another
- **Thundering herd problem**: when the lock becomes available, all waiting threads wake up at once and compete to acquire the lock ⇒ can overwhelm the system

Solution 2: Spin-Locks

- Low-level synchronization mechanism where a thread repeatedly checks to see if a lock is available (uses CPU cycles while waiting)
    - ***Used when expected wait time is short*** (e.g. low-level systems programming) where context switching is more costly than spinning
- Although this is ***enough for lighter-duty applications***, there are performance problems – because of the spin-lock, threads waste CPU time and resources doing nothing useful

Note: Every object has a $wait()$ method that can be used for synchronization

- When a thread calls $o.wait()$, it ***releases any locks it holds on object o***, allowing other threads to acquire the lock
- The thread waits until another thread signals it is okay to proceed (through $notify()$), at which point it reacquires the lock(s)

How do you know when the object becomes "available"?

- $o.notify()$ ⇒ wakes up ***one of the threads*** waiting on o
    - **Limitation**: Lets the OS choose which thread to wake up
- $o.notifyAll()$ ⇒ wakes up all the threads waiting on o
    - Lets the threads compete / decide who gets the lock

Solution 3: Semaphore

- Controls access to a limited number of resources through permits
    - ***Doesn't execute tasks***, simply tracks the number of available permits

- $s.acquire()$ ⇒ takes a permit (decrement available count); ***wait*** if none are available
- $s.tryAcquire()$ ⇒ tries to take a permit but ***doesn't wait***; returns true if successful
- $s.release()$ ⇒ returns a permit (increment available count)
- Ex. A ***semaphore with size 10 allows up to 10 threads to acquire a resource***; the 11th must wait until one is released

Solution 4: Exchanger
- ***Synchronization / rendezvous point allowing 2 threads to swap data***
- Let x be the exchanger, then:
    - $x.exchange(v)$ ⇒ Thread 1 offers its value v and waits for a value from Thread 2
    - $x.exchange(a)$ ⇒ Thread 2 offers its value a and the swap occurs
- First thread that calls $exchange()$ waits until the second thread arrives, then they swap

Solution 5: CountDownLatch
- Allows threads to wait until a set of tasks completes
- Initialized with a count representing the number of tasks to wait for
- $countDown()$ ⇒ called by worker threads to decrement the count when a task is done
- $await()$ ⇒ called by waiting threads to block until the count reaches 0
- ***Useful for simple coordination***, like ***waiting for multiple services to start***
- ***One-time use*** – once the count reaches 0, it can't be reset

Solution 6: CyclicBarrier
- Synchronizes a group of threads at a common point, pausing their execution until the target number has arrived
- ***Initialized with the number of threads*** that must reach the barrier
- $await()$ ⇒ each thread calls this upon reaching the barrier, waiting until all threads arrive
    - Once all threads call $await()$, the barrier opens and all threads resume execution
- **Reusable** – after opening, the barrier resets, allowing threads to synchronize again
- ***Useful for more complex coordination***
    - **Ex**. In a weather simulation, each thread calculates conditions for part of the map. Every 20 secs, threads pause at the barrier to exchange data, then repeat

Note: There is a subtle difference between $CyclicBarrier$ and $CountDownLatch$:
- $CyclicBarrier$ ***waits for threads*** – requires ***n threads to break***
- $CountDownLatch$ ***waits for events*** – requires ***n completed tasks to break***, which can be done by any thread

---

Lecture 10

Java Memory Model: Divides memory into 2 main areas
- ***Assume each thread is implemented correctly***

1. Thread Stack

- ***Each thread maintains its own call stack***, which tracks the sequence of method calls leading up to the current execution point
- All ***local variables*** within those method calls are also stored on the thread stack
    - This ***includes primitive variables and references / pointers to objects***
- ***Even if 2 threads execute the same code***, they each have ***separate local variables***

2. Global Heap

- ***All objects*** created in a Java application are stored on the heap
    - Includes objects wrapping primitives (e.g. $Integer$, $Long$)
- ***All member variables of objects (whether primitives or references to other objects) are stored on the heap too***
    - **Exception**: If an object has a method with local variables, those are stored on the corresponding thread stack
- **Shared Access**: Any thread that holds a reference can access the object and its member variables on the heap
- **Static Variables**: ***Only one copy per class***, regardless of how many objects are created ⇒ shared across all instances of a class

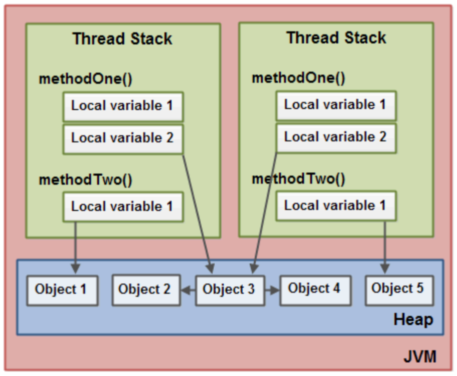
Case Study 1

```java
public class MyRunnable implements Runnable() {

    public void run() {
        methodOne();
    }

    public void methodOne() {
        int localVariable1 = 45;

        MySharedObject localVariable2 =
            MySharedObject.sharedInstance;
        //... do more with local variables.

        methodTwo();
    }

    public void methodTwo() {
        Integer localVariable1 = new Integer(99);
        //... do more with local variable.
    }
}
```

```java
public class MySharedObject {

    //static variable pointing to instance of MySharedObject
    public static final MySharedObject sharedInstance =
        new MySharedObject();

    //member variables pointing to two objects on the heap
    public Integer object2 = new Integer(22);
    public Integer object4 = new Integer(44);

    public long member1 = 12345;
    public long member2 = 67890;
}
```

The code above generates the ***following memory model***:

- Since $sharedInstance$ is a static member, there is only one copy across all instances of $MySharedObject$ (e.g. $Object\ 3$)
- Although $member1$ and $member2$ are primitives, since they are member variables they are stored on the heap



Avoiding Race Conditions in Java

Atomic Operation: Describes an operation that either completes entirely or doesn't
- Ensures no other thread can see the operation in an intermediate state

Visibility: Ensures that when one thread releases a lock, all changes to shared data become visible to any thread that subsequently acquires the lock

1. **Volatile Keyword**
    - "$synchronized\ lite$" – can't do everything $synchronized$ can, but simpler to implement and can be more performant
        - Unlike $synchronized$, volatile variables ***never block threads***, making them also better for ***scalability*** (especially when reads outnumber writes)
    - **Important**: Volatile variables ***provide visibility BUT don't make compound operations atomic***

- ***Happens-Before Guarantee***
    - ***Volatile Write***: Writing to a volatile variable immediately saves its value to main memory; ***all other visible variables in the current thread*** are updated in main memory as well
        - Any write before the volatile write is guaranteed to occur ***before it***
    - ***Volatile Read***: Reading a volatile variable always fetches the value from main memory; also ***refreshes all other visible variables***
        - The volatile read ***happens before any later reads***

Limitations
1. ***Doesn't work for compound operations***
    - Ex. Not strong enough to be used as a thread-safe counter (e.g. $count$++) since incrementing is a compound read-modify-write operation
    - **Exception**: If you can guarantee only one thread updates the counter, $volatile$ keyword is enough
2. Not suitable for classes with ***invariants relating multiple variables*** (e.g. $start <= end$)

Volatile is most often used for simple boolean status flags (indicating an important one-time life-cycle event has happened, e.g. shutdown has been requested)

Sidenote: Imagine the following code: $while\ (o.x == 0)\ \{execute\ code...\}$
- If x was not a volatile variable, the code could be optimized like so:
- $int\ i\ =\ o.x;\ while\ (i == 0)\ \{execute\ code...\}$

Adding $volatile$ puts constraints on optimization and ***may make your program slower*** – this is because the JVM must ensure that reads and writes to the variable occur in the order written in the program ("$happens$-$before$" **guarantee**). Furthermore, the JVM ***can't cache the value of a volatile variable***, further slowing down the program.

2. Synchronized Keyword

Uses ***2 primitives*** "under-the-hood":

- $monitorenter$ – acquires the lock for the object being synchronized on
    - If the lock is available, the thread acquires it and proceeds
    - Otherwise, the thread is blocked / forced to wait
- $monitorexit$ – release the lock when the thread exits the $synchronized$ block

In synchronized code, actions are generally divided into 3 categories:

- A***: normal load / store*** (read from or write to a non-volatile variable)
- B***: volatile load or monitorenter***
- C***: volatile store or monitorexit***

The ***compiler is allowed to reorder instructions for optimization purposes,*** but ***reordering is restricted*** by synchronization rules:

1. A followed by A

- Generally, the compiler can freely swap normal loads and stores (e.g. $store\ x;\ load\ y$)
    - **Exception**: Can't swap $store\ x;\ load\ x;$
        - However, the compiler can eliminate redundant loads (like reloading x immediately after storing it) since the value is already known

2. A followed by B

- $load\ x;\ monitorenter;\ \{critical\ section\ code\}\ monitorexit;$
- The compiler can expand the critical section by moving the $load$ inside it – this is because once the lock is acquired, the load is still valid

3. C followed by A

- $monitorexit;\ store\ z;$
- The compiler can similarly expand the critical section by moving the $store$ inside it

Note: Critical sections ***can be extended, but never reduced***

- **Happens-Before Guarantee**
    - When a thread ***enters*** a synchronized block, ***all variables visible to the thread are refreshed/read*** from main memory
    - When a thread ***exits*** a synchronized block, ***all variables visible to the thread are written back to*** main memory

Sub-Note 2: Logic Programming

Basic entity is a predicate

- Glue together predicates using ***logical operators***: & (and), | (or), and ! (not)
- ***Give up functions and side-effects***
- The goal is to ***think declaratively*** – describe what you want, but not how to get it
- Instead of writing step-by-step instructions, you ***specify the properties of desired answers***, and the system figures out how to compute them

- A core mantra is $algorithm = logic + control$
    - $logic$: defines what constitutes a correct answer (i.e. problem spec)
    - $control$: determines how the system searches for the answer (i.e. efficiency)

Case Study 1: Sorting

$sort(L, S)$ defines a relationship where L is the list to be sorted, and S is its sorted version
- The goal is to specify what sorting means, not how it's done

Basic properties of S compared to L:
1. S is a permutation of L
2. S is sorted

At the top-level, :− ***means "if"*** and , ***means "and"***
- So $sort(L, S)$:− $perm(L, S)$, $sorted(S)$.
- This is equivalent to saying "if S is a permutation of L, and S is sorted, then S is the sorted version of L"
- This can also be written like: $\forall L \forall S\ (sorted(S)\ \wedge\ perm(L, S)\ {->}\ sort(L, S))$

Defining $sorted(S)$
- $sorted$([]). ⇒ the empty list is sorted (fact since it doesn't have a conditional)
    - Equivalent to $sorted$([]) :− $true$ ⇒ rule since it has a conditional
- $sorted$([_]). ⇒ singleton list is sorted
    - _ represents a nameless variable
- $sorted([X, Y|Z])$:− X =< Y, $sorted(Z)$.
    - Z is a list while X and Y are individual elements
    - Z can be the empty list
    - Allows us to sort any list of 2 or more elements

Problem: Lists like [3, 5,-2, 7] are considered sorted since the above only checks elements in pairs. We can address this with the following:

$sorted([X, Y|Z])$:− X =< Y, $sorted([Y|Z])$.
- $[Y|Z]$ is a list with head element Y and tail list Z

Defining $perm(L, S)$
- $perm$([], []). ⇒ permutation of the empty list is itself
- $perm([X], [X])$. ⇒ permutation of a singleton list is also itself
- $perm([X, Y], [X, Y])$.
- $perm([X, Y], [Y, X])$.
- ...

The approach above is way too inefficient and time-consuming. We can introduce the following:

$perm([X|L], R) :- perm(L, PL),$ ⇒ recursively permute the tail list L into PL
$append(P1, P2, PL),$ ⇒ break PL into 2 parts
$append(P1, [X|P2], R).$ ⇒ re-insert X into every possible position to form R
With this, we can remove any $perm()$ rule with lists that have 1 or more element

Defining $append(A, B, C)$
- $append([], L, L).$ ⇒ appending the empty list to L yields L
- $append([X|L], M, [X|LM]):-\ append(L, M, LM).$ ⇒ resulting list must start with X, and the rest of the elements are the recursive result of appending the tail L to M

Note: Prolog processes ***left to right, depth-first recursively***. The time complexity is $O(N!)$ since ***Prolog uses a generate-and-test approach*** – in our case, it first generates a permutation of L, then checks if that permutation is sorted. It also only returns the first match – if you want the next match, type ; $(semicolon)$.

Sub-Note 3: Prolog Syntax

In Prolog, a term is one of the following:

1. **Constant**: Integers, real numbers, or atoms
    - **Atom**: Any name ***starting with a lowercase letter*** (optionally followed by 0 or more additional letters, digits, or underscores)
        - Any string can be an atom if surrounded by single quotes – the string itself doesn't need to be a valid atom name
        - Ex. fred, '*', '[]', 'ABCDEF', 'abc def'
        - ***Two atoms are equal if and only if they have the same name***
2. **Variable**: Any name ***starting with an uppercase letter or underscore*** (optionally followed by 0 or more letters, digits, or underscores)
    - Logical variables ***represent the same value throughout a clause*** (i.e. fact or rule)
    - Become bound to terms on success and unbound on failure
    - Ex. X, Child, _123
3. **Compound Terms**: An atom followed by a ***parenthesized, comma-separated list of terms***
    - **Syntax**: $f(T1, T2 \ldots Tn)$ where f is an atom and $T1, T2 \ldots Tn$ are terms
    - There must be ***at least one argument*** (n > 0)
    - Ex. $x(y, z)$ or $+(1, 2)$ or $pr(3, 9, xz(27))$
    - Alternative notations exist (e.g. $+(1, 2)$ can be written as $1 + 2$), but they ***semantically mean the same thing***

Arity: Number of arguments n
- Last compound term is written as $pr/3$ for a functor pr with arity 3

Internal Representation: Stored as vectors:
- The first slot holds the atom/functor
- The remaining slots hold the arguments (which may point to other data structures)

Although there are technically other symbols (e.g. :–, commas), they are just syntactic sugar for other compound terms:

1. Comparison and arithmetic operators
    - $X =< Y$ is the same as '=<'(X, Y)
    - $3 + 4 * N$ is the same as $+(3, *(4, N))$
2. The empty list [] is the same as the atom '[]'
    - A list like $[X, Y, 3]$ is the compound term ‘.’(X, ‘.’(Y, ‘.’(3, ‘[]’)))
    - $[X|Y]$ is the same as ‘.’(X, Y)
3. X :– Y, Z is the same as ':-'(X, ','(Y, Z))

Lecture 11

Sub-Note 1: Clauses

Clause: A logical statement about the world that’s universally quantified
- Either a ***rule, fact, or query***
- Ex. $ab(X, f(Y), Z)$:– $bc(Z, f(Y), X)$
    - **Meaning**: For all X, Y, and Z, if $bc(Z, f(Y), X)$ then $ab(X, f(Y), Z)$
- **Note**: $ab(A, B, C)$:– $bc(C, B, A)$ is simpler but more general – doesn’t require the second argument to be of the form $f(Y)$

Substitution: ***Set of assignments to logical variables***; e.g. $\{A = X, B = f(Y), C = Z\}$
- Applying a substitution to a more general clause can produce a more specific clause ⇒ helps to prove if one clause is more general than another
- The reverse doesn’t work ⇒ $\{X = A, f(Y) = B, Z = C\}$ is invalid since the ***LHS must be a single logical variable***, which $f(Y)$ doesn’t match

3 Kinds of Clauses
1. **Facts**: Don’t have a condition (e.g. :–)
    - **Ground Terms**: Terms with no logical variables
        - $pr(cs31, cs131)$ ⇒ $cs31$ is a prereq for $cs131$ (atoms)
    - ***Can also involve logical variables***
        - $pr(intro1, \_)$ ⇒ $intro1$ is a prereq for all courses
2. **Rules**: Have condition(s)
    - $prt(A, B)$:– $pr(A, B)$
    - $prt(A, Z)$:– $pr(A, B)$, $prt(B, Z)$ ⇒ for all A, B, Z, if both conditions on the RHS are true, then the LHS is also true
3. **Queries**: Prompt the interpreter to search for answers
    - Prolog searches for matches in a ***depth-first, left-to-right manner*** to find the first match (i.e. in the order the facts/rules are listed)

- If you type a ***semicolon (;)***, Prolog returns the ***next match*** (if one exists)
- Under the hood, Prolog ***uses proof by contradiction*** – it assumes your query is false and looks for a counterexample
- **Ex**. $?- prt(cs31, R)$ tells Prolog to prove there is no R such that $prt(cs31, R)$ is true, but if Prolog finds an R, it returns it as a counterexample

Sidenote: Prolog is ***very gullible*** – it ***takes facts and rules at face-value.*** So, it's up to the programmer to ensure logical correctness.

Sidenote: In GProlog, ***cyclic queries*** like $f(X) = X$ or $X = f(X)$ ***terminate immediately and return "yes"*** since Prolog detects the cycle and unifies X with a cyclic term. Meanwhile, ***cyclic facts or rules*** like $f(X) :- f(X)$ will ***compile successfully***, but invoking them directly through queries like $f(Y)$ result in ***infinite recursion***.

Sidenote: Atoms are only equal to themselves – $a = a$ returns "yes" while $a = 1$ returns "no". A query like $X = f(a)$ assigns $f(a)$ to the variable X.

Case Study 1: Member Predicate

We want to implement a predicate that tests whether an element appears in a list. The ***functor is*** $member$ and its ***arity is 2***:
- $member(X, [X|\_])$.
- $member(X, [\_|L] :- member(X, L)$.

A query like $?- member(a, [a, b, c])$. would ***return "true" or "yes"***
- This is because it ***unifies with the fact*** $member(X, [X|\_])$.
- Typing ***";"*** would ***return "false" or "no"*** since there's no other occurrence of a in the given list (Prolog explored the entire set of clauses and couldn't find another answer)

A query like $?- member(Q, [a, b, c])$. will ***return*** a***, then*** b***, then*** c***, then "no"***
- **Reasoning**: For the first match, Prolog internally unifies $X = Q$ and $L = [b, c]$, meaning X is a. Then, it tries to prove the subgoal $member(Q, [b, c])$, repeating the process

What about the query $?- member(Q, R)$.
- First returns $R = [Q|\_123]$ where _123 ***indicates a placeholder logical variable*** (can represent any atom) and Q represents the first member
- If you type ***";"***, Prolog backtracks and returns $R = [\_123, Q|\_124]$ where Q represents the second member and so on ***(repeats infinitely)***

Case Study 2: Append Predicate (Review)

We want to implement a predicate to append an element to a list:
- $append([], L, L)$. ⇒ ***base case***: appending the empty list to L yields L

- $append([X|L], M, [X|LM]){:-}\ append(L, M, LM).$ ⇒ result starts with X, and the tail is the result of appending L to M

A query like $?-\ append([a, b], [c], R)$ returns $[a, b, c]$; ***under the hood***:
- Prolog matches with the rule: $append([X|L], M, [X|LM]){:-}\ append(L, M, LM).$
    - $X = a, L = [b], M = [c], and\ R = [a|LM]$
- The next subgoal is $append([b], [c], LM)$
    - Prolog matches with the same rule: $X = b, L = [], M = [c], and\ LM = [b|LM2]$
- The next subgoal is $append([], [c], LM2)$
    - Prolog matches with the base case: $LM2 = [c]$
- Then, Prolog backtracks – $LM = [b, c]$ and $R = [a, b, c]$

Note: We can use the $append$ predicate to take apart 2 lists. If we feed Prolog $?-$ $append(R, S, [a, b, c])$, it returns R and S:
- The first match is $R = [], S = [a, b, c]$; then the next match is $R = [a], S = [b, c]$...

Note: If we ask the query $?-\ append(R, S, T)$
- The first match is $R = [], S = T$; then $R = [\_123]$ and $T = [\_123|S]$

Case Study 3: Reverse Predicate

Base Case: $reverse([], []).$
$reverse([X|L], R) :-\ reverse(L, Acc),\ append(Acc, [X], R).$
- $[X|L]$ means the list has a head element X and tail list L (could be empty)
- R is just $[L|X]$, so Acc represents the reverse of the tail list

This implementation has a problem – it is really slow. At the top-level, it runs through the list n times, and each $append$ call is proportional to the length of Acc. To solve this, use a helper predicate and an accumulator.

$rev(L, A, R)$ is true if reversing L and appending A generates R
$rev([], A, A).$ ⇒ base case
$rev([X|L], A, R) :-\ rev(L, [X|A], R).$
$reverse(L, R) :-\ rev(L, [], R).$

Sub-Note 3: Simple Prolog Predicates

1. $fail/\emptyset$
    - Built-in predicate that always fails ⇒ $?-\ fail$ prints “no”
    - Useful for backtracking, e.g. ***printing all matches of a query without manual “;”***
        - Ex. $member(X, [a, b, c]), write(X), fail.$ prints “abc***false***”
2. $true/\emptyset$
    - Built-in predicate that always succeeds ⇒ $?-\ true$ prints “yes”
    - Has 1 fact which is “true”; can be used as a fallback condition in conditional logic

        - Ex. $(X > 10 -> write('Greater');\ true)$.

3. $loop/\emptyset$
    - Has 1 rule: $loop :- loop$. which recurses infinitely
    - Could be used to continuously prompt for user input until an exit condition is met

Ex. $loop :- write('Enter\ a\ number\ (type\ "exit"\ to\ quit):\ ')$,
$read(Input)$,
$<perform\ operations>$
$(Input\ =\ exit\ ->\ true;\ loop)$

4. $repeat/\emptyset$

$repeat.$ ⇒ succeeds once
$repeat :- repeat$ ⇒ succeeds infinitely by calling itself

Note: A query like $?-\ repeat,\ write('ouch'),\ fail$ will print "ouch" repeatedly. $repeat$ always succeeds and $fail$ forces backtracking, so Prolog backtracks to $repeat :- repeat$, causing the process to repeat endlessly.

Note: If we swapped the order of the 2 clauses, $?-\ repeat.$ would result in an ***infinite loop*** since Prolog gets stuck in the recursive rule first.

Lecture 12

What can go wrong with logic programming?

1. **Logic Errors**: Writing flawed logic results in a wrong program
2. **Cyclic Data Structures**: Unification without the occurs check can create infinite loops
3. Control that goes wrong

Debugging in Prolog

- **Trace Facility**: Prolog's ***built-in debugger*** is invoked with the $trace$ predicate
- Each goal is managed through ***4 ports***:
    - **Call**: When a goal is invoked
    - **Exit**: When a goal succeeds
    - **Redo**: When the backtracking re-enters a goal after a failure in subsequent goals
    - **Fail**: The goal itself fails
- Prolog ***sets checkpoints at each port***, dumping information when you cross them

For example, in the query $?-\ p(X),\ q(X,Y),\ r(Z,Z,Y)$. the goals are executed sequentially (i.e. if the first goal succeeds, execute the next). If q or r fails, Prolog backtracks to reattempt previous goals through the $Redo$ port.

This is different from languages like C++, where calling a function leads to a single return path – there is no notion of backtracking at the top-level. ***In Prolog, predicates can succeed more than one time, allowing for automatic backtracking*** to explore different solutions.

Sub-Note 2: Unification

Definition: Process of ***unifying 2 terms*** so they become ***identical***; ***done through substitution***

- When you match a goal against the head of a clause, you are actually unifying it
- **Ex**. Say you have the query $?- member(X, [Y, Z, a])$.
    - Say you also have the clause $member(A, [\_2|L]) :- member(A, L)$.
    - Doing the substitution $\{X = A, Y = \_2, L = [Z, a]\}$ unifies them

Note: Prolog ***unification*** is ***more powerful*** than OCaml's ***pattern-matching***. This is because it is ***2-way instead of 1-way***. You can both deconstruct and generate data, enabling you to write more general rules and automatically explore multiple solutions through backtracking. But, this inadvertently leads to problems:

Case Study 1: Cyclic Data Structures

Suppose you have the clause $p(X, X)$. A query like $p(a, Y)$ works perfectly fine – it binds $X = a$ and $Y = a$. A more complicated query $p(f(X, g(Y)),\ f(h(Z), W))$. would also work – it binds $X = h(Z)$ and $W = g(Y)$. However, the ***following query is problematic***:

$$p(Z, f(Z))$$

We can't arrange for Z and $f(Z)$ to be the same – Prolog says $Z = f(f(f(\ldots$, or in other words, it binds Z directly to the term $f(Z)$ which creates a cyclic data structure.

Ex. Peano Arithmetic in Prolog

Let's denote:

- 0 is represented by the atom z
- $s(X)$ is $X + 1$ (successor function)
    - Ex. $s(s(z)) = 2$

To represent addition:
$plus(z, X, X)$.
$plus(s(X), Y, s(XplusY)) :- plus(X, Y, XplusY)$.
Ex. Now we can type the query $plus(s(s(z)), s(s(z)), R)$ and the interpreter returns $R = 4$

To represent subtraction:
$minus(X, Y, Z) :- plus(Y, Z, X).\ \%\, X - Y = Z$
Ex. Now we can type the query $minus(s(s(s(z))), s(z), R)$ and the interpreter returns $R = 2$

To represent lt (i.e. less than):
$lt(z, s(\_)).$ % 0 $is\ less\ than\ the\ successor\ of\ anything$
$lt(X, s(X)).$
$lt(X, s(Y)) :- lt(X, Y).$

Problem: Running the query $lt(X, X)$. leads to an infinite loop. It unifies X with $s(X)$, but that means Prolog must make X equal to a term that contains X itself. Luckily, Prolog offers a safer, built-in unification predicate called $unify\_with\_occurs\_check/2$:

This predicate ***performs an "occurs check" – if unifying a variable with a term results in a cyclic reference, the unification fails***. This allows us to simplify the definition of lt:
$lt(z, s(\_)).$
$lt(X, s(Y)) :- unify\_with\_occurs\_check(X, Y).$

Now, Prolog checks whether X occurs within Y. If it does, the unification fails. This option is ***safer but slower***. The default unification just sets a pointer – or $O(min(|X|, |Y|))$. The occurs check has to examine the entire structure, making it $O(max(|X|, |Y|))$. ***Prolog chooses to implement the default unification for efficiency reasons***.

Sub-Note 3: Control that Goes Wrong

Case Study 1: Cut Operator (!)

Purpose: Used to freeze some of the choices made by the interpreter
- ***Always succeeds***, so it's used to ***prune the search space*** and ***improve efficiency***

Example: Consider the following clauses for p/1 (functor p with arity 1)
$p(a).$
$p(b)$:- !.
$p(c).$

For the query $p(X)$, the system returns $X = a$ then $X = b$ (third clause isn't tried).
For the query $p(X)$, !, $p(Y)$, the system returns $X = a, Y = a$ then $X = a, Y = b$.

Sidenote: The parent goal of a ! is the head of the clause in which it appears (here, $p(b)$). When the cut is reached, it freezes the choices made between the moment the parent goal was called and the point where the ! was encountered. ***Choices made before the parent goal or after the cut remain open for backtracking***.

Example: Consider the following clauses for sign/2 (functor sign with arity 2)
$sign(X, negative) :- X < 0.$
$sign(X, zero) :- X == 0.$

$sign(X, positive) :- X > 0.$

A query like $p(\text{-}1,\ Y)$ correctly returns $Y = negative$, but ***Prolog still attempts the other rules*** even though logically, only one of these conditions can be true.

Green Cuts (Safe)

- ***Don't change*** the declarative meaning of the program ***(***i.e. the program's output remains the same with or without them)
- $sign(X, negative) :- X < 0,\ !.\ \%\ if\ the\ 1st\ rule\ succeeds,\ don't\ try\ subsequent\ ones$
- $sign(X, zero) :- X == 0,\ !.\ \%\ if\ the\ 1st\ fails\ and\ 2nd\ rule\ succeeds,\ don't\ try\ the\ 3rd$
- $sign(X, positive) :- X > 0.$

Red Cuts (Unsafe)

- Cuts that ***change the meaning*** of the program when introduced/removed
- $sign(X, negative) :- X < 0,\ !.$
- $sign(X, zero) :- X =< 0,\ !.$
- $sign(X, positive) :- X > 0.$
- Although this program is correct, if we remove the cuts and run the query $sign(\text{-}1,\ Y)$, the system returns $Y = negative$ then $Y = zero$ (***incorrect***)

Aside: Cuts can be used to define the negation predicate / unary operator (\+)

- $neg(G) :- call(G),\ !,\ fail.$
- $neg(\_) :- true.$
- If G can be proven using the built-in predicate $call(G)$, freeze the search and return false ($fail$); otherwise return true

Eggert Notes on Cuts (!)

By default, Prolog uses a depth-first, left-to-right search strategy to explore its proof tree (a structure of AND/OR nodes representing every possible way to prove your query)

- This is inefficient when Prolog ***backtracks through unnecessary alternatives***
    - **Solution**: If you know that a particular branch of the proof tree has ***no answer,*** you can ***use a cut (!) to stop exploring that subtree***

Example: Consider the following predicate $p(X, Z) :- q(X, Y),\ member(X, Z),\ r(X, Y, Z).$

- Assume that q generates many alternatives and r is expensive to compute
- We want $member/2$ to prune some alternatives and make the program more efficient
- **Problem**: If Z contains duplicates (e.g. $Z = [a, b, r, a, c, a, d, a]$), $member/2$ will repeat work for the same element even if r has already previously failed for it
    - **Ex**. If $X = a$, we make the same redundant call to r 4 times

Solution: Use a modified version called $member\_chk/2$ that succeeds only once per element:

- $member\_chk(X, [X|\_]) :- !.$

- $member\_chk(X, [\_|L]) :- member\_chk(X, L).$
- The cut (!) ensures that once a match is found, no alternative solutions are considered for the current call

Example: The following predicate $once/1$ is defined as $once(P)$:− P, !.
- $once(P)$ executes P and upon the first success, the cut (!) prevents any further backtracking, guaranteeing P only succeeds once
- Calling $once(member(X, Z))$ is the same as $member\_chk(X, Z)$

Sidenote: Imagine the predicate $bonce(P)$:− !, P. If P fails or you attempt to backtrack, Prolog doesn't try alternative solutions for P. In other words, $bonce(P)$ is essentially just the same as calling P directly.

Sub-Note 4: Theory and Logic

|− P ⇒ P is provable (a slash means P is not provable)
|= P ⇒ P is true (a slash means P is not true)

Problem: Ideally, we want |− P to be identical to |= P
- In Prolog, the operator \+ P ***more accurately means "P is not provable"***
  - If Prolog can't prove P, then \+ P succeeds
  - ***However, the inability to prove P doesn't mean that P is false*** – it may just be that our knowledge base isn't expansive enough

Prolog gets around this by operating under ***CWA (close world assumption) – anything you explicitly state is true, and anything you don't is false***.

Ex. Consider the query $X = 5$, \+$(X = 6)$.
- This binds X to 5, then asks "can you prove that 5 equals 6?"
  - Since 5 is not equal to 6, Prolog can't prove it, so the query succeeds with $X = 5$
- If you reverse the order:
  - Prolog asks "can you prove that X equals 6?"
    - Prolog binds X to 6 but then triggers a ! followed by a failure
- ***Once you introduce cuts or negation, the AND operator no longer is commutative***

2 Main Parts of Logic
1. **Propositional Logic**
   - Deals with propositions that are about the state of the world
   - Ex. $p = "it's\ raining"$
   - Ex. $q = "there's\ a\ demo\ on\ Portola"$
2. **Connective Logic**
   - Let us build larger logical statements out of smaller ones
   - The simplest connective is the negation $\neg p$
   - p, q ⇒ means ***p and q***

- p->q ⇒ means ***p implies q*** (true unless p is true and q is false)
- $p{:}-q$ ⇒ means the reverse of the above (if q is true, then p must be true)

In logic, propositions let us prove statements without knowing their exact meaning. For example, ***if*** p ***implies*** q ***and*** q ***implies*** p***, then*** p ***is true if and only if*** q ***is true*** – a ***tautology*** (always true regardless of what p and q represent).

However, propositional logic struggles with more complex reasoning. Consider the following syllogism by Socrates:

- All men are mortal (p)
- Socrates is a man (q)
- Therefore, Socrates is mortal (r)

In propositional logic, this means $p, q{-}{>}r$. This is just an implication, not a tautology – the conclusion also depends on the content of p, q, r, not just the form. ***First-order logic fixes this by adding logical variables, quantifiers*** (e.g. "for all" ∀ and "there exists" ∃), ***and predicates*** (propositions with arguments).

The syllogism thus becomes the first-order tautology $\forall X, man(X){-}{>}mortal(X)$. Prolog uses first-order logic as well. It works by first converting a question into a series of clauses:

$$C_1 \vee \ldots C_n <- A_1 \wedge \ldots A_m$$

To keep this efficient, ***Prolog uses Horn clauses*** – a restricted form where $n \leq 1$. More specifically:

- If $n = 1, m = 0$ ⇒ Prolog ***fact(s)***
- If $n = 1, m > 0$ ⇒ Prolog ***rule(s)***
- If $n = 0$ ⇒ Prolog ***query(s)***

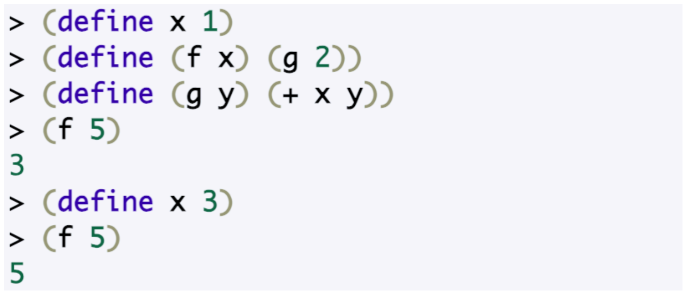
By limiting itself to Horn clauses, Prolog avoids the complexity of full logic systems, focusing on goal-directed reasoning – starting from a query and working backwards through rules and facts to either prove or disprove it.

Lecture 13

Sub-Note 1: Characteristics of Scheme-like Languages

1. **Simple Syntax**
    - Programs are written almost entirely using parentheses with minimal syntax
    - ***Programs are straightforwardly represented as data***
        - While we can represent a C++ program as a string, it is really complicated to turn that string into a program
        - In Scheme, if we have a piece of data, we can just go run it
    - Unlike C++

2. **Memory Management**
    - ***Objects are allocated dynamically and are never freed*** (Scheme garbage collector automatically reclaims unused memory)
    - Unlike C++; like Java, Python, ML
3. **Dynamic Type Checking**
    - Types are checked at ***runtime***
    - Unlike C++, Java, and ML; like Python
4. **Static (Lexical) Scoping**
    - A variable's binding is determined by its position in the source code – not by the order of function calls
        - Each variable is bound in the block where it's defined, and inner blocks inherit these bindings unless they declare their own version
        - Functions "remember" the environment where they were defined and use the variable bindings from that site, regardless of where they are called
    - Unlike Lisp, sh; like Java, Python, ML, C++



```
> (define x 1)
> (define (f x) (g 2))
> (define (g y) (+ x y))
> (f 5)
3
> (define x 3)
> (f 5)
5
```

- Since g is defined in the global scope, it captures the global variable x
- When f is called with $(f\ 5)$, it creates a different local parameter x with value 5
    - Although f calls g, g was still globally defined
- When f calls $g(2)$, the ***function*** g ***looks up the current value of the global*** x ***at execution time - not the value when g was defined***
    - This is ***VERY different from OCaml*** – the ***value of a variable is captured at function definition time***
- However, if the definition of g was inside f, both calls to f would print 7
- ***A function uses variable bindings from the scope where it was defined, NOT where it was called***

All identifiers are resolved at compile-time by examining the nearest enclosing binding, making ***code behavior predictable*** and ***easier to debug***. In contrast, ***dynamic scoping*** resolves variables at runtime by searching the call stack, which ***adds runtime overhead*** and ***reduces software reliability***. However, dynamic scoping can be useful for quickly passing implicit context or state through a series of function calls without explicitly threading parameters.

$(define\ y\ 12)$
$(define\ (f\ x)$
$\quad (-\ x\ y))\ //\ y\ is\ resolved\ from\ the\ global\ environment$

```
(f 13) // prints 13 − 12 = 1
(let ((y 19))
        (f 3)) // prints 3 − 12 = -9
```

Case Study

define

- Used to define a new variable or function
- Redefining a variable in the same scope with *define* ***usually results in an error (most REPLs allow you to though)***, but you can use *define* in different scopes
    - REPLs ***allow this flexibility*** since they're ***designed for interactive experimentation*** - prioritize usability over strict adherence to language rules

set!

- Used to mutate the value of an already defined variable
- Can’t be used to define a new variable

let

- Used to create local variables within a block

5. **Call by Value**
    - ***Arguments are fully evaluated before the function is invoked***; copies of them are then passed to the function
    - Unlike C++ (Call by reference); like ML, Java, Python, C++ (mostly)
6. **Objects**
    - Also include procedures/functions (specifically continuations – low-level, dangerous procedures)
    - Just like any other object – could have a list of procedures
7. **High-Level Arithmetic**
    - Ex. Floats and integers have infinite width
8. **TRO (Tail Recursion Optimization)**

Suppose we have the following code for a recursive factorial function in C++:

```
int factorial(int n) {
        if (n == 0)
                return 1;
        return n * factorial(n − 1);
}
```

In Scheme, this might be written as:

```
(define (factorial n)
        (if (zero? n)
                1
        (* (factorial (− n 1) n))))
```

Sidenote: In Scheme, * ***and*** − ***are functions, not operators***. On a stack-oriented machine, every function call creates a new stack frame that holds local variables and state; these frames are removed when the function returns. Also, recall that ***Scheme evaluates function arguments before the function is called***.

For example, the call $(zero?\ n)$ creates a stack frame to test if n is 0, and its frame is removed once it returns. Note that the recursive call to $factorial$ is ***not in tail position*** – we ***still need to call the*** * ***function after it returns***. As a result, each recursive $factorial$ call must wait on the stack until the multiplication is complete, creating many stack frames (e.g. 100 for n = 100).

Scheme requires tail call optimization (TCO). A tail call is when a function calls another as its very last action, immediately returning that call's result without any further computation. With TCO, the ***current stack frame can be replaced by the new call's frame, preventing stack growth***. We can rewrite $factorial$ so that the recursive call is the tail call:

```
(define (fact n acc) ; 'acc' is an accumulator storing the product so far
        (if (zero? n) ; base case: if n is 0, return the accumulated product
                acc
        (fact (− n 1) (* acc n)))) ; evaluate * before recursive call

(define (factorial n)
        (fact n 1))
```

Under the hood, ***Scheme optimizes most tail-recursive functions by turning them to loops***:

1. **Ordinary** 'Let': A let expression evaluates its bindings before using them

```
(let ((x (+ 9 3 -7))) ; evaluate the expression and bind 5 to x
        (* x (+ x 3))) ; use x's value (5) in the multiplication
```

2. **Named** 'Let': Essentially a local function that allows recursion (acts as a loop)

```
(define (factorial n)
        (let fact ((n n) (acc 1) ; 'fact' is the recursive function
                                 ; first 'n' is the function parameter
                                 ; second 'n' is the input passed into factorial
                                 ; 'acc' starts at 1
                (if (zero? n)
                        acc
                (fact (− n 1) (* acc n)))))
```

If we're just writing loops in disguise, why not just use while and for loops? The answer is because we're applying the same idea from FP – calling functions without side effects, and not modifying variables (so we don't lose track of their values). We want to take this idea and make it easy to write loops if you want to, but if you prefer, you can just write it recursively.

Sub-Note 2: Scheme Syntax

1. Identifiers: Can include the following characters – $\{a\text{-}zA\text{-}Z0\text{-}9\}+-.?*/<=>:\$\%\hat{}\&\_\sim@$
- * is considered a valid identifier ⇒ $(let\ ((*\ \ 5))\ (-\ *\ 3))$ resolves to $* = 2$
- **Exception**: Identifiers can't start with one of the following characters $[0\text{-}9+-.]$
    - Except the following are valid identifiers: $+, -, \ldots\ (3\ actual\ dots)$
    - $->\ldots$ (e.g. ->xyz) is also a valid identifier since a common naming convention for a function converting values of type a to b is $a->b$

2. **Comments**: Begin with a semicolon (;)
3. **Lists**: Consists of pairs; ***each pair contains a pointer to the next element***
    - **Proper List**: Sequence of pairs that ***ends with the empty list*** $()$
        - Ex. $(list\ 1\ 2\ 3)$ or $'(1\ 2\ 3)$ are equivalent to $(cons\ 1\ (cons\ 2\ (cons\ 3\ ())))$
    - **Improper List**: Sequence of pairs that ***doesn't end with the empty list***; instead, the final pair's cdr points to a non-list value
        - Ex. $'(1\ 2\ .\ 3)$ is represented as $(cons\ 1\ (cons\ 2\ 3))$
4. **Boolean**
    - $\#t$ represents true and $\#f$ represents false
    - In Scheme, only $\#f$ is considered false, ***every other value evaluates to true***
    - Ex. $(if\ 0\ 5\ 10)$ yields 5 in Scheme since 0 is false
        - In C/C++, $0\ ?\ 5:\ 10$ would yield 10
5. **Vector**
    - $\#(a\ v\ e\ c\ t\ o\ r)$
    - ***More space-efficient*** than lists of the same length; allow constant-time access to elements unlike lists
6. **Strings**
    - Represented with double quotes; same syntax as in C++
    - Ex. $"hello"$
7. **Characters**
    - Ex. $\#\backslash c$ represents the character c
8. **Numbers**
    - Ex. $1e9$, 12, -19, and $2/3$ (represents the exact fraction $2/3$ without rounding)

Case Study 1: Quoting
- Putting an apostrophe ' before an expression E tells Scheme to ***treat that expression as literal data***, not to evaluate it
- Ex. $'E$ is equivalent to $(quote\ E)$

Special Form: Expressions that look like function calls – $(f\ \ldots)$ – but are actually special forms

- $Ex.\ define$, let, $lambda$, and if are special forms
- In an expression like $(not\ (<\ 3\ 5))$, both not and $<$ are functions because not can be defined in terms of simpler operations, unlike if

Sidenote: $(list\ a\ b\ (+\ c\ 5))$ evaluates $c\ +\ 5$ first and includes its result in the list
- However, $'(a\ b\ (+\ c\ 5))$ keeps the expression $(+\ c\ 5)$ unevaluated
- This ***also applies to nested lists***:
    - $(list\ 1\ 2\ (+\ 1\ (+\ 1\ 1)))$ evaluates to $(1\ 2\ 3)$
    - $'(1\ 2\ (+\ 1\ (+\ 1\ 1)))$ evaluates to $(1\ 2\ (+\ 1\ (+\ 1\ 1)))$

Case Study 2: Quasiquote
- Uses the backtick ` (NOT to be confused with the apostrophe ')
- ***Any element preceded by a comma is evaluated***
- Ex. $`(a\ ,(+\ 1\ 2)\ (+\ c\ 5))$
    - The second element evaluates to 3, so this yields $(a\ 3\ (+\ c\ 5))$
- The ***comma applies to nested expressions*** **–** $`(1\ 2\ ,\ (+\ 1\ (+\ 1\ 1)))$ yields $(1\ 2\ 3)$

Case Study 3: Lambda Expression
- ***Special form*** that creates an ***anonymous (nameless) function***
- **Ex**. $(lambda\ (x)\ (+\ x\ 1))$ defines a function that increments its arguments by 1
- Can also ***nest lambdas*** to create functions that return other functions:
    - **Ex**. $(lambda\ (x)\ (lambda\ (y)\ (+\ x\ y)))$
    - Outer lambda takes x and returns a new function that takes in y and adds it to x
    - This allows us to curry functions:
      $(define\ g\ ((lambda\ (x)\ (lambda\ (y)\ (+\ x\ y)))\ 3))$ and calling $(g\ 7)$ yields 10

Note: $(define\ (f\ x\ y)\ (+\ x\ (*\ y\ 3)))$ is equivalent to $(define\ f\ (lambda\ x\ y)\ (+\ x\ (*\ y\ 3)))$

Sidenote: Variadic Functions
- Functions that can accept a ***variable number of arguments*** (0 or more)
- Can be defined in one of 2 ways:
    - $(define\ (myfunc\ .\ args)\ <function\ body>)$
        - Call it like $(myfunc\ 1\ 2\ 3\ 4)$
    - $((lambda\ args\ <function\ body>)\ <insert\ args>)$
- To pass a list of arguments, use the $apply$ function:
    - Ex. $(apply\ myfunc\ '(1\ 2\ 3\ 4))$ ⇒ calls $myfunc$ with the arguments $1\ 2\ 3\ 4$
    - Ex. $(apply\ (lambda\ args\ (display\ args))\ '(1\ 2\ 3\ 4))$

Ex. $(define\ (average\ x\ .\ nums)\ //\ takes\ at\ least\ 1\ argument$
$(/\ (apply\ +\ x\ nums)\ (+\ 1\ (length\ nums))))$

In the above, the first argument is bound to x and the rest are collected into the list $nums$. The $apply$ function takes a procedure, any number of regular arguments, and a list of remaining elements. For example, $(apply\ +\ 1\ '(2\ 3))$ is equivalent to $(+\ 1\ 2\ 3)$. So, $(average\ 1)$ returns 1 while $(average\ 1\ 2\ 3)$ returns 2.

Case Study 4: Let Scoping

```
(define x 10)
(define a 3)
(let ((x (+ a 3)) // new local x: computed as 3 + 3 = 6
      (y (* x 2))) // x refers to the global binding, so y = 10 * 2 = 20
         (+ (* x x) (- y x))) // x refers to the local x; returns 36 + (20 - 6) = 50
```

In a let expression, all initializers are evaluated in the outer scope. For $(y\ (*\ x\ 2))$, the x is taken from the outer scope (value 10), not the new local x. Once the initializers are computed, the new bindings (x and y) become ***available only in the body of the*** let. ***Every occurrence of x or y within the body uses these new local bindings.***

The let is syntactic sugar. The example above is equivalent to:

```
((lambda (x y)
         (+ (* x x) (- y x)))
         (+ a 3) (* x 2)) // line represents the 2 arguments being passed to the lambda
```

Aside: More Special Forms

1. $and \Rightarrow (and\ E1, E2, \ldots En)$
   - If any expression evaluates to $\#f$, return $\#f$
   - If all expressions are true, ***return the value of the last expression*** – not necessarily $\#t$
2. $or \Rightarrow (or\ E1, E2, \ldots En)$
   - Returns the value of the first expression that isn't $\#f$
   - If every expression evaluates to $\#f$, return $\#f$

Sub-Note 3: Tail Recursion

Even if the recursive call appears within a branch (e.g. in an if), ***if no further computation is performed after the call, it is considered a tail call***.

```
(define (f n)
         (if (zero? n)
                  (f x) // tail call: nothing is done after this call
                  37))
```

Tail Call Contexts

- If Expressions: $(if\ E1\ E2\ E3)$ // $if\ E1, then\ E2, else\ E3$

    - When an entire if expression is in tail position, both the $then$ and $else$ parts are considered tail calls
- Special Forms like and / or: $(and\ E1, E2, \ldots En)$
    - If an and / or expression is in tail position, only the last subexpression En is a tail call because the overall form commits to returning that final value

Scheme lets you create your own special forms (similar to C++ macros) using the $define$-$syntax$ keyword. Here are 2 examples:

```
(define-syntax and
        (syntax-rules ()
                ((and) #t) // the 'and' of nothing is #t
                ((and x) x)
                ((and x y ...)
                        (if x (and y ...) #f)))) // if x is true, check the rest; otherwise return #f
```

The expression $(and\ (<\ x\ 1)\ (<\ y\ 3)\ (p\ x))$ expands to $(if\ (<\ x\ 1)\ (if\ (<\ y\ 3)\ (p\ x)\ \#f)\ \#f)$

```
(define-syntax or
        (syntax-rules ()
                ((or) #f) // the 'or' of nothing is #f
                ((or x) x)
                ((or x y ...)
                        (let ((xc x)) // evaluate x once, store it in xc
                                (if xc xc (or y...))))))
```

The let ***binding (***xc***) ensures that*** x ***is evaluated only once***. If or was defined similarly to and, x would be evaluated twice, leading to potential unintended side effects. Additionally, ***Scheme's*** $define$-$syntax$ ***system is hygienic***, meaning that any identifiers introduced inside the macro are automatically renamed to avoid collisions with identifiers in the surrounding code. This ***ensures referential transparency***.

Lecture 14

Sub-Note 1: Categorizing Scheme

1. **Primitives vs Library**
    - **Primitives**: Built-in operations like $(if\ \ldots)$ and $(lambda\ \ldots)$ – can't be defined in terms of simpler components (should be minimal but functional)
    - **Library**: Defined using primitives, like $(not\ \ldots)$ and $(and\ \ldots)$
2. **Required Features vs Optional Features vs Extensions**
    - **Required**: ***Built-in*** to every Scheme implementation (e.g. integer arithmetic)
    - **Optional**: Provided at the implementer's discretion (e.g. multiple-precision FP)

- **Extensions**: Implementation-specific features, not standardized
- Using ***optional features or extensions reduces portability***

Sub-Note 2: Mistakes and Bugs in Scheme

1. **Implementation Restrictions**: Limits imposed by the system, e.g. available memory
2. **Unspecified Behavior**: Behavior depends on the implementation
    - **Ex**. $(eq?\ 0\ (-\ 2\ 2))$
        - Some Scheme implementations optimize numbers st. all instances of 0 are the same object in memory while some don't

Aside: Equality Predicates

1. $eq?$: $O(1)$ ***pointer*** comparison – checks if 2 references point to the same object
2. $eqv?$: $O(n)$ ***shallow*** comparison – compares contents of simple values (e.g. numbers, chars)
- ***Non-recursive***, n is the size of the object

3. $equal?$: ***deep*** comparison
- ***Recursively*** checks if lists, pairs, ... are ***structurally equal***
- $O(\infty)$ – imagine you're comparing 2 circular lists

4. $=$: $O(n)$ ***numeric*** comparison – like the == operator in C++
- n is the number of bits

Ex. $(let\ ((x\ (/\ 0.0\ 0.0)))$
$(eq?\ \ x\ x))\ //\ true\text{:}\ \ compares\ pointers\ (same\ NaN\ object)$
Ex. $(let\ ((x\ (/\ 0.0\ 0.0)))$
$(=\ x\ x))\ //\ false\text{:}\ \ NaN\ is\ not\ equal\ to\ itself\ numerically$

3. **Error Signaling**
    - Scheme raises errors for critical issues, stopping computation
    - Ex. $(open\text{-}input\text{-}file\ "foo")\ //\ signals\ an\ error\ if\ no\ file\ "foo"\ exists$
    - Traditionally, Scheme only raised errors for very expensive operations
        - $(car\ 39)$ – Racket throws an error but Scheme does undefined behavior
4. **Undefined Behavior**
    - Most unpredictable type of error – ***implementation can do anything***
    - ***Unspecified means the result is reasonable but variable***; undefined means there's no guarantee what will happen

Sub-Note 2: Continuations

Online Notes
Definition: ***Represents the code that will execute after a function or expression returns*** (i.e. ***"what to do next"*** once you get a result)

Consider the following function in Scheme:

```
(define (foo)
        (display "hello\n")
        (display (bar)) (newline)
        (exit))
```

When you call (bar), its continuation is the code that comes right after it – display the returned value, print a new line, then call $exit$. We can represent this continuation as a function that takes the result r:

```
(lambda (r)
        (display r) (newline)
        (exit))
```

In most languages, continuations exist, but you can't see or capture them. In Scheme, ***you can capture the continuation using*** $call/cc$***.*** This gives you a function that, when called, jumps back to the point where the continuation was captured.

Under the hood, capturing a continuation saves:

- The current place in the program (e.g. the next line of code)
- The current memory state
- The current variables
- The call stack (who called whom)

This saved snapshot is then wrapped in a function. Calling the function makes the program "jump back to that snapshot.

Example 1

```
(+  2 (call/cc (lambda (cont) 3)))
```

- $call/cc$ captures the continuation as $cont$, but since we don't use $cont$, the lambda ignores it and returns 3

Example 2

```
(+  2 (call/cc (lambda (cont) (cont 10) 3)))
```

- Here, the lambda calls $(cont\ 10)$, which jumps back to the point where $call/cc$ was called, as if it had returned 10
- The 3 is never reached

Example 3

```
(define add-2 #f)
(+  2 (call/cc (lambda (cont)
        (set!  add-2 cont) ; save the continuation in add-2
```

$3)))$

- Save the continuation in $add\text{-}2$, then return 3
- $cont$ ***does nothing on its own*** – it ***only "jumps out" when you explicitly call it***
- We can call $add\text{-}2$ at a later time

Use Cases

- **Exception Handling**
    - Continuations ***allow you to jump out of deeply nested calls if an error occurs*** – much like throwing an exception – without manually unwinding each call
- **Optimizing Non-Tail-Recursive Functions**
    - Continuations can make returning from the "last call" a ***constant-time rather than linear-time*** operation

Eggert Notes

A continuation represents the "rest of the program" at any given point. Think of it as a snapshot or a to-do list of what still needs to be done in the program.

Every Scheme interpreter needs to have:

- **Instruction Pointer (ip)**: Tells the system which instruction to execute next
- **Environment Pointer (ep)**: Tells the interpreter what to do when the program returns from a function. It also stores the context (e.g. variable bindings, other state info)

$call/cc\ p$: ***Captures the current continuation***, passing it as an argument to the function p

- Internally, the continuation is stored as a pointer to a block of memory holding the current ip and ep
- Once you have a continuation (e.g. k), you can use it as a function
    - Calling $(k\ v)$ will "jump" back to the point where the continuation k was captured. It ***resets the ip and ep to the ones stored in k***, and then the program continues executing from that point
- ***Similar to a*** $goto$ ***in C but it's non-local*** – you can jump out of the current scope or back into a function that's already returned

Case Study 1: Early Exit

Imagine you want to compute the product of a long list of integers. For example:
(19, 27, 56, 1932057816, 0, 12 ...)

Multiplying many numbers could result in a huge value – but if you eventually multiply by 0, the result is 0. It would therefore be wasteful to continue multiplying after encountering a 0.

By capturing the continuation at the start of the function, we can ***immediately "jump out" when we encounter a 0***:

```
(define (prod ls)
  (call/cc
    (lambda (break)
      (let pr ((ls ls))
        (if (null? ls) ; list is empty
          1 ; multiply by 1
          (if (zero? (car ls)) ; first item in the list is a 0
            (break 0) ; no point in continuing computation after
            (* (car ls) ; multiply the first item and recurse
              (pr (cdr ls)))))))))
```

$(call/cc\ (lambda\ (break)\ ...))$ creates a continuation named $break$ that represents "what to do next" in the $prod$ function. If the function finds a 0, $(break\ 0)$ is called, which immediately ***stops the ongoing multiplication and returns 0 for the entire function***.

Note: We use the local helper function pr to avoid re-capturing the continuation on each recursive call. If we called $prod$ directly, it would create a new continuation ($break$) each time, defeating the purpose since it's essentially just regular recursion. ***By defining*** pr ***inside*** $call/cc$***, we capture*** $break$ ***once and reuse it throughout the recursion***.

Case Study 2: Green Threads

We can use continuations to implement various forms of multitasking. The simple "light-weight process" mechanism defined below allows multiple computations to be interleaved. Since it is nonpreemptive, it requires that each process voluntarily "pause" from time to time in order to allow the others to run.

```
(define gt-list '()) ; queue of tasks, holds all green threads

(define (gt-cons thunk) ; takes a thunk (nullary function) and adds it to the end of gt-list
  (set! gt-list (append gt-list (list thunk)))) // need set! since append isn't in-place

(define (start)
  (let ((next-gt (car gt-list))) ; grab the next thread from the list and store it in next-gt
    (set! gt-list (cdr gt-list)) ; remove the thread from the list
    (next-gt))) ; call the thread to execute its code

; below is an implementation of yielding
(define (yield)
  (call/cc
    (lambda (k)
      (gt-cons k)
        (start))))
```

In $yield$, $(call/cc\ (lambda\ (k)\ \ldots))$ captures the current continuation and names it k. It then adds this continuation back to the thread list. Note that calling $yield$ will jump back to this original point. Immediately after registering the yield, start is called to resume execution from the next thread in the queue.

Example 1

Suppose we add 3 threads to $gt\text{-}list$:

```
(gt-cons (lambda ()
  (let f ()
    (display "h")
    (yield)
    (f)))) ; function prints "h" then yields

(gt-cons (lambda ()
  (let f ()
    (display "i")
    (yield)
    (f)))) ; function prints "i" then yields

(gt-cons (lambda ()
  (let f ()
    (newline)
    (yield)
    (f)))) ; function prints a new line then yields

(start)
```

Each thread performs an action, then calls $yield$ to give up control before recursively calling itself to repeat the action. Since the function executes after the thread is removed from the list, no action is repeated. The result is an ***infinite loop*** where ***“hi\n” is repeatedly printed***.

Aside: Continuation Passing Style (CPS)

Normally, the computer keeps track of what to do next (e.g. continuation) automatically. In CPS, you ***explicitly write down “what to do next” by passing a continuation*** (e.g. a function) ***as an argument***.

In other words, ***every function accepts a continuation*** k. ***Instead of returning a value, they call*** k ***with the result***, telling it what to do next. Here is $prod$ rewritten in CPS:

```
(define (prod ls break)
  (let pr ((ls ls) (k break))
    (if (null? ls) ; base case: empty list
      (k 1) ; base case: return 1 and pass it to k, which represents "what-to-do-next"
      (if (zero? (car ls))
        (break 0) ; if we find a 0, stop by calling "break"
        (pr (cdr ls)
          (lambda (n) ; create a new continuation to handle the result, represents the new "k"
          (k (* n (car ls)))))) ; multiply and pass result to k
```

There is a subtle difference between k and $break$. k ***is the regular what-to-do-next continuation*** – initially, it's set to $break$, but it changes in each recursive call as it's redefined. In contrast, $break$ always refers to the original "escape" continuation captured by $call/cc$.

Pros: CPS allows you to ***pass multiple results to the continuation***, not just one. Additionally, ***you can pass different continuations for success and failure***, like callbacks.

Sub-Note 3: Memory/Storage Management

Memory is organized into sections based on purpose, listed from lower to higher addresses:
- **Text**: Read-only data (e.g. constants)
- **Data**: Initialized variables defined at compile-time
- **BSS**: Uninitialized variables, or those initialized to 0
- **Heap**: Part of BSS that grows dynamically, used for memory allocation
- **Unused Block**
- **Stack**

Anything above the heap (e.g. text, data) is handled at compile-time. The data and BSS segments are ***static***, meaning their locations are predetermined.

Examples
1. Fortran 1950
    - ***All variables and frames were statically-allocated***
    - **Pros**: Simple since no need for stack or heap pointers/management, no memory exhaustion (e.g. stack overflow), fast
    - **Cons**: Inflexible (must define size of everything at compile-time), no recursion
2. C 1975
    - ***Activation records (aka frames) are allocated on the stack***, and local variables reside within these frames
    - **Pros**: Stack management is ***cheap*** (allocate by subtracting from the stack pointer, free by adding), flexible memory allocation (can malloc/free in any order)
    - **Cons**: malloc/free are more expensive, stack allocation is LIFO
3. Algol 60

- Similar to C, but has the ability to ***dynamically define the size of local arrays at declaration***

Lecture 15

Sub-Note 1: Stack vs Heap

Stack: Stores activation records (aka frames) with all the information needed to run a function
Heap: ***Stores data that needs to live on even after a function finishes***

- Needed since statically allocated objects on the stack ***must have a size known at compile-time, but not all objects do*** – e.g. strings (size can depend on user input)
- ***Best in situations where dynamic data is being used***, like reading large user input (e.g. file parsers) ***OR when you need memory that persists beyond a function's scope*** (e.g. game engines that frequently create and destroy objects)

Example 1

$$\begin{array}{l}(define\ f\\ \quad (lambda\ (x)\\ \qquad (lambda\ (y)\\ \qquad\quad (+\ x\ y))))\end{array}$$

When you call $(f\ 12)$, it returns a new function that "remembers" the value 12 (bound to x). This returned function is like a ***fat function pointer***:

- **Instruction Pointer (ip)**: Points to the machine code that calculates $x + y$
- **Environment Pointer (ep)**: Points to the activation record of f that holds $x = 12$
- ***Even though f has finished executing, its activation record must be kept around since the returned function needs it***

Dynamic Chain

- Built from ip/ep pairs
- Each ep points to the current function's activation record, which in turn holds a pointer to its caller's activation record
- Keeps track of the sequence of function calls (i.e. ***which function called which***)

Static Chain

- Used to find non-local variables (those defined outside the current function)
- Links a function to its lexical parent (the function in which it was defined)
- Typically shorter and has an upper-bound limit

In Scheme, ***activation records for functions like f are actually stored on the heap***.This allows them to persist as long as needed – such as when the returned function is used. ***Currying generally forces activation records to be stored on the heap*** so the returned function can access its original data even after its creator function finishes).

Sub-Note 2: Heap Management

LIAO (Last-in, Anything-Out): Objects are allocated in the heap and can be freed in any order
- Heap contains many objects, which may hold pointers to one another
- ***Must be aware of external pointers*** (e.g. from stack / global registers) as these ***"roots"*** are the starting points for identifying which heap objects are being used

Case Study 1: Tracking Roots
1. **Manual Memory Management (C/C++)**
    - ***Programmer explicitly allocates and frees*** memory
    - Heap manager has no knowledge of where the roots are
    - **Pros**: Offer higher performance and simplifies the heap manager design
    - **Cons**: Very error-prone (e.g. dangling pointers after a $free$ operation)
        - Even just $return\, ptr \mathrel{!}= NULL$ after ptr is freed is undefined behavior
2. **Program-Driven**
    - ***Program explicitly tells*** the heap manager the locations of roots
    - When a root is created / destroyed, a subroutine updates the HM's state
    - Still room for human error
3. **Compiler-Driven**
    - ***Compiler automatically records*** root locations in read-only tables embedded within the executable
    - Used by Scheme/ML; require a more sophisticated compiler/runtime environment

Managed Heap: Uses a garbage collector to automatically free memory that is no longer used – no explicit $free$ or del calls
- Garbage collector starts with known ***"roots" (external pointers that reference heap objects)***, and using the layout of objects, it traces through intermediate pointers to determine which objects are unreachable (these are freed)

Question: How do we track unused memory?
- **Free List**: ***Linked list*** that ***keeps track of all the unused areas of the heap***
- Each node in the free list contains:
    - Pointer to a free block and the next node
    - Size of that free block
- Instead of maintaining a separate metadata area, this information is stored directly inside the free memory blocks (typically using the first 1–3 words of each block)

Question: How do we make finding a free block fast (i.e. how do we make $malloc(N)$ $O(1)$)? The most common allocation algorithms are:
1. **First-Fit**
    - Find the ***first block*** that is large enough
    - Typically ***fast ($O(1)$ for small allocations)*** by "peeling" off the front of the free block and adding the back portion back to the free list

  - Worst-case TC is $O(|free\ list\ length|)$
  - Can lead to ***external fragmentation***, making ***larger allocations more difficult***
2. **Best-Fit**
    - Scans the entire free list to find the ***block closest in size*** to the request
    - Reduces external fragmentation but ***requires checking every free block (slow)***
3. **Roving Pointer / Next-Fit**
    - Use a pointer that continuously cycles through the free list. For each allocation request, ***start searching where the last allocation left off***
    - Avoids scanning the same part of the list and promotes a more even distribution
4. **Segregated Free Lists**
    - Memory is divided into multiple free lists, each dedicated to blocks of a specific size class (often based on powers of two)
    - **4.1: Segregated Storage**
        - Allocator searches for a free block of the requested size
        - If no exact-size block is found, it requests more memory and creates a new pool of that size
        - Quick lookup for exact sizes BUT doesn't merge adjacent free blocks
    - **4.2: Segregated Fit**
        - If no requested-size block is found, find a larger block and split it
        - When blocks are freed, adjacent free blocks are merged
        - Reduces external fragmentation but splitting/merging introduces overhead
5. **Buddy System**
    - Start with a large block of memory and split it into 2 equally-sized blocks (aka "buddies")
    - If there's no perfect fit, the allocator repeatedly split a larger block into 2 "buddies" until it finds the right size
    - ***Requested memory sizes must be powers of 2*** – if the request isn't a power of 2, round up to the next one
    - When a block is freed, the system checks if its buddy is also free; if so, they merge back into a larger block
    - Efficient coalescing but repeated splitting can create overhead
    - Increases internal fragmentation but reduces external fragmentation

Sidenote: If the size of a free block is nearly the same as the requested size, the entire block may be allocated without leaving any leftover portion

Question: How do we make the cost of $free()$ $O(1)$?
- Specifically, we ***want coalescing to be fast*** – if the recently-freed block is adjacent to other free blocks, they should be merged into a larger block
- To do so, we ***trade space for time***:
    - When objects are allocated, ***have each allocated block store additional metadata such as its size and possibly data about neighboring blocks***
    - This allows $free()$ to quickly determine whether adjacent blocks are free and coalesce them in constant time

Question: How can we handle fragmentation?

1. **External Fragmentation**
    - Problem: Even if the total free memory is sufficient, it might be broken into many small blocks that are too scattered to satisfy a large request
    - In C, if no single free block is large enough, malloc returns a null pointer
    - **Solution**: Design the heap to minimize fragmentation – ideally by creating one large continuous area, rather than many small ones
2. **Internal Fragmentation**
    - **Definition**: This happens when the allocated block is slightly larger than needed, so some of the space inside the block is wasted
    - Although it wastes some memory, internal fragmentation is ***generally less problematic*** than external fragmentation
    - Our $\sim O(1)$ implementation of $free()$ by definition has internal fragmentation since it needs space to keep track of more metadata

Mark-and-Sweep GC Algorithm

- **Assumptions**
    - ***System knows where the roots are*** (e.g. global variables, stack pointers)
    - Knows the ***layout of objects***, specifically which fields are pointers
    - Each object has an extra ***mark bit*** (initially 0) to indicate if it's being used
- **2 main steps:**
    1. **Mark phase**
        - Start from the roots and recursively traverse all reachable objects (usually in a depth-first manner)
        - As each object is visited, set its mark bit to 1
        - TC is $O(\#\ of\ reachable\ objects\ being\ used)$
    2. **Sweep phase**
        - Go through the entire heap; any object whose mark bit is still 0 is added to the free list
        - Reset the mark bits of live objects back to 0, preparing for the next collection cycle
    - **Sidenote**: Many systems don't store mark bits inside objects; instead, the heap is split into 2 areas – ***one for the objects and one for the mark bits***

In general, ***GC address 2 big problems***: dangling pointers and memory leaks

- But with mark-and-sweep, there is ***delayed reclamation*** – memory leaks survive until the next GC cycle

Conservative Garbage Collection: Used by gcc internally

- ***Treats any value that might be a pointer as a pointer***
- **Ex**. $\#define\ free(p)\ ((void)\ 0)$ ⇒ makes $free(p)$ do nothing, avoiding dangling pointers but leading to memory leaks
- Only modification to traditional mark-and-sweep is in the mark phase:

        - Since the compiler doesn't reveal root locations or object layouts, it ***scans all possible root areas***:
            - These areas include the ***entire stack and all registers***
            - ***If a value falls within the heap's address range, it's treated as a pointer*** and used to mark objects live

Python Garbage Collector: Primarily uses ***reference counting***

- Each object in memory has an ***extra counter (e.g. word) that tracks how many references point to it***. A reference can be held by:
    - Local variables in functions, global / static variables, attributes of other objects
    - When an object's ***reference count drops to 0, it's immediately freed***
    - **Pros**: Straightforward / simple, freeing is immediate, low overhead (no need for full mark-and-sweep cycles under normal circumstances)
    - **Cons**: Need extra memory since each object must store its reference count, can't handle cyclic references by itself

Take the following code:
$a = []$
$a.append(a)$
$del(a)$

In this code, $del(a)$ just removes the reference to the list stored in a, it doesn't delete the list object itself. The ***list still contains a reference to itself***, so the ref count remains non-zero. To handle this, ***Python also uses a cycle-detecting GC to handle circular references***. Under the hood, this ***periodically performs a mark-and-sweep scan***.

Java Garbage Collector: In modern Java, allocating a new object is almost as fast as stack allocation in C

- New objects are allocated in the young generation (e.g. nursery) by simply moving a pointer forward, reserving space

1. **Generational Garbage Collection**
    - Heap is split into generations based on object age:
        - **Nursery**: Where ***new objects are created***
            - Managed using 2 pointers – heap pointer (marks start of free memory) and limit pointer (marks end of the nursery space)
        - **Old Generation**: Where ***long-lived objects are eventually moved***
    - This is based on the idea that ***most objects die young***, so the GC should be optimized to clean the nursery quickly and frequently
2. **Copying Collector**
    - When the heap pointer reaches the limit pointer, ***live objects are copied to a new space*** (or promoted to the old generation if they've aged enough)
    - Original references (aka roots) are ***updated to point to the copies***
    - **Cons**: Copying live objects and updating references takes time

Aside: Thread-Local Allocation Buffers

In multi-threaded applications, ***each thread gets its own TLAB***, a private chunk of the nursery:
- ***Can allocate faster*** – threads can allocate objects just by moving their local pointer, ***no locks or contentions with other threads***
- When a thread fills up its TLAB, it can request a new one from the JVM

Note: In C, you can avoid expensive malloc() calls by ***managing your own private free lists***

Aside: Of the three main garbage collection types (e.g. mark-and-sweep, reference counting, copying), ***copying is often the best choice***:

1. **Pros**
    - **Automatic Compaction**: Live objects are copied to a new memory space, compacting them and eliminating fragmentation (no need to manage a free list)
    - **Improved Locality**: Placing live objects contiguously boosts cache performance due to better spatial locality
    - **Rapid Memory Reclamation**: Once live objects are copied, the entire old memory space is discarded, speeding up reclamation
    - **Optimized for Short-Lived Objects**: Copying only processes live objects, so it requires far less work than mark-and-sweep which inspects every object
    - Handles circular references naturally (unlike reference counting)
2. **Cons**
    - **Higher Memory Overhead**: Requires 2 separate memory spaces (one for the new heap and one for the old heap)
    - **Limited Flexibility**: Worse for systems where many objects persist for long periods of time and don't follow a generational pattern

---

Lecture 16

Sub-Note 1: Introduction to Names / Identifiers

Sir Walter Scott: Author of Waverley
- King didn't know Sir Walter Scott was the author of Waverley
- Should be able to substitute "author of Waverley" for the name Sir Walter Scott
    - King didn't know the author of Waverley was the author of Waverley
- Why can't we do the substitution? Because there's other meanings to the name Sir Walter Scott ⇒ applies to programming languages as well

Similarly, in programming, an identifier isn't just a placeholder for a number / string – it's associated with several pieces of information:
1. **Value**: Data it holds (e.g. 27)
2. **Type**: What kind of data it is (e.g. int)

3. **Memory Address**: Location in memory where it's stored
4. **Alignment**: How the data is aligned in memory

Ex. $long\ int\ i\ =\ 27;$
$return\ i\ +\ 3;$ // equivalent to $return\ 27\ +\ 3;$

At first glance, it might seem you can always replace i with 27 – however, this substitution doesn't always work. Imagine the statement $return\ sizeof(i);$ – on an x86-64 system, $sizeof(i)$ might return 8 (because i is a $long\ int$), while $sizeof(27)$ would return 4 (size of a typical int). This is because i isn't just bound to the value 27, it's also bound to the type $long\ int$.

Ex. $long\ int\ i\ =\ 27;$
$return\ \&i\ +\ 1;$

Here, i is also bound to a specific memory address. A literal like 27 doesn't have an address, so replacing i with 27 would be incorrect.

Binding: ***Association between a name and its various attributes***
- Ex. $int\ i\ =\ 19$ creates several bindings for i:
    - $i\ \text{−>}\ int\ (type), 19\ (value), 4\ (alignment),\ 0x79c0040\ (address)$
- **Set of Bindings**: Similar to a Python dictionary, ***maps names to their properties***
    - Ex. For the statement $return\ x\ +\ y;$
        - $\{x\ \text{−>}\ int, 19; y\ \text{−>}\ double, 20.0\}$
    - **Explicit**: Directly specify the binding in your code (e.g. $int\ i$)
    - **Implicit**: Language automatically infers the binding based on the context / usage (e.g. In the function $fun\ i\ \text{−>}\ i\ +\ 1$, the type of i is bound to int)
        - In Fortran 1958, identifiers didn't require explicit type declarations; names starting with I-N were implicitly assumed to be of type int

Sub-Note 2: Binding Time

Binding Time: Point in a program's lifecycle when a name / identifier becomes bound to a particular property (e.g. value, type, address)

Ex. $int\ f()\ \{$
$\quad long\ int\ i\ =\ 27;$
$\quad <more\ code>$
$\}$

- The binding "$i\ is\ bound\ to\ the\ value\ 27$" happens during ***execution***
- The binding "$i\ is\ bound\ to\ the\ type\ long\ int$" happens at ***compile-time*** since the type is explicitly declared in the program

- The binding “$i\ is\ bound\ to\ \&i$” happens happens when the ***block is entered*** (or maybe at declaration depending on the compiler)

Ex. INT_MAX
- The binding “$INT\_MAX\ is\ bound\ to\ the\ value\ 2\text{^}31 - 1$” happens at ***platform definition time*** since INT_MAX is a fixed value / constant

Ex. $static\ int\ j$
- The ***static*** keyword indicates there’s ***only one copy*** of j for the entire program’s lifetime
- The binding “$j\ is\ bound\ to\ \&j$” happens at link-time, but if ASLR or PIE are enabled, the actual address ***may only be fixed when the program is loaded***

Separate Compilation: When a large program is compiled in pieces (modules), the compiler may not know all the details about the bindings in each module. It therefore generates code that works generically regardless of the specific binding details that will be resolved later

Sub-Note 3: Declaration vs Definition

Definition: Gives the ***complete implementation*** of a function
- Ex. $double\ difftime(timet\ a,\ timet\ b)\ \{return\ a - b;\}$

Declaration: ***Tells other modules that a function exists*** and specifies its interface (aka signature) without revealing the internal details
- Allows other modules to know how to use the function
- Ex. $double\ difftime(timet\ a,\ timet\ b);$

Inside a module, you include the full definition. When you share your code with other modules, you provide only the declaration
- This filtering ***encapsulates the logic*** – only the essential interface is exposed

Question: Why is encapsulation important? Why is modularization important?
- Encapsulation allows you to prevent external code from modifying an object’s internal state, and it localizes changes to the implementation without affecting other parts of the system ⇒ ***use cases include APIs and banking software***
- Modularization gives you parallel development / collaboration, reusability, and scalability (simplifies integration) ⇒ ***use cases include web development (separating into front-end vs back-end), microservices architecture***

Sidenote: The signature must generally match between declaration and definition, with exceptions for parameter names, default arguments (only in declaration), and inline/extern keywords (which can appear in either)

Sub-Note 4: Namespaces

Allow you to use the same name for functions or classes in different parts of a program without causing conflicts

- ***Without namespaces***, you might need to rename functions or add prefixes to avoid conflicts, ***making the code harder to manage*** when integrating third-party libraries

Case Study 1: Block-Structured Declarations

Consider the following example:

```
int x, y, z;
int f() {
	int y, z, w;
	return y + w + x;
}
int g() {return x + y;}
```

In $f()$, new local variables y, z, w are declared, with the local y and z shadowing the global y and z. ***When the block (function) ends, the global names are used again***.

Scope: The scope of a name is the ***part of the program where the name is visible***

- x is visible throughout the entire program
- Global y is visible in functions like $g()$, but the local y is used inside $f()$

Visible: Name written in the program has the desired meaning

- If you know scope, you know the visibility and vice versa

Case Study 1: Primitive Namespace

```
int f() {
	struct f {float f;} f;
	enum f {h, i} g; // can't use 'f' here since 'f' already exists in the ordinary namespace
	#include <f> // includes a header file named 'f' (diff namespace)
	#define f g; // 'f' (in the macro namespace) expands to 'g'
	f: goto f; // labels are a separate namespace, becomes g: goto g
}
```

Clarifications:

- The ***function name*** f lives in the ***global namespace***
- The declaration $struct\ f$ … introduces:
    - Struct tag called f in its own namespace
    - Member of that struct named f (accessible only after a . or $->$)
- The ***variable*** f (instance of struct f) declared inside the function occupies the ***ordinary / local namespace***

- Although $enum$ tags are in a different namespace than $struct$ tags, using f in the $enum$ declaration or "name" would conflict with the existing local variable f

Sub-Note 5: Information Hiding

Information hiding is the practice of ***separating a module's internal details*** (its implementation) ***from the interface it exposes to the rest of the program***:

- **Enhances Modularity**: Modules can be reused, updated, or replaced without affecting other parts of your program
- **Supports Interchangeability**: Different modules with the same external interface can be swapped in easily

1. **Controlling Visibility in Modules**
    - When defining a module, you decide which identifiers (e.g. functions, variables) are visible outside it
    - In C:
        - **static:** Marks an identifier as ***private*** to the module
        - **extern:** Makes an identifier ***visible across all modules***
    - For Java, the table below shows where each modifier (e.g. public, protected, default, private) allows a method to be accessed:

Modifier	Same class	Other class in the same package	Subclass in another package	Other places where the class is visible
Public	Visible	Visible	Visible	Visible
Protected	Visible	Visible	Visible	
(default)	Visible	Visible		
Private	Visible			

Java's module/package hierarchy benefits from $protected$ since it ***gives you a balance between encapsulation and flexibility***. It allows members to be accessible within the same package or to subclasses in other packages, but doesn't expose everything publicly.

2. **Explicit Namespace Operators**
    - Often begin with a detailed namespace that reveals all internal details but want to present a simpler, more abstract interface. Languages like ***OCaml enable this at compile-time through signatures***:
    - The ***full queue implementation is available within the module***, but the signature only exposes the necessary operations (like $enqueue$)
    - ***Checked at compile-time***, making it more flexible than what Java gives you

Case Study 1: OCaml Queue

Concrete Module Implementation:

```
module Q = struct
    type 'a queue = | Empty | Node of int * 'a * 'a queue
    let enqueue = … (* detailed implementation *)
end
```

Abstract Interface with a Signature:

```
module type QI = sig
    type 'a queue // says we have an abstract type but don't know its implementation
    val enqueue: 'a * 'a queue -> 'a queue
end
```

Sub-Note 6: Ways to Address Errors/Faults/Failures/Bugs

- **Error**: Human mistake
- **Fault**: Hidden problem in your program
- **Failure**: When the program runs but doesn't behave as expected

1. **Compile-Time (Static) Checking**
    - ***Detects mistakes before the program runs*** (e.g. static type checking in Java/Python prevents programs from making type mistakes)
    - ***Gives you reliability and some performance but reduces flexibility***
2. **Preconditions**
    - Language support (e.g. in Eiffel) for declaring conditions that must be true before entering a function or a code block
    - Ex. $int\ sqrt(int\ n)$ {
      $pre\ n \geq 0$; // caller must ensure $n \geq 0$
      }
3. **Total Definitions**
    - A function is "total" if it ***provides a valid output for every possible input*** (common in Rust)
    - Ex. $double\ sqrt(double)$; – on Linux, calling $sqrt(\text{-}1.0)$ returns NaN, ensuring a double is always returned no matter the input
4. **Fatal Exits**
    - Program ***intentionally crashes*** when a critical error occurs
    - Ex. Calling $abort()$ to crash the program instead of printing the wrong answer
5. **Exception Handling**
    - Structured way to manage errors during program execution by ***separating normal logic from error-handling code***

```
try {
    putchar(getchar());
```

```
        do_some_stuff(); // core code
} catch (IOError e) {
        // Note: More often than not in real software, error handling is the bulk of your code
        // This is because you want to maintain reliability
        fixup(e);
} finally {
        printf("Done")
}
```

Trade-Off: Exceptions ***give you better readability and modularity***. However, they ***can obscure control flow*** – it may not immediately be clear what happens when an error occurs, especially if exceptions are thrown deep within nested calls. To contrast, here's how C does error handling:

```
int c = getchar();
if (c < 0)
        fixup(c);
if (putchar(c) < 0)
        fixput();
```

In a sense, the C approach is harder to read since the error handling is intertwined with the actual code. ***If you're trying to write reliable code, use exception handling.*** However, in systems where ***clear, predictable behavior is essential*** (e.g. in critical applications like a payroll system), the explicit nature of ***C-style error handling may be preferred***.

6. **Checked Exceptions (Java)**
    - Methods explicitly declare the exceptions they might throw using Java's built-in $Throwable$ class
    - **Benefit**: At compile-time, you can see which exceptions a method might throw and plan for them

---

Lecture 17

Sub-Note 1: Parameter Passing

1. Semantics

- **Correspondence Models**
    - **Positional Correspondence**
        - When you call the function $f(a, b)$, each argument is matched in order with the corresponding parameter in the function definition
        - Ex. $def\ f(x, y)$ maps a to x and b to y
    - **Varargs Correspondence**

        - If you call a function with more arguments than there are explicit parameters (e.g. $f(a, b + 1, c - 5)$ for a function defined as $def(f, *y)$, the last 2 arguments are packaged into a tuple and passed as $*y$
    - **Keyword Correspondence**
        - Functions can also be called with named arguments (e.g. $f(y = 5, x = 10)$), explicitly matching arguments with parameters regardless of their order

2. Efficiency

a. **Call by Value**
    - Caller evaluates each argument and passes a copy of its value to the callee
    - ***Gives you isolation*** (changes made to the parameter inside the function don't affect the original value) ***and speed*** (efficient for small data types like integers)
    - ***Inefficient for large data structures*** since it requires copying the entire value

b. **Call by Reference**
    - Caller ***passes the address*** of the argument
    - ***More efficient for large data*** since only an address is passed
    - ***Inefficient for small data*** (have to create an address, put it into RAM...)
    - Parameter can serve as both input and output

The ***biggest downside of call-by-reference*** is ***aliasing***, where ***2+ parameters refer to the same memory location***:

```
int f(int& x, int& y) {
        print(x);
        y = x++; // undefined behavior in C++ if x and y refer to the same variable
        print(x);
}
```

Calling $f(i, i)$ with $i = 12$ might not produce the expected output (12 then 13). Aliasing is ***problematic*** since it ***makes the code harder to reason about***. It also ***slows down execution*** since the ***compiler can't optimize when aliasing occurs (no caching allowed)***.

c. **Call by Result**
    - Caller provides a location for the result and the function writes to an uninitialized variable, which is then copied back to the caller upon return
    - **Ex**: In the function $int\ read(int\ fd, char\ *buf, int\ bufsize)$, the ***buffer*** buf ***is a call-by-result parameter***
        - The initial contents of buf are irrelevant since the function fills it with data
    - ***Enables functions to effectively return multiple values*** (e.g. in the above, we return a status code but also indirectly an output array)

d. **Call by Value-Result**
    - First ***passes a copy of the argument*** (like call by value) and ***then copies the result back to the caller*** when the function returns (like call by result)
    - Similar to call by reference but ***avoids aliasing*** by making copies

        - Gives you ***safety*** but is ***inefficient*** (requires copying the value twice)

Case Study:

```
$begin$
    $integer\ n;$
    $procedure\ p(k\colon\ integer);$
        $begin$
            $n := n + 1;$
            $k := k + 4;$
            $print(n);$
        $end;$
    $n := 0;$
    $p(n);$
    $print(n);$
$end;$
```

1. Call by Value
    - k receives a copy of n; ***changes to*** k ***don't affect the original variable*** n
    - Output is $1\ 1$
2. Call by Value-Result
    - k receives a copy of n; BUT when the procedure finishes, the ***updated value of*** k ***is copied back to*** n
    - Output is $1\ 4$
3. Call by Reference
    - k is an alias for the original n; ***changes to*** k ***also affect*** n ***directly***
    - Output is $5\ 5$

e. Macro Calls
- A ***macro*** is a ***piece of code that is expanded or replaced at compile time*** before the program runs
    - The ***macro's arguments are literal pieces of code*** that are directly substituted into the macro's body
    - Ex. $\#define\ SQUARE(x)\ ((x)\ *\ (x))\ int\ b\ =\ SQUARE(5);$
- Since macros are expanded at compile-time, they ***eliminate the overhead of function calls***
- However, they ***don't perform type checking*** (incorrect arguments will lead to errors that only appear at runtime)
    - Therefore, ***errors can be hard to trace*** since the substitution occurs before the program runs

f. Call by Name
- ***Caller passes a "thunk"*** (small, unevaluated procedure) that ***tells the callee how to evaluate the expression*** when needed

- Avoids premature evaluation which can ***prevent runtime errors*** (e.g. division by 0), thereby ***giving you safety***
- ***Slower*** since the expression may be re-evaluated multiple times

Example 1

Callee:

```
$void\ printavg(int\ nItems,\ int\ avgItemVal)$ {
    $if\ (nItems == 0)\ print("no\ items");$
    $else\ print("avg\ is\ ", avgItemVal);$
}
```

Caller:

```
$for\ (int\ i = 0; i < n; i{+}{+})$ {
    $sum \mathrel{+}= a[i];$
$printavg(n, sum\ /\ n);$
```

With call by value, $sum\ /\ n$ ***is computed immediately***, which can cause a crash if n is 0. ***With call by name, the expressions for*** n ***and*** $sum\ /\ n$ ***are passed as thunks. The function can check*** $n == 0$ ***before evaluating the division, avoiding a potential crash.***

g. **Lazy Evaluation (Call by Need)**
    - ***Extension of call by name*** where ***each "thunk" is evaluated at most once*** (its result is cached)
    - Languages like Haskell use lazy evaluation
        - ***Supports infinite data structures*** – can define an ∞ list of primes (e.g. $let\ pl = list\ of\ all\ primes$) and only the parts needed will be computed

h. **Call by Unification** (Works similarly to Call by Value Result)

Case Study:

```
$begin$
    $array\ a[1\ldots 10]\ of\ integer$
    $integer\ n$
    $procedure\ p(b\colon\ integer);$
        $begin$
            $print(b);$
            $n := n + 1;$
            $print(b);$
            $b := b + 5;$
        $end;$
    $a[1] := 10;$
    $a[2] := 20;$
    $a[3] := 30;$
```

```
        $a[4] := 40;$
        $n := 1;$
        $p(a[n + 2]);$
        $new\_line;$
        $print(a);$
$end;$
```

1. Call by Reference
    - Since $n = 1$, b ***becomes a reference to*** $a[n + 2] = a[3] = 30$
    - $b := b + 5$ changes $a[3]$ to 35
    - Output is $30\ 30\ \backslash n\ 10\ 20\ 35\ 40$
2. Call by Name
    - ***Expression*** $a[n + 2]$ ***is substituted and re-evaluated each time*** b ***is used***
    - $print(b)$ is like $print(a[n + 2])$, and since $n = 1$, $a[3] = 30$ is output
    - $b := b + 5$ is like $a[n + 2] := a[n + 2] + 5$, and since $n = 2$ (it was incremented previously), it changes $a[4]$ to 45
    - Output is $30\ 40\ \backslash n\ 10\ 20\ 30\ 45$
3. Call by Need
    - Expression $a[n + 2]$ isn't evaluated until b is used, but ***unlike call by name, once evaluated, the result is cached and reused***
    - $print(b)$ causes $a[n + 2]$ to be evaluated, and since $n = 1$, it evaluates to $a[3] = 30$, which is printed and cached
    - Output is $30\ 30\ \backslash n\ 10\ 20\ 35\ 40$
    - Would differ from call by reference if we incremented n before the first $print(b)$

Sub-Note 2: Extra Hidden / Transformed Parameters

Many languages automatically pass additional parameters during function calls. These include:
1. $this$ **pointer (OOP)**: refers to the ***current object***
2. **Static Chain Pointer**: points to the ***frame where the function is defined***
3. **Dynamic Chain Pointer**: points to the ***caller's frame*** (e.g. $\%rbp$ on $x86$-64)
4. **Return Address**: address on the stack where control should return after the function finishes (i.e. top of the stack)
5. **TLS (Thread-Local Storage)**: pointer to data specific to the current thread

Aside: Extra Slots in Objects (OOP)

Objects in OOP are typically represented by a pointer to their value. They may also contain extra slots for:
- **Type Information**: pointer to the object's type data
- **Garbage Collection**: used for memory management
- These extra slots provide metadata and support behind-the-scenes system operations

Sub-Note 3: Cost Models

Cost Model: Mental model for understanding the computational resources needed to efficiently execute a program

- **Big O Notation**: Describes how an algorithm scales (asymptotic efficiency)
- **Absolute Costs**: Constant factors (i.e. those ignored in Big O) that also matter in real-world systems
- Some example costs include:
    - **Real time / System time** (i.e. time doing OS operations like kernel calls) / **CPU time** (i.e. time spent by processor executing instructions, excludes I/O)
    - **Network access** (e.g. Slow servers can slow real time but not CPU time)
    - Memory (e.g. registers, RAM)

Prolog Unification Cost: $O(size\ of\ the\ smaller\ tree)$ since the process stops when the smaller structure is fully unified

Case Study 1: Lisp vs Python Lists

Classic Model: Lisp Lists

- Implemented as ***linked list*** structures
- Append Cost:
    - $(append\ a\ b)$ takes $O(size\ of\ a)$ time because it ***creates a copy of*** a and points the last element to b
    - $(append\ '(foo)\ x)$ is $O(1)$ while $(append\ x\ '(foo))$ is $O(x)$ ⇒ only know the difference if you looked at the cost model

Comparison: Python Lists

- ***Dynamic arrays*** represented as a header that points to an array of item pointers
- Append Cost:
    - Usually $O(1)$ since it's just a load and two stores
    - Occasionally $O(size\ of\ list)$ when the array needs to be resized, but amortized $O(1)$ over many appends

Case Study 2: C/C++ Array Access Cost

Suppose we want to access an element from an array:
$double\ a[100000];\ double\ x = a[i];$

Question: How can we compute $a[i]$ fast if we don't know i at compile-time?

- Assume the base address of a and the index i are stored in registers
- The address for $a[i]$ is just $a + (i * 8)$ since $sizeof(double) = 8$

Suppose we want to add reliability through subscript checking. A naive assembly implementation might look like:

```
cmpq %rdi, 0
jl error // if the index < 0
cmpq %rdi, $16 // assuming array size is 16
jge error // if the index  ≥ size
```

Using an unsigned comparison reduces bounds checking overhead by a ***factor of 2***:

```
cmpq %rdi, $16
jae error // jump if unsigned index is above 16
```

Lecture 18

Sub-Note 1: Features of Rust

1. **Memory Safety**
    - ***Rust performs safety checks at compile-time***, catching errors like null pointer dereferencing before the program runs
        - Unlike OCaml / Scheme / Python which only do this at runtime
    - ***Can catch common bugs*** that crash programs and are hard to debug ***early-on***
2. **Zero-Cost Abstraction**
    - ***Rust allows you to write safe, abstract code without sacrificing the speed of low-level C/C++ code***

Case Study: Static Checking in Encryption-Based Programs

- **Scenario**
    - You want to run your program $y = f(x)$ on a remote server (e.g. Amazon AWS) while keeping your data private
- **Encryption Workflow**
    - You encrypt the input x before sending it to the remote machine
    - The server uses a modified version of your function, $f(E)$, that operates on the encrypted data and produces an encrypted output $E(y) = f(E(x))$
- **Problem**
    - Since the program runs on encrypted data, if an error occurs the ***resulting backtrace is also encrypted***, making it hard for both you and the server to diagnose the problem
- **Solution**
    - ***Perform static checking on the original, unencrypted program*** $y = f(x)$ to catch errors before encryption

3. **Ownership and Borrowing Model**

Rust's ownership system is at the heart of its memory safety guarantees. It provides a ***zero-cost abstraction***, but there is a ***learning curve*** (many new users "fight with the borrow checker"). This model ***simplifies safe concurrency***, enabling more efficient multi-core computing than in C, where managing locks is a constant concern.

- **Static Ownership Tracking**
    - Each variable binding "owns" the data it's bound to; when it goes out-of-scope, Rust automatically frees its resources
    - Rust ***keeps track at compile-time which reference owns an object and which ones are merely borrowing access***
    - Compiler uses these static checks to prevent mutation of shared state, ***reducing race conditions*** – unlike in C, where manual locking is required

- **Enforcing Mutability Rules**
    - Since there is only one owner of memory at a time, Rust enforces either ***one reference to mutable memory*** (e.g. only one thread accessing/updating the object) ***or multiple immutable references***
    - In C/C++, data races can only occur if you have ***both shared and mutable data***
        - ***FP*** enables parallel computation by ***eliminating mutability***
        - ***Rust doesn't allow shared and mutable references at the same time***
    - When ownership is transferred to a new binding, the original binding typically becomes invalid (except for types implementing the $Copy$ trait, like primitives), ensuring that two bindings don't access the same resource concurrently

- **Scope-Based Cleanup**
    - When a reference pointer that owns an object drops out of scope, its drop method is automatically invoked (also handles transitive closure)
    - This ***automatic cleanup*** prevents issues like dangling pointers, ***eliminating the need for a garbage collector***

- **Understanding Immutability**
    - In Rust, "immutable" means you have shared access ***without the ability to modify data through that reference***
    - Similar to C's $const$ (e.g. $int\ const\ \ *p$), immutability refers to the current state of access rather than permanently preventing changes; the object can still change through other means

4. **Additional Features Necessary for Memory Safety**
    - **Runtime Bounds Checking**: When you access an array with $A[i]$, Rust checks at runtime that i is within bounds; otherwise it prints a diagnostic message
        - Rust uses ***fat pointers*** – in addition to a pointer to an array, also includes an upper bound field
    - No pointer arithmetic, automatic resource cleanup (no need for manual $free/del$ since the drop method is automatically invoked), no null pointers

Together, these mechanisms help prevent use-after-free, double free, buffer overrun, null pointer dereferencing, and data races (if implemented correctly).

Aside: As an option, Rust allows sections of code to be marked as $unsafe$ to bypass some of the compiler's safety checks. This ***gives you C/C++ performance***, but you ***must manually verify that the code is memory-safe***, which is a hassle.

5. **Rust Failure Handling**
    - $panic$ **primitive**: Prints a backtrace for debugging, unwinds the stack and then halts the program
    - Rust uses the $enum\ Result{<}T, E{>}\ \{\ Ok(T),\ Err(E)\ \}$ to handle recoverable errors
        - Very similar to $type\ ('t,\ 'e)\ result\ =\ |\ Ok\ of\ 't\ |\ Err\ of\ 'e$ in OCaml

6. **Cargo Build System**
    - Cargo is Rust's build system and package manager
    - Automates tasks such as dependency management, testing, performance benchmarking, and lint checking, streamlining the development process

Sub-Note 2: Program Semantics

Semantics is divided into 2 main subcategories:

1. **Static Semantics**
    - ***What you can determine*** about the program's meaning ***before running it***
    - Most useful for traditional software applications
    - **Attribute Grammars**
        - **Parse Tree with Attributes**: In a basic grammar, each node only carries a nonterminal symbol. By ***assigning additional attributes (e.g. type and scope) to these nodes***, we gain more semantic information
        - **Synthesized Attributes**: Get their values from their ***children***
            - Information flows up from children to parent

Ex. Type
For the rule $Expr1\ =\ Expr2\ +\ Expr3$, you might have the following supplement:
$type(Expr1)\ =\ if\ type(Expr2)\ ==\ int\ and\ type(Expr3)\ ==\ int\ then\ int\ else\ float$

- **Inherited Attributes**: Get their values from the ***parent or siblings***
    - Information flows down (from parent to child) or sideways (between siblings)

Ex. Scope
If a variable is declared higher up in the tree (e.g. in an outer scope), the current node needs to inherit the current symbol table to know what variables are in scope.

Aside*: Simple information (e.g. type) is often synthesized* since you can compute it from the children. ***More complex information (e.g. scope) is often inherited*** since it depends on context beyond the subtree.

2. **Dynamic Semantics**
    - Describes ***what the program means as it runs***; there are 3 main approaches:

1. **Operational Semantics (Best for Imperative Languages)**
    - Explains what a program does by ***describing how it executes step-by-step***
    - Forms the basis for interpreters and compilers

Imagine you have a new language L. To understand it:
- Write an interpreter I for L
- Implement this interpreter in a simpler language K you already understand
- Run programs P written in L by executing them through I
- Alternatively, study the source code of I to understand how L works

Case Study: Defining Semantics for ML in Prolog

Describe how an ML expression E evaluates in a context C, producing a result R. To do so, define a predicate $m(E, C, R)$ where:
- E is the ML expression to evaluate
- C is the context (dictionary of variable bindings, e.g. $[x = 12;\ z = fun(\ldots)]$)
- R is the result

$m(E, \_, E) :- \ integer(E).$ // integer evaluates to itself, context doesn't matter
$m(E, C, R) :- \ atom(E),\ member(E = R, C)\ !$ // look up the variable's value in the context
$m(let(I, E, F), C, R) :- \ m(E, C, V),\ m(F, [I = V|C], R).$
- Evaluate E to get V, then evaluate F in an expanded context where I = V

$m(fun(I, E),\ C,\ lambda(I, C, E)).$
- A function doesn't evaluate its body immediately, instead, it returns a representation of the function including:
    - Parameter I
    - Context (C) where it was defined
    - Body E (evaluated when the function is called)

$m(call(lambda(I, C, E1), E), \_, R) :- \ m(let(I, E, E1), C, R).$
- Function call is like a let binding:
    - Evaluate E (the argument), bind it to I, then evaluate the function body (E1) in the function's saved context C

Classic Example: Lisp 1.5 defined the semantics of Lisp by writing a Lisp interpreter in Lisp. Although this seems bogus because it's circular, it ***works in practice through bootstrapping***.

2. **Axiomatic Semantics (Best for Logic Languages)**
    - Provides a logical framework for reasoning about program correctness without executing the code
    - **Hoare Triples**: $\{P\}\, S\, \{Q\}$
        - **P**: Precondition; **S**: Statement; **Q**: Postcondition
        - ***If P is true and S executes successfully, Q must be true***
        - **Ex**. $\{i < 0\}\, i := i + 1\, \{i \leq 0\}$
            - If $i < 0$ is true before executing i++, then $i \leq 0$ will be true after

Conditionals: To represent $if\ B\ then\ T\ else\ E$, establish a triple for both branches:
- $\{P \land B\}\, T\, \{Q\}$ and $\{P \land \neg B\}\, E\, \{Q\}$ where ∧ means AND and ¬ means NOT

Loops: To represent $while\ W\ do\ S$, define an invariant P such that:
- While the loop condition W is true, executing S maintains P ⇒ $\{P \land W\}\, S\, \{P\}$
- When W is false, P guarantees the postcondition Q ⇒ $\{P \land \neg W\} -> Q$
- Use a monotonically decreasing non-negative integer function to prove termination

This allows us to verify correctness properties of our code statically without running the program, which is a huge advantage over operational semantics.

3. **Denotational Semantics (Best for Functional Languages)**
    - Semantics of a program P is represented as a function from inputs I to outputs O
    - $semantics(P)\colon I -> O$

Aside: Static vs Dynamic Checking

Static Typing:
- **Early Error Detection**: Mistakes – like adding a number to a boolean – are caught before the program even runs
- **Clear Documentation**: Function signatures show exactly what types are expected in arguments
- **Better Optimizations**: Compiler can use type information to speed up code, for example, by replacing indirect calls with direct ones

HOWEVER, ***static type checking only approximates runtime behavior***. Some errors may still slip through since not every property is tracked. Additionally, ***safe programs might be rejected by overly strict type systems*** (think “phantom types” or “wobbly types”).

Dynamic Typing:
- **Flexibility for Prototyping**: Ideal for building systems quickly when requirements are changing or not fully defined
- **Handling the Unpredictable**: Naturally supports features like dynamic loading, method interception, and runtime reflection
- However, this means errors might be discovered too late in the development process

In general, ***static checking works best for large, critical projects*** (e.g. enterprise applications, performance-critical systems) that need reliability and early error detection. ***Dynamic checking is better for projects that require rapid development and flexibility.***