CS131 Notes

March 20, 2025

Contents

1 *<2025-01-07 Tue>* Lecture 1

1.1 Introduction

In the late 1960s, Donald E. Knuth wrote the first three volumes of *The Art of Computer Programming*. Dissatisfied with the typesetting produced by Addison Wesley for the second edition printing, he decided to create a computer typesetting system called TEX to produce a book that was up to his standards.

TEX was written in Pascal, and in order to teach other people about it, Knuth decided to then write a book on TEX, and of course, wished to use TEX to typeset it. However, he faced the challenge of updating the TeXbook whenever TEX was updated.

To solve this issue, Knuth came up with the idea of literate programming, where documentation and code are interwoven in the same document. In this method of programming, the code is subservient to the documentation: the explanation and human readability comes first. Utilities are thus needed to transform this master document into either documentation or source code.

In 1986, Jon Bentley, who had a column in *Communications of the ACM*, was interested in literate programming, and asked Knuth to write a program to print a sorted list of the n most frequent words in some input text along with their frequencies. In addition, he asked Doug McIlroy (who came up with the idea of using the | character for shell pipes) to write a critique of Knuth's style.

Knuth produced an elegant, 4 page long algorithm in WEB, his literate programming system that produced both documentation and Pascal source code from the same file. McIlroy, in contrast, produced a bug-free, 6 line long shell script that accomplished the same goal.

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

The Sapir-Whorf hypothesis (in linguistics) states that the language we use, to some extent, affects how we think. Similarly, the programming languages we use affect the way we think about the problems we solve–this is why we study programming languages.

1.2 Judging Languages

To judge a programming language, we need to look at their costs/benefits: this includes things like training + community, reliability, maintenance costs, runtime/build/porting expense, and licensing fees.

1.3 Judging Languages (Continued)

In addition, we need to think about issues pertaining to the design of the language itself, including:

- Orthogonality
  - Do operations change just one thing, or do they potentially modify multiple aspects of a running program?
- Efficiency
  - CPU/real time speed, RAM/cache usage, power usage, and how networking and I/O work
  - Cost of building a program vs the cost of running it
- Maintainability
  - Write-ability/readability: how easy is it to write/read programs in the language? Note: in practice, readability is usually more important than write-ability.
  - Extend-ability/mutability: how easy is it to change/extend the language over time? Successful languages evolve with time.
  - Stability: how sure are we that the features of the language are stable, and that our programs won't break in the future due to changes to the language?
  - Documentation/training: how easy is it to pick up the language or document programs written in it?

- Safety: what assurances do we have that our program won't randomly crash?
- Portability: can our program run on Windows and Mac?
- Debug-ability: how easy is it to diagnose issues with our program?

1.4 Types of Programming Languages

1.4.1 Imperative Languages

Imperative languages consist of a series of commands given to the computer to execute in a specific order that change the program's state. Think A; B; C;. Imperative languages offer significant control over the computer. Examples include C/C++, Python, and Java.

Note that the sequential, command-based nature of imperative languages makes it inherently difficult to scale programs written in them to modern, multi-threaded and distributed computing models.

1.4.2 Functional Languages

Functional languages, on the other hand, are based on the idea of mathematical functions. Think A(B(x, y), C(z)). In contrast to imperative languages, this way of thinking gives us some degree of parallelism right off the bat: we could evaluate B(x, y) and C(z) at the same time. Examples include ML, OCaml, Scala, Lisp, and Scheme.

In addition, to their parallelizability, functional languages are motivated by clarity. Mathematical functions are well tuned to how humans think: a statement like `i = i + 1` makes no sense mathematically. In addition, in imperative languages, a statement like `i + j` can lead to different values depending on the time/location of its execution. Functional languages give us referential transparency, i.e. we always know what an identifier refers to.

In order to wipe out things like race conditions (which plague imperative languages), purely functional languages need to give up any side effects, including things like assignments.

However, note that without the side effect of I/O, we can't ever get work done. Thus, both functional/logic languages need some escape hatch. For instance, OCaml's REPL (read-eval-print loop) allows its command line interface to communicate with the user. In the real world, functional languages aren't purely functional but generally encourage the user to write in a functional style.

1.4.3 Logic Languages

Logic languages are a subset of functional languages based on predicates, which are simply logical statements. Using logical operations, we can combine predicates to reason about things. The logic model may not be used as much as the functional/imperative models, but the logical way of thinking can be applied to things like database queries and static analysis of other languages. The most successful logic language is Prolog.

1.5 ML

The ML language has:

- Compile-time type checking
- Type inference
- Garbage collection
- Higher order functions
  - Functions that take functions as arguments/return functions as results

1.6 OCaml

OCaml extends the Caml dialect of ML with object-oriented features. We ignore the object-oriented part. Code listings are produced from OCaml's built in REPL.

1.6.1 The Beginning

```
# let x = 37 * 37;;
val x : int = 1369
```

Everything in OCaml has a value, and every value has a type. Note that OCaml infers the type of `x` automatically. In this case, we've bound the value `1369` to the identifier `x`. Note that bindings in OCaml are immutable: the value assigned to some name never changes.

```
# if x < 2000 then "a" else "b";;
- : string = "a"
```

In OCaml, `if ... then ... else ...` is an expression, not a statement: it evaluates to some value. Think of it as a ternary operator. The `-` in the response from the REPL loop indicates that we've evaluated an anonymous expression.

```
# if x < 2000 then "a" else 27;;
(* Error: This expression has type int but an expression was expected of type string *)
```

This results in an error: OCaml checks the types of `"a"` and `27` and notes that they are different before executing the code.

```
# let dummy = "hi" = "hello";;
val dummy : bool = false
```

Note that in OCaml, the `=` symbol is used both for definition and comparison. Its negation (for comparison) is `<>`.

1.6.2 Tuples and Lists

```
# (1, "a");;
- : int * string = (1, "a")

# (1, "a", 3.5);;
- : int * string * float = (1, "a", 3.5)
```

These are OCaml tuples: fixed length collections of elements of any type. Tuples with two elements are known as pairs. Note that the types of these tuples are denoted using the `*` symbol and that elements are separated using commas.

```
# ();;
- : unit = ()
```

The empty tuple is of the type `unit`. Units are typically associated with side effects and indicate the absence of useful data.

```
# [1; -3; 17]
- : int list = [1; -3; 17]

# [1; "a"]
Error: This expression has type string but an expression was expected of type int
```

These are OCaml lists: in contrast to tuples, each member of a list must be of the same type. However, lists allow an infinite number of members whereas tuples have a fixed number of members (depending on their type). Note that unlike tuples, list elements are separated with semicolons.

```
# [];;
- : 'a list = []

# [1; 3] = [];;
- : bool = false
```

`'a` denotes a type variable standing for any type. This allows us to, say, compare `[]` with an `int list`, at which point `[]` is inferred to be an `int list`.

```
# 3 :: [4; -2; 7];;
- : int list = [3; 4; -2; 7]
```

We can prepend to lists using `::`, the "cons" operator. This name is taken from Lisp.

1.6.3 Functions

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

# f 3;;
- : int = 4
```

Functions are also values in OCaml, meaning we can define them with the `let` keyword as well. In this case, our function `f` has a single parameter `x` and the function body `x + 1`. There is no return statement in OCaml. Note that `f`'s type is `int -> int`.

```
# (fun x -> x * x) 50;;
- : int = 2500
```

We can also define anonymous functions using the `fun` keyword.

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

# cons ("a", ["b"; "c"; "d"]);;
- : string list = ["a"; "b"; "c"; "d"]
```

In this manner, we define a `cons` function to do the same operation as `::`. Note that the compiler infers `x` to be of type `'a` and `y` to be of type `'a list`, indicating that it expects `x` to be of the same type as those in the list `y`.

1.6.4 Multiple Parameters, Partial Application, and Higher-Order Functions

Let's define a function that adds its two integer parameters.

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
```

Note the typing of this `add` function: it implies we could apply `add` to just a single argument (`x`), instead of the two we defined it with (`x` and `y`). Indeed:

```
# let add5 = add 5;;
val add5 : int -> int = <fun>

# add5 2;;
- : int = 7
```

This means we can think of our `add` function as a higher-order function that takes one integer parameter and returns a function of type `int -> int` (Recall that higher-order functions are those that take functions as parameters, return them, or both).

There is a powerful implication here: functions with multiple parameters are actually just syntactic sugar for single-argument functions that themselves return functions. More precisely,

```
let f x1 x2 ... xn = e
```

```
let f =
  fun x1 ->
```

```
    (fun x2 ->
      (...
        (fun xn -> e)...))
```

are semantically equivalent. This process of turning a function that takes multiple arguments into a sequence of single-argument functions is known as currying. Note that this means that function types are right associative and function application are left associative:

```
t1 -> t2 -> t3 -> t4
```

```
t1 -> (t2 -> (t3 -> t4))
```

are equivalent; so are

```
e1 e2 e3 e4
```

```
((e1 e2) e3) e4
```

1.6.5 Pattern Matching

To do pattern matching in OCaml, we use the `match` keyword. For instance, we can write a simple `map` function:

```
# let rec map f u =
    match u with
    | [] -> []
    | x :: y -> f x :: map f y;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# map (fun x -> x * x) [1; 2; 3; 4];;
- : int list = [1; 4; 9; 16]
```

Note the `rec` keyword to denote a recursive function. Our match statement allows us to take care of the two cases: an empty list and the case where we can match the list `u` with the pattern `x :: y`, in which case the identifier `x` will be bound to the value of the first item in `u` and the identifier `y` will be bound to the value of `u`'s tail.

```
# let g x =
  if x = "foo" then 1
  else if x = "bar" then 2
  else 0;;
val g : string -> int = <fun>

# let g' x = match x with
    | "foo" -> 1
    | "bar" -> 2
    | _ -> 0;;
val g' : string -> int = <fun>
```

The two functions `g` and `g'` are functionally identical. Note that we can pattern match using values and that the `_` pattern is a catch-all, i.e. it matches with anything and discards the value.

```
# let car (x :: _) = x;;
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>
```

As a benefit of its typing system, OCaml helpfully warns us when a pattern match is not exhaustive.

```
# let second p =
    match p with
    | (_, y) -> y;;
```

```
val second : 'a * 'b -> 'b = <fun>

# second (42, "apple");;
- : string = "apple"
```

Note that we can access the members of a tuple by pattern matching. We can do a similar pattern matching with lists, but we need to be careful to take care of every possible case as lists can have a variable number of elements.

2 *<2025-01-09 Thu>* OCaml - Continuing OCaml Concepts

2.1 Currying

2.1.1 Definition and Example

Currying in OCaml allows you to define functions that take arguments one at a time.

```
fun x y -> x + y + 1;;
```

This is equivalent to:

```
let p x y = x + y + 1;;
```

2.2 Recursion

2.2.1 Basic Recursion Example: Factorial

```
let rec fact n =
  match n with
  | 0 -> 1
  | _ -> n * fact (n-1);;

fact 5;;
```

2.2.2 List Reversal - Inefficient $O(N^2)$ Version

```
let rec reverse l =
  match l with
  | [] -> []
  | h::t -> (reverse t) @ [h];;
```

This version is inefficient, having a time complexity of $O(N^2)$ due to repeated list concatenation.

2.2.3 List Reversal - Efficient Accumulator Version

```
let rec rev l a =
  match l with
  | [] -> a
  | h::t -> rev t (h::a);;

let reverse l = rev l [];;

reverse [1;2;3;4;5;6];;
```

2.2.4 List Reversal - More Concise Version

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

Here, `a` acts as an **accumulator**, storing the partially reversed list.

2.2.5 Finding the Maximum Element in a List - Initial Attempt (with Issue)

```
let rec max = function
  | [] -> -9999999 (* How to represent negative infinity? *)
  | h::t -> let m = max t
            in if h < m then m else h (* < only works on int *)
```

The issue is how to handle negative infinity properly for the base case with an empty list.

2.2.6 Finding the Maximum Element in a List - Improved Version with Higher-Order Function

```
let rec max i lt = function
  | [] -> i
  | h::t -> let m = max i lt t
            in if lt h m then m else h;;

let imax = max (-99999999999999) (<);;
let a = imax [1;2;3;4;5;6];;
```

This version is more flexible as it takes an initial value `i` and a less-than comparison function `lt` as arguments, allowing it to work with different types and comparison criteria.

2.3 `fun` vs `function` in OCaml

2.3.1 `fun` for Currying and Anonymous Functions

`fun` is used to create anonymous functions and is inherently curried.

```
fun x y -> x + y;;
```

This is equivalent to:

```
fun x -> fun y -> x + y;;
```

2.3.2 `function` for Pattern Matching

`function` is primarily used for defining functions with pattern matching.

```
function
  | 0 -> -3
  | x -> x + 1;;
```

2.4 Defining Custom Types - Discriminant Unions

2.4.1 Option Type (`'a option`)

The `'a option` type is a predefined discriminant union in OCaml:

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

It represents a value that may or may not be present.

2.4.2 Custom List Type (`'a mylist`)

Defining a list type from scratch as a discriminant union:

```
type 'a mylist =
  | Empty
  | Nonempty of 'a * 'a mylist

let consmylist a l =
  Nonempty (a, l)
```

3 *<2025-01-14 Tue>* Functional Programming Introduction

3.1 Motivation for Functional Programming

3.1.1 Need for Better Software Development

- Current software development is often complex, leading to unreliable and slow programs.
- Desire for clarity and maintainability in code.

3.1.2 Clarity as a Primary Goal

- Functional programming aims for programs that are easier to write and maintain.
- Motivation since the 1970s, pioneered by John Backus (Fortran's creator).
- Problem with Imperative Languages (like Fortran, C, C++, Java, Javascript): Programs are recipes or sequences of commands.
  - Forces solutions to be expressed as long, thin sequences.
  - Not always the most natural or clear way to solve problems.

1. Lack of Referential Transparency in Imperative Languages **Example in C-like syntax:**

```
int a = F(B + 1);
// ... some code that might modify B or F's environment ...
int z = F(B + 1);
// Is 'a' necessarily equal to 'z'? - Maybe not in imperative languages.
```

Reasons for potential inequality:

- Function `F` might have side effects or depend on global state.
- Variable `B` might be modified between the two calls to `F`.

- Referential Transparency Definition: "Obvious" what every identifier refers to. Same expression always yields the same value.
- Imperative languages often sacrifice referential transparency, making programs harder to reason about and maintain.
- C++'s name itself ("C plus plus") implies non-referential transparency (due to the `++` operator which modifies state).

3.1.3 Performance and Optimization as a Motivation

- Functional programming aims for better performance through easier optimization.
- Goal: Compiler does more work to optimize, programmer focuses on problem.

Example in C:

```
int x = 0;
int y = 0;
int z = 0;
```

```
int F() {
  x = 12;
  y = F(x);
  return x + z;
}
```

Compiler Optimization Challenges in Imperative Code:

- Side effects make it difficult to optimize.
- Example: Compiler cannot assume `x` is still 12 after the call to `F(x)` because `F` (or functions it calls) might modify `x` or global state.
- Prevents optimizations like reusing computed values or reordering operations.
- Functional Programming's Potential Solution: Referential transparency enables more aggressive compiler optimizations.

3.2 Functional vs. Imperative vs. Logic Programming

3.2.1 Three Major Programming Paradigms

- 1. Imperative Languages: (e.g., C, Java, Javascript) - Focus on commands, statements, and step-by-step execution.
- 2. Functional Languages: (e.g., OCaml, Haskell) - Focus on functions, expressions, and evaluating functions.
- 3. Logic Languages: (e.g., Prolog) - Focus on describing problems and relationships, system finds solutions. (To be covered later).
- Functional and logic programming both aim to address clarity and performance issues of imperative programming, but in different ways.

3.2.2 Imperative Programming Paradigm

- Basic Unit: Command or Statement.
- Glue for Program Pieces: Sequencing (semicolon `;`). Statements executed in order.
- Relies on Variables with State: State modified through assignment statements.

3.2.3 Functional Programming Paradigm

- Basic Unit: Function.
- Glue for Program Pieces: Function Calls. Functions call other functions, return values used as arguments.
- Partial Order of Computation: Function calls create a partial order. `F2` and `F3` must complete before `F1` starts, but `F2` and `F3` can potentially run in parallel.
- Relies on Referential Transparency: Names always refer to the same value within their scope.
- Gives up Assignment Statements: Variables hold values, but values are immutable (don't change after initialization).
- Gives up Side Effects (Ideally): Functions ideally should not modify external state or have effects beyond returning a value.
- Functional Forms/Higher-Order Functions: Functions that take other functions as arguments or return functions as results. (e.g., Sigma operator, `Qsort` in C with a comparison function).

3.2.4 MapReduce Example (Google's Early Technology)

- Functional programming concept applied in a distributed, parallel setting.
- Breaks problems into smaller functions (`map`, `reduce`).
- Used for efficient query processing in distributed systems.
- Components implemented in C++, but the idea of combination from functional programming.

3.3 Questions and Clarifications During Lecture

3.3.1 Parallelism and Functional Programming

- Question: Are there cases where `F2` and `F3` in `F1(F2(), F3())` cannot be computed in parallel in functional programming?
- Answer: In a pure functional setting, ideally no (unless hardware limitations like single-core CPU). Functional evaluation provides a partial order, allowing potential parallelism. Imperative languages (Java, C) often impose stricter sequential evaluation rules.

3.3.2 Multithreading and Functional Programming

- Question: Wouldn't the sequential nature of function call stacks in out-of-order processors and multithreading negate the performance benefits of functional languages compared to imperative languages?
- Answer: Functional programming aims to avoid the complexities and performance bottlenecks of explicit thread synchronization with locks common in imperative multithreading. Functional approach promises no need for locks, reducing bugs and improving performance by avoiding lock contention.

3.3.3 OCaml vs. SML (Standard ML)

- Question: Why OCaml and not ML (or SML)? (Book uses SML).
- Answer:
  - Cultural preference in the department (UCLA CS).
  - OCaml is arguably more widely used in industry ("more real-worldish").
  - Practical hacks and features in OCaml (due to real-world use) that SML lacks, though this adds complexity.
  - Differences between OCaml and SML are not critical for this course's core functional programming concepts. Examples of syntax differences (list syntax, comparison operators, keywords).

3.4 Basic Properties of OCaml

3.4.1 Static Type Checking (Mostly)

- Types are checked at compile time, like C, C++, Java.
- Goal: Reliability. Catch type errors before runtime.
- Unlike dynamic languages like Python, Javascript.
- OCaml aims to be more of a production language than a scripting language.

3.4.2 Type Inference

- Types often do not need to be explicitly declared. OCaml infers types.
- Like Javascript, Python in this aspect, unlike traditional C, C++.
- C++, C++ and Java are evolving to incorporate type inference (e.g., `auto` keyword).

3.4.3 Automatic Storage Management (Garbage Collection)

- No manual memory management (`free`, `delete`). Garbage collector reclaims unused memory.
- Like Javascript, Python, Java, unlike C, C++, Fortran.
- Goal: Reliability and convenience, willing to trade some performance for it.

3.4.4 Good Support for Higher-Order Functions

- Functions as first-class citizens. Functions can be arguments and return values of other functions.
- Underlying feature that benefits from the other properties (static typing, type inference, GC).

3.5 Introduction to OCaml in Interpreter

3.5.1 Starting OCaml Interpreter

- Demonstration of starting the OCaml interpreter.
- Typing expressions followed by `;;` and Enter to evaluate.
- Interpreter output: `- : type = value` (for nameless expressions) or `name : type = value` (for named definitions with `let`).

3.5.2 Basic Expressions and `let` Bindings

- Arithmetic expressions (e.g., `3 + 4 * 5;;`).
- `let name = expression;;` for defining named values.
- Example: `let n = 10;;`, `let s = if n > 0 then "a" else "b";;`.

3.5.3 Type Checking and Type Errors

OCaml performs static type checking before execution.

- Type mismatch errors occur if types in an expression are inconsistent.
- Example of type error: `if n > 0 then "a" else 29;;` (incompatible types `string` and `int` in `then` and `else` branches).
- Example of type error: `if (n = 1369) = "a" then ...` (comparing boolean with string).
- Static type checking as both a friend (catching errors) and an opponent (strictness).

3.5.4 Lists in OCaml

- List syntax: `[v1; v2; v3; ...]`.
- Example: `let li = [3; 5; 7];;`, `let lb = [3; 5; 7; -1];;`.
- List type: `'a list` (e.g., `int list`). All elements in a list must be of the same type (homogeneous).
- List equality comparison: `li = lb;;` (compares list values, not just references).

3.5.5 Tuples in OCaml

- Tuple syntax: `(v1, v2, v3, ...)` or `v1, v2, v3, ...` (parentheses often optional).
- Example: `let ti = (1, 3, 5);;`, `let tj = (1, "hello", (3.14, "world"));;`.
- Tuple type: `'t1 * 't2 * 't3 ...` (e.g., `int * string * (float * string)`).
- Tuple components can be of different types (heterogeneous).
- Tuple type includes the number of components and types of each component.
- Tuple type is distinct from list type. `li = ti;;` (type error).

3.5.6 Unit Type in OCaml

- Degenerate tuple: `()` (empty tuple).
- Type: `unit`. Only one value: `()`.
- Used for functions that don't return a meaningful value (side-effecting functions in imperative style, though discouraged in functional style).
- Example of unit type usage (less relevant in purely functional context).

3.5.7 Type Variables (Polymorphism) with Empty List

- Empty list: `[]`.
- Type of empty list: `'a list` (list of 'anything'). Type variable `'a`.
- Type variable indicates the list can be a list of any type.
- Example: `let empty = [];;` (type is `'a list`).
- Empty list is type-compatible with lists of specific types (e.g., `int list`, `string list`).
- Example: Comparing `empty` (`'a list`) with `[1; 2; 3]` (`int list`) is allowed.
- Contrasts with tuples: Empty tuple `()` (unit) is not type-compatible with tuples of other types.

3.5.8 Equality Operators: `=` vs. `==`

- `=` (single equals): Structural equality. Compares values of data structures (lists, tuples, etc.).
- `==` (double equals): Physical equality. Compares if they are the same object in memory (more like pointer comparison, though no direct pointers in OCaml at user level).
- If `x == y` is true, then `x = y` is also true, but not vice versa.
- Example: Lists `x = [1; 2; 3]` and `y = [1; 2; 3]`. `x = y` is true (same values). `x == y` is false (likely different list objects in memory).

3.5.9 `if-then-else` with Lists

- Example: `if n > 0 then [] else [1; 2; 3];;` (allowed, both branches are lists, though potentially of different type due to type variable).

3.5.10 Type Conversion (Limited and Safe)

- OCaml performs limited, safe type conversion in some contexts (e.g., type variable instantiation).
- Example: `[]` (`'a list`) can be treated as `int list` when compared with `[1; 2; 3]`.
- Type conversion is not arbitrary or unsafe; it's guided by type system rules.

3.5.11 Function Definition in OCaml

- Function definition syntax: `let function_name argument = function_body;;` or `let function_name = fun argument -> function_body;;`.
- Example: `let f x = x + 1;;` or `let f = fun x -> x + 1;;`.
- Function type: `'argument_type -> 'return_type` (e.g., `int -> int`).

Every function in OCaml takes a single argument and returns a single result (currying concept).

- Simulating multiple arguments using tuples: `let g (x, y) = x + y;;` (function takes a tuple `int * int` as one argument).
- Functions returning tuples: `let h x = (x, x + 5);;` (function returns a tuple `int * int`).
- Parentheses in function definitions and calls are often optional in OCaml, especially for single arguments.

3.5.12 Currying - Functions Returning Functions

- Example: `let ocons head = fun tail -> head::tail;;` (or shorter syntax `let ocons head tail = head::tail;;`).
- `ocons` is a function that takes a `head` and returns another function.
- Returned function takes a `tail` (list) and returns a new list with `head` prepended.
- Currying motivates single-argument functions for simplicity and flexibility.
- Function calls in OCaml are left-associative: `F G H` is parsed as `(F G) H`.
- Function types are right-associative: `a -> b -> c` is parsed as `a -> (b -> c)`.

3.5.13 `cons` vs. `ocons` - Tuple Argument vs. Curried

- `cons head_tail_tuple = ...` : Function takes a tuple as a single argument (head and tail bundled together).
- `ocons head = fun tail -> ...` : Curried function. Takes `head` first, returns a function that then takes `tail`.
- Curried functions (`ocons`) are more flexible: You can partially apply them (provide only the `head` initially and get a function to use later with different tails).

3.5.14 `car` (Head) Function - Issue of Partiality

- Attempt to define `car` (head) function using pattern matching:

```
let car list =
  match list with
    head :: tail -> head
```

or:

```
let cat (head :: tail) = head
```

- Warning message: "Warning 8: this pattern-matching is not exhaustive..."
- Warning indicates potential runtime errors: Function is not defined for all possible inputs (specifically, the empty list `[]`).
- Runtime error when calling `car []`: `Exception: Match_failure ...`.
- Rule: Avoid functions with potential runtime errors. Aim for robust, total functions (defined for all valid inputs of their type).

Next class will discuss how to handle cases like `car` function properly in a functional style (avoiding runtime errors and warnings).

4 *<2025-01-16 Thu>* Patterns, Recursion, and User-Defined Types in OCaml

4.1 Patterns in OCaml - Taking Apart Data Structures

4.1.1 Match Expression

- Basic Syntax:

```
match expression with
| pattern1 -> expression1
| pattern2 -> expression2
| ...
| patternN -> expressionN
```

- Evaluation: Evaluates `expression`, then tries to match its value against `pattern1`, `pattern2`, etc. The first matching pattern's corresponding `expression` is evaluated and its value is returned.
- Runtime Error: If no pattern matches, a runtime error occurs.

4.1.2 Common Patterns

1. Constant Pattern
   - Matches itself.
   - Example: `0`, `"a"`, `true`.

   ```
   match value with
   | 0 -> (* ...code for value being 0... *)
   | _ -> (* ...default case... *)
   ```

2. Identifier Pattern
   - Single identifier (e.g., `x`, `value`).
   - Matches any value.
   - Binds the identifier to the matched value.

   ```
   match value with
   | x -> (* ...code using 'x' bound to 'value'... *)
   ```

3. Underscore Pattern (`_`) - Don't Care
   - Matches any value, but discards it.
   - Identifier `_` is not bound.
   - Used for default cases or when a value is irrelevant.

```
match value with
| pattern1 -> expression1
| _ -> (* ...default case, value is ignored... *)
```

- Analogy to `default` in C switch statements for fall-through cases.

4. Parenthesized Pattern `(p)`
   - Matches whatever pattern `p` matches.
   - Used for grouping or clarity, often optional.
5. Tuple Pattern `p1, p2, ..., pn`
   - Matches n-tuples.
   - Each component `pi` must match the corresponding element of the tuple.

```
match tuple_value with
| p1, p2 -> (* ...code using components matched by p1 and p2... *)
```

6. List Pattern `[p1; p2; ...; pn]`
   - Matches lists of length `n`.
   - Each `pi` must match the corresponding element of the list.
   - Rarely used directly in recursive list processing, `::` pattern is more common.

```
match list_value with
| [p1; p2; p3] -> (* ...code for 3-element list... *)
```

7. Cons Pattern `p1 :: p2`
   - Matches non-empty lists (length >= 1).
   - `p1` matches the head (first element) of the list.
   - `p2` matches the tail (rest of the list, which is itself a list).
   - Crucial for recursive list processing.

```
match list_value with
| head :: tail -> (* ...code using 'head' and 'tail'... *)
| [] -> (* ...code for empty list... *)
```

   - Type Constraints: List patterns enforce homogeneity. If `p1` matches type `'a`, `p2` must match type `'a list`.
   - Equivalence: `[p1; p2; p3]` is equivalent to `p1::p2::p3::[]` due to right-associativity of `::`.

4.1.3 Syntactic Sugar: `let pattern = expression` as `match expression with pattern -> ...`

- `let (a, b) = tuple_expression` is syntactic sugar for:

```
let cons x =
  match x with
  | (a, b) -> a :: b
```

- Vertical bar `|` before the first pattern in `match` is optional but often used for consistency.

4.2 Recursive Functions and `rec` Keyword

4.2.1 `rec` Keyword for Recursive Definitions

- In OCaml, identifiers cannot be defined in terms of themselves by default.

`let identifier = expression` - `identifier` in `expression` refers to an already defined identifier, not the one being defined.

- Recursive function definitions require the `rec` keyword: `let rec function_name ... = ... function_name ...`.
- `rec` allows the function's name to be used within its own definition for recursive calls.
- Example: Recursive `eo` (every other element) function.

4.2.2 Example: `eo` (Every Other Element) Function (Improved Version)

- Initial (longer) version:

```
let rec eo list =
  match list with
  | [] -> []
  | h :: _ :: t -> h :: eo t
  | singleton_list -> singleton_list (* Default case for single element list *)
```

- Simplified (shorter) version, leveraging pattern exhaustiveness:

```
let rec eo list =
  match list with
  | [] -> []
  | h :: _ :: t -> h :: eo t
  | last_element -> last_element (* Default case, must be singleton *)
```

- Further simplified by using underscore for the last pattern (don't care about name):

```
let rec eo list =
  match list with
  | [] -> []
  | h :: _ :: t -> h :: eo t
  | _ -> list (* Default case, must be singleton list *)
```

- Even shorter, by directly returning the list in the default case:

```
let rec eo list =
  match list with
  | [] -> []
  | h :: _ :: t -> h :: eo t
  | list -> list (* No functional difference, stylistic choice *)
```

4.3 `function` Keyword - Syntactic Sugar for Pattern Matching Functions

4.3.1 `function` as Shorthand for `fun argument -> match argument with ...`

- `let function_name = function` is equivalent to `let function_name = fun argument -> match argument with argument`.
- Used when a function immediately pattern-matches its single argument.
- Code Example: `ep` (Every Other - Function version):

```
let ep = function
  | [] -> []
  | h :: _ :: t -> h :: ep t
  | list -> list
```

- `function` is syntactic sugar for conciseness, especially in pattern-matching heavy code.

4.4 Anonymous Functions - `fun` Keyword

4.4.1 `fun` Keyword for Nameless Functions (Lambda Expressions)

- `fun argument -> expression` creates an anonymous function.
- Functions are first-class citizens - can be created and used without names.
- Example: `fun x -> x + 1` is a function that adds 1 to its input.
- Functions can be returned as results from other functions.
- Example: `nnn` function returning different functions based on input.

4.4.2 Example: `nnn` - Function Returning Functions

```
let nnn = function
  | 0 -> (fun x -> x)  (* Identity function if input is 0 *)
  | _ -> (fun _ -> 1)  (* Function always returning 1 otherwise *)
```

- Illustrates functions returning other functions. `nnn 0` returns the identity function. `nnn 5` returns a function that always returns `1`.

4.5 `fun` vs. `function` - Key Distinction

4.5.1 `fun` - Low-Level, Currying

- Basic function definition keyword.
- Primarily for creating functions with a single argument and a body expression.
- Supports currying implicitly.
- General-purpose function creation.

4.5.2 `function` - Pattern Matching, Concise

- Syntactic sugar for functions that immediately pattern-match their argument.
- Makes code more concise when pattern matching is the primary purpose.
- Typically used when at least two or more patterns are needed (otherwise `fun` might be simpler).

4.6 Generic Minimum Function - `gmin` - Higher-Order Functions and Generality

4.6.1 Generalizing `min` Function for Different Types

- Problem with initial `min` function: Hardcoded for integers and uses a specific "infinity" value (-99999999999999).
- Goal: Create a generic `min` function that works with any type and uses a user-provided comparison function and identity value.

4.6.2 `gmin` Function - Taking Comparison and Identity Value as Arguments

```
let rec gmin lt id_val = function
  | [] -> id_val
  | h :: t ->
      let t_min = gmin lt id_val t in
      if lt h t_min then h else t_min
```

- `gmin` takes three arguments (curried):
  - 1. `lt`: A "less than" comparison function (e.g., `<` for integers).
  - 2. `id_val`: An identity value (returned for empty list, e.g., a very large number for min).
  - 3. An input list.
- Higher-Order Function: `gmin` takes a function (`lt`) as an argument.
- Flexibility: `gmin` can be used with different types and comparison criteria by providing appropriate `lt` and `id_val`.

4.6.3 Example Usage of `gmin` with Integers

```
let imin = gmin (<) 99999999999999;; (* Create integer-specific min function *)
imin [5; -12; 15; 19];; (* Call imin *)
```

4.7 User-Defined Types - Discriminant Unions

4.7.1 `type` Keyword for Defining New Types

- `type type_name = type_definition`
- Simple Type Alias: `type type_name = existing_type` (e.g., `type int_alias = int`).
- Discriminate Union Type Definition: `type type_name = Constructor1 | Constructor2 of type2 | Construct`
  `of type3 ....`
- Constructors (e.g., `Foo`, `Bar`, `Baz`) start with capital letters by convention.

4.7.2 Discriminate Unions vs. C/C++ `union`

- OCaml Discriminate Unions (Safe):
  - Tagged unions: Each value carries a tag indicating its constructor/variant.
  - Type-safe: Compiler enforces correct usage based on the tag. Pattern matching is the primary way to access variants.
  - Prevents errors like accessing the wrong union member.
- C/C++ `union` (Error-Prone):
  - Untagged unions: Memory location can hold values of different types, but no tag to indicate the current type.
  - Programmer responsibility to track the active type.
  - Error-prone: Accessing the wrong member leads to undefined behavior and potential crashes.

4.7.3 Example: `my_dtype` Discriminate Union

```
type my_dtype =
  | Foo
  | Bar of int
  | Baz of int * string
```

- `Foo`: Constructor with no associated data.
- `Bar of int`: Constructor `Bar` takes an `int` payload.
- `Baz of int * string`: Constructor `Baz` takes an `int * string` tuple payload.

4.7.4 Pattern Matching on Discriminate Unions

```
let process_dtype value =
  match value with
  | Foo -> 1
  | Bar n -> n + 7
  | Baz (n, _) -> n - 3
```

- Pattern matching is the standard way to work with discriminate unions.
- Compiler ensures exhaustiveness and type safety in pattern matching.

4.7.5 Generic Discriminate Union: `option` Type

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

- `'a option` is a predefined generic type in OCaml.
- `None`: Constructor representing absence of a value (like null, but type-safe).
- `Some of 'a`: Constructor `Some` holds a value of type `'a`.
- Used to represent optional values, handle cases where a value might be missing, and avoid null pointer exceptions.

4.7.6 Example: Custom List Type (Conceptual)

```
type 'a my_list =
  | My_Empty
  | My_Nonempty of 'a * 'a my_list
```

- Conceptual definition of a list as a discriminant union. OCaml's built-in `list` is similar internally.
- `My_Empty`: Constructor for the empty list.
- `My_Nonempty of 'a * 'a my_list`: Constructor for non-empty list, holding a head element of type `'a` and a tail (recursive `my_list`).

4.8 Reading Assignment for Next Class

- Read chapters 1-7, 9, and 11 of the textbook, focusing on syntax and ML concepts.

5 *<2025-01-21 Tue>* Syntax - Form vs. Meaning in Language

5.1 Why Syntax Matters

5.1.1 Syntax as Form, Semantics as Meaning

- Syntax: Form of programs, easier to define and analyze.
- Semantics: Meaning of programs, more complex to specify.
- Focus on Syntax First: Syntax is a prerequisite for understanding semantics.

5.1.2 Syntax - A Success Story in Computer Science

- 1960s Discovery: Deep connection between programming language syntax, human language structure, and formal systems.
- Respectability of Computer Science: Formal syntax provided a rigorous foundation, moving CS beyond "hacking."
- Connection to Linguistics: Syntax linked CS to formal linguistics, enhancing academic credibility.

5.1.3 Syntax for Specification vs. Implementation

- Syntax as Specification: Formal syntax allows precise, unambiguous definition of what a program is (structure), without detailing how it's executed.
- Analogy: CS32 Assignment Spec vs. Code: Specification defines input/output, constraints; code is the implementation.
- Formal Syntax is Ironclad and Complete: Unlike informal English descriptions, formal syntax is mathematically rigorous and covers all aspects of program structure.
- Ambition of Formal Specification: Aim to define "what we want" (syntax) formally, before implementation.

5.2 Criteria for Good Syntax

5.2.1 Inertia (Familiarity) vs. Simplicity

- Competing Goals in Syntax Design.

1. Inertia (Familiarity)
   - Syntax close to existing notations (e.g., mathematical conventions) is easier for humans to learn and use.
   - Example: C-style infix notation `(a + b) * c` is familiar from mathematics.
2. Simplicity
   - Syntax that is easy to parse, compile, and process by machines.
   - Examples of Simpler Syntaxes:
     - PostScript Syntax: Reverse Polish notation `a b + c *` (stack-based, very simple to parse).
     - Lisp Syntax: Prefix notation `(* (+ a b) c)` (parenthesis-based, unambiguous, simple grammar).
   - Trade-off: Simpler syntaxes might be less familiar/readable to humans initially.

5.2.2 Unambiguity

- A good syntax should be unambiguous.

Each syntactically valid program should have only one interpretation (parse tree, meaning).

- Avoids confusion and ensures consistent program behavior.
- Example of Ambiguity in C++: `a = b+++++c;` (multiple possible parsings).
- Ambiguity is undesirable in programming languages (unlike diplomacy or poetry).
- Formal syntax specifications (grammars) help resolve or prevent ambiguity.

5.2.3 Readability and Writability

- Syntax should be easy to read and understand by humans (readability).
- Syntax should be easy to write and express intended programs (writability).
- Readability vs. Writability Trade-off: Highly concise syntax (e.g., APL one-liners) can be writeable but less readable/maintainable.
- Importance of Readability for Software Maintenance: Code is read more often than written, readability is crucial for long-term maintainability and collaboration.

5.2.4 Redundancy

- Redundant syntax can enhance reliability by enabling early error detection.
- Redundancy means including extra syntactic information that is not strictly necessary but helps catch errors.
- Example of Lack of Redundancy (Fortran 1956): Implicit type declarations based on identifier names (I-N for `int`, others for `float`).
- Problem with Fortran's Implicit Declarations: Misspelled variable names were silently treated as new variables (not type errors), leading to bugs (e.g., rocket failure due to misspelled variable).
- Redundancy for Error Detection: Redundant syntax (e.g., explicit type declarations, mandatory semicolons) helps compilers catch errors and prevent undefined behavior.
- Trade-off: Redundancy can make code slightly longer to write, but improves reliability.

5.3 Grammars - Formal Syntax Specification

5.3.1 Grammar as Syntax Specification

- Formal grammar: A precise, mathematical way to define the syntax of a language.
- Grammar specifies the valid structures (form) of programs in a language.

5.3.2 Tokens - Basic Units of Syntax

- Tokens: Individual, meaningful units of a language's syntax (like words in English, but more precisely defined).
- Examples of Tokens in Programming Languages: Identifiers, keywords, operators, literals (numbers, strings), punctuation.
- Tokenization/Lexical Analysis: Process of breaking character input into a sequence of tokens.
- Not every character sequence is a valid token sequence (e.g., unterminated string).

5.3.3 Characters - Building Blocks of Tokens

- Characters: Atomic symbols from which tokens are built.
- Character Encoding: Characters are represented as byte sequences (e.g., UTF-8 encoding for Unicode characters).
- Character Sequence -> Token Sequence: Lexical analysis stage transforms a stream of characters into a stream of tokens.

5.4 Example: Email Message-ID Grammar (RFC 5322)

5.4.1 RFC 5322 - Internet Standard for Email Format

- Formal specification for email message format, including headers and body.
- Grammar in RFC 5322 defines valid email syntax.
- Example Grammar Rule: `message-id = "Message-ID:" msg-id CRLF`.
  - LHS (Left-Hand Side): `message-id` (non-terminal symbol).
  - RHS (Right-Hand Side): `"Message-ID:"` (terminal symbol - literal string), `msg-id` (non-terminal), `CRLF` (terminal - carriage return line feed).

5.4.2 Grammar Rules - Non-terminals and Terminals

- Grammar Rule: Defines how a non-terminal symbol is composed of a sequence of terminal and non-terminal symbols.
- Non-terminal Symbols (Internal Nodes in Parse Tree): Represent abstract syntactic categories (e.g., `message-id`, `msg-id`). Can be further expanded by grammar rules. Represented as ovals in parse tree diagrams.
- Terminal Symbols (Leaves in Parse Tree): Basic, indivisible symbols of the language (e.g., `"Message-ID:"`, `<`, `@`, `CRLF`). Cannot be further expanded. Represented as rectangles in parse tree diagrams.
- Start Symbol: Top-level non-terminal symbol (e.g., `message-id` might be part of a larger email grammar).

5.4.3 BNF (Backus-Naur Form) - Simple Grammar Notation

- Formal system for defining grammars using rules with LHS and RHS.
- Grammar as Specification for Parsing: Grammar rules can be used to build parsers that check if input conforms to the syntax and to construct parse trees.

5.4.4 EBNF (Extended Backus-Naur Form) - Shorthand for BNF

- EBNF extends BNF with meta-symbols for convenience and conciseness, without adding expressive power.
- EBNF Meta-Symbols in RFC 5322 Example:
  - `|` (Vertical Bar): Or (alternation). `dot-atom-text | obs-id-left` means either `dot-atom-text` OR `obs-id-left`.
  - `[ ... ]`: Optional. `[ CFWS ]` means `CFWS` is optional (0 or 1 occurrence).
  - `*`: Repetition (zero or more times). `*dtext` in `no-fold-literal = "[" *dtext "]"` means zero or more `dtext`.

5.4.5 Example EBNF Rule Breakdown: `no-fold-literal = "[" *dtext "]"`

- `no-fold-literal`: Non-terminal being defined.
- `=`: "Is defined as".
- `"["`: Terminal symbol - literal opening square bracket.
- `*dtext`: Zero or more occurrences of the non-terminal `dtext`.
- `"]"`: Terminal symbol - literal closing square bracket.

5.4.6 Translation EBNF to BNF - Example `no-fold-literal`

- EBNF Repetition `*` to BNF Recursion: EBNF repetition constructs like `*` and `+` can be rewritten using recursion in BNF.
- Example: `no-fold-literal` EBNF to BNF:

```
EBNF: no-fold-literal = "[" *dtext "]"
BNF Equivalent (using recursion):
sdtext = dtext sdtext | ""  // sdtext is a new non-terminal for "sequence of dtext"
no-fold-literal = "[" sdtext "]"
```

- `sdtext` is a new non-terminal introduced to represent "sequence of dtext" (zero or more `dtext`).
- BNF is more verbose but equally powerful as EBNF.

5.4.7 XML 1.1 Grammar Example - EBNF in Practice

- XML Grammar Example Rule: `element ::= EmptyElemTag | STag content ETag`
- XML Grammar Meta-Symbols (Example):
  - `::=`: "Is defined as" (similar to `=`).
  - `|`: Or (alternation).
  - `?`: Optional (0 or 1 occurrence).
  - `( ... )`: Grouping.
  - `*`: Repetition (zero or more times).
  - `[a-zA-Z]`: Character class (any alphabetic character).
  - `[^<>]`: Negated character class (any character **except** `<` and `>`).
- Lowercase Non-terminals: By convention, lowercase names in XML EBNF often indicate recursive non-terminals (like `element` and `content`).

5.4.8 Regular Grammars vs. Context-Free Grammars - Power and Limitations

1. Regular Grammars and Regular Expressions
   - Regular Grammars: Less powerful than context-free grammars (BNF/EBNF).
   - Regular Grammars are equivalent to Regular Expressions.
   - Example: Email Message-ID grammar (mostly regular).
   - Advantage of Regular Grammars/Regexps: Efficient parsing (e.g., `grep` speed).
2. Context-Free Grammars (BNF/EBNF)
   - Context-Free Grammars (CFGs): More powerful than regular grammars.

- BNF and EBNF are notations for context-free grammars.
- CFGs can handle recursion and nested structures, which regular grammars cannot.
- Example: Grammar for balanced parentheses: `S ::= "(" S ")" | "a"` - Not regular, requires CFG.
- Example: XML Grammar: Context-free (due to nested tags).
- Limitation of Regular Grammars/Regexps: Cannot handle balanced parentheses or arbitrary nesting (cannot "count").
- Recursion in Grammars: Recursion (non-terminal rules that can lead back to themselves) is what makes grammars context-free and more powerful than regular grammars. Recursion also introduces potential parsing complexity and infinite loops if not handled carefully.
- Example of Recursion Issue: Grammar rule `A ::= A | 'a'` leads to infinite recursion.

6 *<2025-01-23 Thu>* ISO Standard for EBNF - Extended BNF and Grammar Issues

6.1 ISO Standard for EBNF - Variety of EBNF Notations

6.1.1 Need for Standardization

- Many variations of EBNF exist, leading to potential confusion and interoperability issues.
- ISO (International Standards Organization) defined a standard for EBNF to promote consistency.
- ISO EBNF Standard (ISO/IEC 14977): Attempts to create a unified and rigorous EBNF notation.

6.1.2 ISO EBNF Notation - Key Features

- Terminal Symbols: Enclosed in single or double quotes (e.g., `"terminal"`, `'terminal'`).
- Optional Parts: `[ option ]` (square brackets). Encloses optional elements (0 or 1 occurrence).
- Repetition (Zero or More): `{ repetition }` (curly braces). Encloses elements that can repeat zero or more times. Also `*` prefix operator.
- Grouping: `( grouping )` (parentheses). Groups elements for applying operators.
- Exception Operator: `-` (minus sign). `A - B` means "A but not B" (set difference of languages). Semantically complex and not commonly used in EBNF.
- Concatenation Operator: `,` (comma). `A , B` means A followed by B. Explicit concatenation operator, unlike implicit concatenation by juxtaposition in some EBNF variants.
- Alternation (Or): `|` (vertical bar). `defn | defn` means either `defn` or `defn`.
- Rule Definition: `LHS = RHS ;` (Left-Hand Side equals Right-Hand Side, semicolon terminator).

6.1.3 Example: EBNF Grammar for EBNF itself (ISO EBNF in ISO EBNF)

- Meta-Syntax Example: Defining the syntax of EBNF using EBNF itself (meta-grammar).

```
syntax         = syntax rule , { syntax rule } ;
syntax rule    = meta id , '=' , defns list , ';' ;
defns list     = defn , { '|' , defn } ;
defn           = term , { ',' , term } ;
term           = ... ; (* More rules to complete the grammar of EBNF *)
```

- Example Rule Breakdown: `syntax = syntax rule, { syntax rule } ;`:

- `syntax`: Non-terminal representing a complete EBNF syntax (grammar).
- `=`: "Is defined as".
- `syntax rule`: Non-terminal representing a single syntax rule.
- `,`: Concatenation - followed by.
- `{ syntax rule }`: Repetition - zero or more `syntax rule` s.
- `;`: Terminal symbol - rule terminator.

- Self-Definition and Bootstrapping: EBNF is powerful enough to describe its own syntax (meta-circularity).

6.2 What Can Go Wrong with Grammars - Issues and Challenges

6.2.1 Tokenization Issues (Recap from Previous Lecture)

- Character Confusion: Visually similar but distinct characters (e.g., Cyrillic 'o' vs. Latin 'o').
- Reserved Words: Conflicts between keywords and identifiers (e.g., `int class = 12;`).
- Case Sensitivity: Handling case-insensitive languages or parts of languages (e.g., URLs).
- Greedy Tokenization: Tokenizers always matching the longest possible token (e.g., `a = b+++++c;`).
- Long Tokens: Handling very long tokens (e.g., long strings, identifiers).
- Comments and Whitespace: Correctly ignoring comments and whitespace during tokenization.

6.2.2 Grammar Issues - Problems with Grammar Definitions

1. Useless Rules
   - Unreachable Non-terminals: Non-terminals that cannot be derived from the start symbol. Example: `S -> A; T -> B;` (if `S` is start symbol, `T` is unreachable).
   - Blind Alley Rules: Rules that lead to derivations that cannot produce terminal strings (sentences). Example: `S -> a; S -> bT; T -> cT;` (`T` can only derive more `T` s and 'c's, never terminating).
   - Redundant Rules: Rules that are unnecessary as they don't add to the language defined by the grammar. Example: `S -> a; S -> bS; S -> ba;` (`S -> ba` is redundant).
   - Impact of Useless/Redundant Rules: Grammars become more complex and harder to understand, but language defined remains the same.
2. Extra-Grammatical Constraints
   - Grammars (BNF/EBNF) are context-free grammars (CFGs). CFGs have limitations.
   - Context-Free Grammars cannot capture all syntactic rules of programming languages.
   - Example: Declaration before Use: CFGs cannot enforce that variables must be declared before being used (semantic rule, not purely syntactic).
   - Example: Type Checking: CFGs cannot enforce type correctness (semantic rule).
   - Workaround: Side Notes in English: Programming language standards often use English prose to specify constraints that CFGs cannot express (semantic rules, context-sensitive rules).
3. Ambiguity
   - Ambiguous Grammar: A grammar that allows more than one parse tree (derivation) for the same input sentence (token sequence).
   - Undesirable in Programming Languages: Ambiguity leads to uncertainty about program meaning and behavior. Compilers might interpret code differently than intended.
   - Example of Ambiguous Grammar (Arithmetic Expressions):

```
S ::= id | num | S "+" S | S "*" S | "(" S ")"
Example ambiguous sentence: id "+" num "*" id
Two possible parse trees (different precedence):
(id "+" num) "*" id  vs.  id "+" (num "*" id)
```

6.2.3 Resolving Ambiguity - Precedence and Associativity

- Precedence Rules: Define the order of operations (e.g., multiplication before addition).
- Associativity Rules: Define grouping for operators of the same precedence (e.g., left-associativity for subtraction: `a - b - c` is `(a - b) - c`).
- Techniques to Eliminate Ambiguity in Grammars:
  - Rewrite the Grammar: Modify the grammar structure to enforce precedence and associativity directly in the rules (more complex grammar). Example: Layered grammar for arithmetic expressions.

```
E ::= E "+" T  |  T
T ::= T "*" F  |  F
F ::= id   |  num  |  "(" E ")"
```

- Use Disambiguation Rules (Outside Grammar): Keep the grammar simpler (possibly ambiguous) but add external rules to resolve ambiguity during parsing (e.g., "else binds to the nearest if" rule for dangling-else problem).

6.2.4 Dangling-Else Ambiguity (Example of Ambiguity in C-like Languages)

- Problem: Ambiguity in `if-else` statements when nested.

```
if (condition1)
  if (condition2)
    statement1;
  else
    statement2; // To which 'if' does this 'else' belong?
```

- Two Possible Interpretations (Parse Trees):
  - 1. `else` associated with the inner `if`: `if (condition1) { if (condition2) statement1; else statement2; }` (Correct interpretation in C-like languages).
  - 2. `else` associated with the outer `if`: `if (condition1) { if (condition2) statement1; } else statement2;` (Incorrect interpretation).
- Ambiguity Resolution Rule (in C-like languages): "Dangling `else` associates with the nearest preceding `if` that lacks an `else`." (Lexical scoping rule, not directly in grammar).
- Grammar Fix for Dangling-Else (Complicating Grammar): Introduce new non-terminals to enforce the correct association in the grammar itself (more complex, less readable grammar). Example: Distinguish between "matched" and "unmatched" if statements in grammar rules.

```
stmt ::= <if-stmt> | s1 | s2
<full-stmt> ::= <full-if> | s1 | s2
<full-if> ::= if <expr> then <full-stmt> else <full-stmt>
<if-stmt> ::= if <expr> then <full-stmt> else <stmt>
            | if <expr> then <stmt>
<expr> ::= e1 | e2
```

6.3 Parsing - From Grammars to Parsers

6.3.1 Parsing Problem: Deriving Parse Trees from Token Sequences

- Parsing: The process of taking a sequence of tokens (program code) and determining its syntactic structure according to a grammar.
- Goal of Parsing: To construct a parse tree (abstract syntax tree - AST) that represents the syntactic structure of the input code, or to detect syntax errors if the input is not valid according to the grammar.

6.3.2 Three Main Issues in Parsing Implementation

1. Recursion in Parsers
   - Grammars are often recursive (non-terminals defined in terms of themselves).
   - Parsers need to handle recursion to parse arbitrarily nested structures.
   - Recursion Handling in Parsers (Relatively Easy): Recursive descent parsing is a common technique that directly implements grammar rules as recursive functions.
2. Disjunction (OR) in Grammars
   - Grammars use alternation (`|`) to define choices (multiple possible rules for a non-terminal).
   - Parsers need to handle disjunction by exploring different parsing paths when encountering alternatives in the grammar.
   - Disjunction Handling in Parsers (Requires Backtracking or Alternatives): Parsers may need to try different alternatives and backtrack if a choice leads to a dead end. Matchers and Acceptors (in Homework 2) are designed to handle disjunction.
3. Concatenation (Sequence) in Grammars
   - Grammar rules define sequences of symbols (concatenation - `,` or implicit juxtaposition).
   - Parsers need to process input tokens sequentially according to the order defined in grammar rules.
   - Concatenation Handling in Parsers (Sequential Processing): Parsers read input tokens in order, matching them against the expected sequence of symbols in grammar rules. Matchers and Acceptors handle concatenation by sequential matching.

6.3.3 Matchers and Acceptors - Approach for Parsing (Homework 2 Hint)

- Matchers: Functions that try to match a prefix of the input token sequence against a part of a grammar rule. Return the matched part and the remaining (unmatched) suffix.
- Acceptors: Functions that check if the remaining suffix (from a matcher) is "acceptable" according to the rest of the grammar rule.
- Matchers and Acceptors Work Together: Matcher matches a part, Acceptor validates the rest, combining to parse larger structures.
- Homework 2 Focus: Using Matchers and Acceptors to handle disjunction and concatenation in parsing.

6.4 Reading Assignment for Next Class

- Continue reading textbook chapters as assigned previously.
- Review Homework 2 and prepare questions for discussion section.

7 <2025-01-28 Tue> Parsing - Matchers and Compiler Stages

7.1 Parsing - Matchers and Acceptors - Functional Approach to Parsing

7.1.1 Question from Previous Lecture - Overhead of Recursive Function Calls in OCaml

Recursive function calls in OCaml have some overhead, but it's generally not excessive in terms of memory or time.

- Garbage Collection Overhead: More significant performance factor in OCaml compared to manual memory management in languages like C/C++. OCaml prioritizes safety and convenience over raw speed.
- Closures Overhead: Closures (functions returning functions) can introduce some additional overhead.
- Trade-off: OCaml makes trade-offs for safety and expressiveness, sometimes at the expense of raw performance compared to highly optimized C/C++ code.

7.1.2 Matchers and Acceptors - Functional Parsing Technique

- Matchers and acceptors are functions used to build parsers in a functional style.
- Simpler than Full Parsers: Focus on syntax validation (yes/no) and basic parsing, not full parse tree construction or semantic analysis (in this lecture).

1. Form 0 Matcher - Boolean Matcher
   - Simplest Matcher: Function from input token sequence (fragment) to a Boolean (`true` if match, `false` if no match).
   - Language as Set of Sentences: Programming language defined as a set of valid token sequences (sentences).
   - Matcher as Membership Test: Form 0 matcher checks if an input token sequence is a member of the language (valid syntax).
   - Limitation: Too simple for practical parsing, doesn't provide information about parse structure or backtracking.
2. Form 0.1 Matcher - Grammar to Matcher Function
   - Higher-Order Function: Function that takes a grammar as input and returns a matcher function.
   - `make_matcher : grammar -> matcher` (where `matcher = fragment -> bool`).
   - Still limited: Doesn't handle backtracking for disjunction (OR) efficiently.
3. Form 0.2 Matcher - Matcher with Acceptor Argument
   - Matcher with Acceptor: `matcher : acceptor -> fragment -> bool`
   - Acceptor: Function that constrains what a matcher should accept. `acceptor : fragment -> bool`. Tells the matcher if a partial match is acceptable in the larger context.
   - Backtracking Enablement: Acceptors enable backtracking. If a matcher's initial attempt fails, it can backtrack and try other alternatives, guided by acceptors.
4. Form 0.4 Matcher - Matcher Returning Fragment Option
   - Matcher Returning Fragment Option: `matcher : acceptor -> fragment -> fragment option`
   - Fragment Option: `'a option = None | Some of 'a`. `None` indicates match failure, `Some frag'` indicates success and returns the unmatched suffix (`frag'`) of the input fragment.
   - More Informative Matcher: Returns not just success/failure (Boolean), but also the remaining input after a successful match.
   - Chaining Matchers with Acceptors: Matchers can be chained together using acceptors. A matcher calls another matcher with an acceptor that embodies the expectation of what should follow in the input.

7.1.3 Acceptors - Defining Acceptability

1. `accept_none` - Reject All Fragments

```
let accept_none = fun _fragment -> None
```

   - Acceptor that always returns `None`, rejecting any fragment.
   - Represents failure or non-acceptance.

2. `accept_all` - Accept All Fragments

```
let accept_all = fun fragment -> Some fragment
```

   - Acceptor that always returns `Some fragment`, accepting any fragment as valid.
   - Represents unconditional acceptance.

3. `accept_nonempty` - Accept Non-Empty Fragments

```
let accept_nonempty = function
  | [] -> None
  | f -> Some f
```

   - Acceptor that accepts only non-empty fragments, rejects empty fragments.

7.1.4 Matchers - Implementing Match Logic

1. `match_empty` - Match Empty String/Fragment

```
let match_empty acceptor fragment =
  acceptor fragment
```

   - Matcher for the empty string (epsilon match).
   - Always succeeds (matches empty prefix).
   - Action: Directly calls the provided `acceptor` on the original fragment (no input consumed by empty match).
   - Identity Function Analogy: `match_empty` is essentially the identity function in the context of acceptors. `match_empty = fun acc frag -> acc frag = fun acc -> acc`

2. `match_nothing` - Match Nothing, Always Fail

```
let match_nothing _acceptor _fragment = None
```

   - Matcher that always fails to match, regardless of input or acceptor.
   - Always returns `None`.
   - Represents failure or impossible match.

3. `match_anything` - Match Anything, Always Succeed (Iffy)

```
let match_anything _acceptor fragment = Some fragment
```

   - Matcher that always succeeds and matches the entire input fragment.
   - Ignores the acceptor (potentially problematic - "iffy").

   Returns `Some fragment` (original fragment as suffix).

4. `make_ts_matcher` - Match Terminal Symbol

```
let make_ts_matcher ts = fun acceptor -> function
  | [] -> None (* Fail on empty fragment *)
  | h :: t -> if h = ts then acceptor t else None (* Match head against ts, accept tail if match *)
```

- Matcher for a specific terminal symbol `ts`.
- Takes a terminal symbol `ts` as argument and returns a matcher function.
- Matcher Function Behavior:
  - Fails (returns `None`) on empty input fragment.
  - If input is non-empty, checks if the head (`h`) of the fragment matches the target terminal symbol `ts`.
  - If head matches, calls the provided `acceptor` on the tail (`t`) of the fragment (remaining input after matching `ts`). Returns the result of the `acceptor` call.
  - If head does not match `ts`, fails (returns `None`).

7.1.5 Combining Matchers - Disjunction (OR) - `make_disj_matcher`

```
let rec make_disj_matcher (ma, mb) = fun acceptor -> fun frag ->
  match ma acceptor frag with
  | Some _ as r -> r (* If ma succeeds, return its result *)
  | None -> mb acceptor frag (* If ma fails, try mb *)
```

- Matcher for disjunction (OR) of two patterns (represented by matchers `ma` and `mb`).
- Takes two matchers (`ma`, `mb`) as input and returns a new matcher.
- Disjunction Matcher Behavior:
  - Tries to match using the first matcher `ma`.
  - If `ma` succeeds (returns `Some _`), `make_disj_matcher` immediately returns the result of `ma` (success). First-match wins behavior.
  - If `ma` fails (returns `None`), `make_disj_matcher` tries the second matcher `mb`. Returns the result of `mb` (success or failure of `mb`).
  - Backtracking Implementation: If `ma` fails, it backtracks and tries `mb`.

7.1.6 Combining Matchers - Concatenation - `make_concat_matcher`

```
let rec make_concat_matcher (ma, mb) = fun acceptor -> fun frag ->
  let fancy_acceptor frag' = mb acceptor frag' in (* Create acceptor for mb *)
  ma fancy_acceptor frag (* Call ma with fancy acceptor and original fragment *)
```

- Matcher for concatenation (sequence) of two patterns (`ma` followed by `mb`).
- Takes two matchers (`ma`, `mb`) as input and returns a new concatenated matcher.
- Concatenation Matcher Behavior:
  - Creates a fancy acceptor (`fancy_acceptor`) for the first matcher `ma`. This is the key to concatenation.
  - `fancy_acceptor`'s Role: `fancy_acceptor` is given to `ma`. If `ma` matches a prefix of the input fragment, `ma` will call `fancy_acceptor` on the remaining suffix (unmatched part).
  - `fancy_acceptor`'s Implementation: `fancy_acceptor frag' = mb acceptor frag'`. `fancy_acceptor` simply calls the second matcher `mb` with the original top-level `acceptor` and the suffix fragment (`frag'`) passed to `fancy_acceptor`. Chaining the matchers.
  - `ma fancy_acceptor frag`: Calls the first matcher `ma` with the `fancy_acceptor` and the original fragment. Starts the parsing process with the first matcher in the sequence.
- Sequential Matching: `ma` matches first, and if it succeeds and calls `fancy_acceptor`, then `mb` is invoked to match the remainder of the input.

7.1.7 Combining Matchers - Concatenation of List of Matchers - `make_appended_matchers`

```
let make_appended_matchers matchers =
  let rec chain_matchers matcher_list acceptor fragment =
    match matcher_list with
    | [] -> match_empty acceptor fragment (* Base case: Empty list of matchers - match empty string *)
    | m :: rest_matchers ->
        let fancy_acceptor frag' = chain_matchers rest_matchers acceptor frag' in (* Recursive fancy acceptor *)
        m fancy_acceptor fragment (* Call current matcher with fancy acceptor *)
  in
  chain_matchers matchers
```

Generalization of `make_concat_matcher` to handle concatenation of a list of matchers.

- Recursive Function `chain_matchers`: Processes the list of matchers recursively.
- Base Case: Empty Matcher List: If `matcher_list` is empty, it means we've matched all matchers in the sequence. Call `match_empty` to succeed (empty string match) and invoke the top-level `acceptor`.
- Recursive Step: Non-Empty Matcher List:
  - `m :: rest_matchers` : Decompose `matcher_list` into the head matcher `m` and the tail (rest of the matchers) `rest_matchers`.
  - `fancy_acceptor frag' = chain_matchers rest_matchers acceptor frag'`: Creates a recursive `fancy_acc` similar to `make_concat_matcher`, but now it calls `chain_matchers` recursively with the tail of the matcher list (`rest_matchers`). Recursive construction of acceptors for chaining matchers.
  - `m fancy_acceptor fragment`: Calls the current matcher `m` with the `fancy_acceptor` and the original fragment. Processes the current matcher in the sequence.

7.2 Translation Stages in Compilers (Recap and Context)

7.2.1 Compiler Stages (GCC Example) - Revisited

- Tokenization (Lexical Analysis): Characters -> Tokens.
- Parsing (Syntax Analysis): Tokens -> Parse Tree. (Matchers and acceptors are used in parsing).
- Semantic Analysis: Parse Tree -> Attributed Tree (type checking, declaration checking, static semantic checks).
- Code Generation: Attributed Tree -> Assembly Code.
- Assembler: Assembly Code -> Machine Code (Object Files - `.o`).
- Linker (ld): Object Files + Libraries -> Executable. (Resolves external references, links libraries).
- Loader: Loads executable into memory and runs it (OS Kernel component).

7.2.2 Alternative Compilation Approaches - IDEs and Hybrid Environments

- IDE Approach (In-Memory Compilation): Compiler directly produces machine code in RAM for faster development cycles.
- Hybrid Environments (Interpreters + JIT):
  - Interpreter: Compiles to bytecode (intermediate representation) and executes bytecode safely.
  - Just-In-Time (JIT) Compiler: Dynamically compiles frequently used bytecode to native machine code for performance optimization (e.g., Java, JavaScript engines).

7.3 Next Lecture - Types

- Next lecture will cover "Types" in more detail, building upon the foundation of syntax and parsing.
- Reading Assignment: Chapters 1-9 and 11 of textbook.

8 *<2025-01-30 Thu>* Types in Programming Languages

8.1 What is a Type?

8.1.1 Initial Thoughts on Type Definition

- Predefined data structure?
  - Argument against: User-defined types exist.
  - Argument against: Not all types are data structures (e.g., `enum` in C).
  - Types are related to data structures but not the same.

8.1.2 Definition 1: Type as Predefined or User-Defined Data Structure

- Predefined: Built into the language (e.g., `int`, `float`).
- User-defined: Created by the programmer (e.g., classes, structs).
- Problem: Not all types are data structures (e.g., enums).

8.1.3 Definition 2: Type as What Compiler Knows About a Value

- Set of things the compiler knows about a value.
- Example: `int n = ...; n + 1;` - Compiler knows it's a 32-bit integer in a specific range.
- Arguments against:
  - Variables without initial values still have types.
  - Hardware dependency of `int` size (16-bit, 32-bit, 64-bit).
  - Overflow behavior: `int_max + 1` in C/C++ leads to undefined behavior, not necessarily a type change, but the compiler might deduce a tighter range.
  - For Python, variables seem to have a union of all possible types, making this definition less applicable directly.

8.1.4 Definition 3: Type as a Set of Values and Operations

- A type is a set of values.
  - Example: Type of `n + 1` could be the set of possible values after incrementing an `int`.
- In object-oriented languages (C++, Python):
  - Type = Set of values + Set of associated operations.
  - Classes in C++ encapsulate both (data through instance variables, operations through methods).
- Set-based type definition was explored in the experimental language "Settl" (Set programming language).

8.1.5 Context Dependency of Type Definition

- No single universally accepted definition of "type".
- The best definition depends on the context and purpose.
- Need to clarify the intended purpose when defining "type".

8.2 Aspects of Types

8.2.1 Primitive vs. Constructed Types

- **Primitive Types:** Basic, built-in types.
  - Examples: `int`, `char`, `boolean`, `float`, `double`.
  - Seemingly "solved" problem, but implementations vary across languages.
- **Constructed Types:** User-defined types built from primitive or other types (classes, structs, enums, etc.).

1. Variations in Primitive Types: `int` Example
   - **`int` in C/C++:**
     - Minimum requirement: Represent values from -32767 to 32767 (at least 16 bits).
     - Implementation-defined size (e.g., 32-bit on many platforms).
     - `INT_MIN`, `INT_MAX` macros define the range for a specific implementation.
   - **`int` as Mathematical Integers:**
     - No size limits (limited only by memory).
     - No overflow issues.
     - Example: Some languages
   - **`int` in Javascript:**
     - Subset of `float`. Integers are floating-point numbers without a decimal part.
     - No fixed range like C/C++.
     - Limited by the precision of `float`, but can be very large.
2. Primitive Types are Not Always "Solved": `float` Example (IEEE 754 Floating Point)
   - `float` (IEEE 754 single-precision floating point) - 32 bits.
     - Sign bit (1 bit): 0 for non-negative, 1 for non-positive.
     - Exponent (8 bits): Biased exponent, actual exponent is E - 127.
     - Fraction (23 bits): Mantissa (with a hidden leading '1').
   - Represents numbers of the form: $(-1)^{\text{sign}} * 2^{(\text{E-127})} * (1.\text{F})_2$
   - **Normalization:** Hidden bit '1' for normalized numbers.
     - Problem: Cannot represent 0 directly with normalization.
     - Problem: Underflow when normalizing very small numbers.

   (a) Special Cases in IEEE 754 `float`
   - **E = 0 (and F != 0): Tiny Numbers (Denormalized/Subnormal)**
     - Representation: $(-1)^{\text{sign}} * 2^{(-126)} * (0.\text{F})_2$ (Exponent is fixed at -126, no hidden '1')
     - Used for numbers very close to zero, avoids abrupt underflow to zero.
     - Reduced precision due to leading zeros in the fraction.
     - Includes +/- 0 when F is all zeros.
   - **E = 0 and F = 0: Zero**

- Two representations: +0 (sign=0) and -0 (sign=1).
- Numerically equal, but can behave differently in some operations (e.g., division).

- **E = 255 (all ones) and F = 0: Infinity**
  - Two representations: +Infinity (sign=0) and -Infinity (sign=1).
  - Result of overflow or division by zero.
  - Can be generated by `1.0 / 0.0`.
  - Choice of trapping (program crash) or returning infinity on overflow (default).
- **E = 255 (all ones) and F != 0: NaN (Not a Number)**
  - Represents undefined or unrepresentable results (e.g., infinity - infinity, sqrt(-1)).
  - Many different NaN values depending on F, but usually treated alike.
  - `NaN != NaN` is always true. Use `isnan()` function to check for NaN.
  - Can be generated by operations like `0.0 / 0.0`, `infinity - infinity`.

(b) Implications of `float` Complexity for Type Design

- Designing even "simple" types like `float` is complex.
- Need to consider edge cases, special values, and error handling.
- Decisions on handling errors: Trap (exception) or return exceptional values (NaN, Infinity).
- Trade-offs in type design: Precision, range, performance, and handling of exceptional conditions.

8.3 Type System Aspects

8.3.1 Types as Annotations

- Types provide information for human readers (like comments with "teeth").
- Types are used by compilers for optimization (e.g., register allocation).
- Types can affect execution (e.g., type casting involves runtime conversion).

8.3.2 Type Inference

- Inferring types of expressions based on context.
- Example: `int i; float f; i * f;` - Inferring the result type as `float` due to mixed-type operation rules.

8.3.3 Type Checking

- Detecting errors in programs related to types.
- Prevents simple mistakes and improves reliability.
- Two main flavors: Static and Dynamic.

1. Static Type Checking

   - Type checking done by the compiler **before** program execution.
   - Advantages:
     - Early error detection (compile-time).
     - No type errors at runtime (guarantee).
     - Potential for faster execution (less runtime overhead).
   - Disadvantages:
     - Can be strict and less flexible.
     - Compiler errors can be frustrating during development.
   - Languages: C++, Java, OCaml (mostly).

2. Dynamic Type Checking
    - Type checking done **during** program execution (runtime).
    - Advantages:
        - More flexible and forgiving.
        - Programs can run even with potential type issues (until runtime error).
    - Disadvantages:
        - Runtime type errors can occur.
        - Slower execution due to runtime type checks.
        - No compile-time guarantee of type safety.
    - Languages: Python, Javascript.
3. Hybrid Approach: Static and Dynamic Checking
    - Languages can combine both static and dynamic checking.
    - Example: Java.
        - Mostly static type checking.
        - Dynamic type checking with casts (e.g., casting `Object` to `String`).

8.3.4 Strongly Typed Languages

- Languages where you cannot "cheat" about types.
- Type system enforces type rules strictly.
- Prevents treating a value of one type as another incompatible type without explicit and safe conversion.
- Can be static (OCaml) or dynamic (Python).
- Languages that are **not** strongly typed: C, C++.
    - Allow type casting and pointer manipulation that can bypass type system (e.g., casting `int*` to `float*`).

8.4 Type Equivalence

8.4.1 Type Equality Question

- Given two type expressions T and U, are they the same type?
- Two main approaches to determine type equivalence: Name Equivalence and Structural Equivalence.

1. Name Equivalence
    - Two types are the same if and only if they have the same name.
    - Example (C/C++ structs):

```c
struct S { int val; struct S *next; };
struct T { int val; struct T *next; };
struct S v;
struct T w;
// v = w; // Type error, S and T are different types even if structurally identical.
```

    - C/C++ structs and classes use name equivalence by default.
2. Structural Equivalence
    - Two types are the same if they have the same structure or representation.
    - Example (C `typedef`):

```
typedef int S;
typedef int T;
S x;
T w;
x = w; // Allowed, S and T are structurally equivalent to int.
```

- C `typedef` uses structural equivalence.
- Structural equivalence can be more complex with recursive types.

3. Abstract vs. Exposed Types

- **Abstract Type:**
  - Only name and operations are known.
  - Implementation details (structure) are hidden.
  - Name equivalence is more suitable for abstract types.
  - Promotes flexibility and modularity in implementation.
- **Exposed Type:**
  - Implementation details (structure) are known.
  - Structural equivalence can be considered.
  - Can lead to more efficient code generation.
  - Implementation details are visible and changes can affect users.
- `float` is partially exposed: We know its bit representation (IEEE 754), but the exact bit order might be implementation-defined.

8.5 Subtypes and Polymorphism

8.5.1 Subtype Relation

- "T is a subtype of U" (T <: U).
- Concept of subsets in terms of values.
- Related to subclasses in object-oriented programming.
- Subtype has "more operations" than its supertype (in OO context, subclass can add methods).

1. Example: `day` and `weekday` Types

- `type day = {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday}`
- `type weekday = {Monday, Tuesday, Wednesday, Thursday, Friday}`
- `weekday` is a subtype of `day` (subset of values).
- Functions defined for `day` can generally be used with `weekday` values.

8.5.2 Polymorphism

- "Many forms."
- Single expression or function can have different interpretations based on types.
- Weber's book chapter on Polymorphism

1. Overloading (Ad-hoc Polymorphism)

- Multiple functions with the same name but different type signatures.
- Compiler selects the correct function based on argument types.

- Example: `cosine(x)` - different implementations for `float` and `double`.
- Function names are "overloaded" to mean different things.

2. Coercion (Ad-hoc Polymorphism)
   - Automatic or implicit type conversions performed by the compiler.
   - Compiler inserts conversions to make code type-compatible.
   - Example (Fortran): `float x; int i; x * i;` - `i` is coerced to `float` before multiplication.
   - Can lead to unexpected behavior due to implicit conversions.
   - Example of coercion issue (C): `uidt_t x = -1; if (x < 0) ...` where `uidt_t` is unsigned. `-1` is coerced to `unsigned int max`, leading to unexpected comparison result.

   (a) Combination of Overloading and Coercion - Potential Ambiguity
   - Example: `arctan(y, x)` overloaded for `(float, float)` and `(double, double)`.
   - Call `arctan(3.0, 5.9f)` (double, float).
   - Possible coercion of `float` to `double` to use `(double, double)` version.
   - Ambiguity can arise with multiple overloaded functions and possible coercions.
   - Example (C++): `int f(float); int f(int, float); f(3, 7);` - Ambiguous call, compiler error.
   - "Ad-hoc polymorphism" (Weber's term): Overloading and coercion rules in languages like C++, Java are complex and sometimes inconsistent.

3. Parametric Polymorphism (Generic Polymorphism)
   - More structured and organized approach to polymorphism.
   - Types have type parameters.
   - Polymorphism achieved by instantiating type parameters with specific types.
   - "Not ad-hoc" - more principled and less error-prone.

   (a) Generic Types (Java Example)
   - Java Generics for parametric polymorphism (similar to OCaml type constructors).
   - Example (Java): `Collection<String> c;` - Collection specifically of Strings.
   - Type safety at compile time.
   - Example (Java code comparison):
     - **Traditional Java (pre-generics):**

```
Collection c = ...;
Iterator i = c.iterator();
while (i.hasNext()) {
  String s = (String) i.next(); // Cast needed
  if (s.length() == 1) {
    i.remove();
  }
}
```

   - **Modern Java (with Generics):**

```
Collection<String> c = ...;
Iterator<String> i = c.iterator();
while (i.hasNext()) {
  String s = i.next(); // No cast needed, type is String
  if (s.length() == 1) {
    i.remove();
  }
}
```

- Generics provide compile-time type safety and cleaner code (less casting).

(b) Generics vs. Templates (Java/OCaml vs. C++)

- **Generic Types (Java, OCaml):**
  - Compiler checks types and generates machine code **once** for a generic class/function.
  - Type parameters are checked at compile time.
  - Single version of machine code is used for all instantiations with different types.
  - Cleaner and safer, but less flexible for type-specific optimizations.
- **Templates (C++):**
  - Templates are like "code with holes".
  - Compiler instantiates the template for each specific type used.
  - Type checking and code generation happen **at instantiation time**.
  - Different machine code is generated for each instantiation.
  - More flexible (type-specific optimizations possible), but potentially more compile-time overhead and less strict type checking initially.
- Generics are typically used in higher-level languages (Java, OCaml), Templates in lower-level languages (C++).

9 *<2025-02-04 Tue>* Continuing Types and Introduction to Java

9.1 Generics vs. Templates (Recap and Further Discussion)

9.1.1 Generic Types

- **Static Checking:** Done when the generic type/method is **defined**.
- **Code Generation:** Compiler can generate code for generic types (e.g., `List<X>`) even without knowing the concrete type `X`.
- **Runtime Representation:** Generic values often represented as a fixed-size (e.g., 64-bit) value (pointer in OO languages), assuming all types can be represented uniformly.

9.1.2 Templates

- **Static Checking:** Done at **type instantiation** time (when the template is **used** with a concrete type like `List<String>`).
- **Code Generation:** Compiler generates **separate code** for each instantiation (e.g., `List<String>` vs. `List<Thread>`).
- **Optimization:** Potential for better optimization as the compiler knows the concrete type at instantiation.
- **Code Duplication:** Can lead to more code and larger executables.

9.1.3 Trade-offs

- **Generics:** Simpler, less code duplication, static checking at definition, potentially less optimized code.
- **Templates:** More complex, code duplication, static checking at instantiation, potential for better optimization.

9.2 Problems with Generics: Subtype Issues in Java

9.2.1 Example: `List<String>` vs. `List<Object>`

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls; // Invalid in Java

lo.add(new Thread()); // Adding a Thread to List<Object>

String s = ls.get(0); // Potential ClassCastException if List<String> is actually List<Object>
```

- **Problem:** If `List<String>` were a subtype of `List<Object>`, the above code would compile but lead to runtime type errors.
- Java prevents `List<String>` from being a subtype of `List<Object>` to ensure type safety.

9.2.2 Subtype Rule and Operations

- If X is a subtype of Y (X <: Y), then:
  - Every value of type X should be usable in a context expecting type Y.
  - Every operation on Y values should also work on X values (or X has a superset of Y's operations).

9.2.3 `char*` vs. `const char*` in C/C++ Analogy

```
char c = 'a';
const char *p = &c; // char* is a subtype of const char*

*p = 'b'; // Error: Cannot modify through const char* pointer

void f(char *ptr) {
  *ptr = 'z'; // f can modify the char pointed to by ptr
}

f(&c);
printf("%c\n", *p); // Output might be 'z' if f modified c
```

- `char*` (pointer to mutable char) is a subtype of `const char*` (pointer to immutable char through **this** pointer).
- `const char*` has fewer operations (cannot modify through it).
- Subtyping is about **capabilities** and **restrictions**, not complexity of type name.

9.2.4 Why `List<String>` is Not Subtype of `List<Object>`

- `List<Object>` allows adding any `Object` (e.g., `Thread`).
- `List<String>` should only allow adding `String` objects.
- If `List<String>` were a subtype of `List<Object>`, it would violate type safety because you could add a `Thread` to what is supposed to be a `List<String>`.
- Java lists are **mutable**, which is the root of this problem. Immutable lists would allow for covariant subtyping.

9.3 Wildcards in Java Generics

9.3.1 Unbounded Wildcard: `List<?>`

```
public static void printList(List<?> list) {
  for (Object obj : list) { // Safe to iterate as elements are Objects
    System.out.println(obj);
  }
}

List<String> strings = new ArrayList<>();
printList(strings); // Works now!
```

- `List<?>` means a list of **some** unknown type.
- Safe for **reading** elements as they are known to be `Object` (supertype of all objects).
- Not safe for **writing** (adding) elements because the actual type is unknown. You cannot `list.add(something)` in a `List<?>` generally.

9.3.2 Bounded Wildcards

1. Upper Bounded Wildcard: `List<? extends Shape>`

   ```
   public static void printShapes(Collection<? extends Shape> shapes) {
     for (Shape s : shapes) { // Safe to iterate as elements are Shapes or subtypes
       printShape(s); // Assuming printShape(Shape s) exists
     }
   }

   Collection<Rectangle> rectangles = new ArrayList<>();
   printShapes(rectangles); // Works if Rectangle extends Shape

   Collection<String> strings = new ArrayList<>();
   // printShapes(strings); // Error: String is not a subtype of Shape
   ```

   - `<? extends Shape>`: Type must be `Shape` or a subtype of `Shape` (e.g., `Rectangle`, `Square`).
   - Good for **reading** (consuming) elements of type `Shape` or its subtypes.

2. Lower Bounded Wildcard: `List<? super Shape>`

   ```
   public static <T> void copyToCollection(T[] ar, Collection<? super T> co) {
     for (T obj : ar) {
       co.add(obj); // Safe to add T because Collection accepts supertype of T
     }
   }

   Shape[] squareArray = new Square[10];
   Collection<Shape> shapeCollection = new ArrayList<>();
   copyToCollection(squareArray, shapeCollection); // Works: Shape is supertype of Square

   Collection<Rectangle> rectangleCollection = new ArrayList<>();
   // copyToCollection(squareArray, rectangleCollection); // Error: Rectangle is not supertype of Square
   ```

   - `<? super Shape>`: Type must be `Shape` or a supertype of `Shape` (e.g., `Shape`, `Object`).
   - Good for **writing** (producing/adding) elements of type `Shape` or its subtypes to the collection.
   - `super` is needed for methods that **consume** or **store** elements into the generic type.

9.3.3 Type Variables <T> for Type Consistency

```
public static <T> void convert(T[] ar, Collection<T> co) { // T is a type variable
  for (T obj : ar) {
    co.add(obj); // Type safe as both array and collection are of the same type T
  }
}

String[] stringArray = {"a", "b", "c"};
Collection<String> stringCollection = new ArrayList<>();
convert(stringArray, stringCollection); // T is inferred as String

// Collection<Object> objectCollection = new ArrayList<>();
// convert(stringArray, objectCollection); // Error: Collection<Object> is not Collection<String> if T must be
↪  same
```

- `<T>` introduces a type variable `T`.
- Ensures that the array and collection must be of the **same** type `T`.
- Allows copying between collections of the **same** type.
- Can be combined with bounds: `<T extends Shape>` or `<T super Shape>` (though less common for `super`).

9.3.4 Question Mark as Syntactic Sugar

- `?` wildcard can be seen as shorthand for a type variable that is used only once and doesn't need a name.
- Longer form with named type variables can be used, but wildcard `?` is often preferred for brevity when the type variable is not referenced elsewhere.

9.4 Duck Typing: A Simpler Alternative

9.4.1 Philosophy

- "If it walks like a duck and quacks like a duck, then it must be a duck."
- Type is determined by **behavior** (methods and properties) rather than explicit type declarations.

9.4.2 Advantages

- **Simplicity:** No need for complex type systems, generics, or wildcards.
- **Flexibility:** Code can work with any object that implements the required methods, regardless of its class.

9.4.3 Disadvantages

- **Runtime Errors:** Type errors are detected only at runtime when a method is called that an object doesn't implement.
- **Less Static Checking:** Misses potential errors at compile time, leading to potential runtime failures.
- **Maintainability:** Can be harder to understand code as type relationships are less explicit.

9.4.4 Languages Using Duck Typing

- Python, Javascript, Ruby, etc. (Dynamically typed languages).

9.4.5 Trade-off: Static vs. Dynamic Typing

- **Static Typing (with Generics):** More complex type system, more compile-time checking, better reliability, potentially more verbose code.
- **Duck Typing (Dynamic Typing):** Simpler type system, less compile-time checking, more flexible, potentially runtime errors, more concise code.
- Choice depends on the application: Reliability-critical systems often prefer static typing, exploratory or scripting tasks may favor dynamic typing.

9.5 Java: Performance, Concurrency, and Design Principles

9.5.1 Java in TIOBE Index

- Python #1, C++ #2, Java #3 in popularity (TIOBE Index).
- Python's rise due to machine learning.
- Java's strength: Server-side applications and **multithreading**.

9.5.2 World's Fastest Computer: El Capitan

- 1.75 Exaflops (Linpack benchmark).
- 43,808 nodes, AMD CPUs and GPUs.
- 11 million+ cores.
- 30 Megawatts power consumption.
- 59 Gigaflops/Watt efficiency.
- Operating System: Red Hat Enterprise Linux.
- Future of computing: massively parallel, heterogeneous architectures.

9.5.3 Java's Origins and Motivation (Sun Microsystems, 1990s)

- **Problem:** Limitations of C/C++ for networked and embedded devices.
  - Multiple Architectures: Need for portability across CPUs.
  - Slow Networks/Large Executables: Inefficient for distribution.
  - Crashes: Reliability issues with C/C++.
  - Slow Build/Test Cycles: Productivity issues.
- **Inspiration:** Xerox PARC and Smalltalk.
  - Smalltalk IDE: Integrated environment, bytecode interpreter, garbage collection, reliability.

9.5.4 Java Design Principles: "C++ Meets Smalltalk"

- **Bytecode and Interpreter:** Solves portability and executable size issues.
- **Garbage Collection:** Solves memory management and crash issues.
- **Runtime Checks (e.g., array bounds):** Improves reliability.
- **Single Inheritance:** Simplicity over C++'s multiple inheritance.

- **No Pointers (References):** Improved safety and simplicity. References are opaque and less powerful than C/C++ pointers.
- **Portable Primitive Types:** Fixed sizes and behavior across platforms (byte, short, int, long, float, double, boolean). Standardized behavior for portability.

9.5.5 Java's Strengths: Portability, Reliability, Multithreading

- Designed for networked environments and server-side applications.
- Excellent support for multithreading (important for server applications).

10 *<2025-02-11 Tue>* Java, Concurrency, and Object Model Details

10.1 Java vs. C++: Design Goals and Trade-offs

10.1.1 Java: Higher-Level Abstraction

- Designed to be more abstract than C++.
- **Portability:** Run on various machines without recompilation hassles (bytecode).
- **Reliability:** More robust than C++ (garbage collection, safety features).
- **Concurrency:** Built-in support for multithreading.
- **Performance:** Trade-off for portability and reliability; aims for a "sweet spot" between performance and safety.

10.1.2 C++: Lower-Level Control

- Designed for performance and system-level programming.
- More direct access to hardware.
- Manual memory management (programmer responsibility).
- Less emphasis on portability and safety compared to Java.

10.2 Java Bytecode and Just-In-Time (JIT) Compilation

10.2.1 Bytecode

- Portable intermediate representation of Java programs.
- Stack-oriented virtual machine instructions.
- Interpreter executes bytecode on different platforms.
- Advantages: Portability, smaller executables.
- Disadvantage: Slower than native machine code initially.

10.2.2 Just-In-Time (JIT) Compiler

- Runtime component that translates bytecode to native machine code **during execution**.
- Identifies "hotspots" (frequently executed code) for JIT compilation.
- Optimizes performance by compiling frequently used bytecode to native code for the specific platform.
- Aims to combine portability of bytecode with performance approaching native code.

10.3 Single Inheritance in Java

10.3.1 Simplicity and Performance

- Java adopts single inheritance (class extends only one superclass) for simplicity and potential performance benefits.
- Simpler type hierarchy compared to C++'s multiple inheritance.
- Easier to implement efficiently.

10.3.2 Java Type Hierarchy

- Tree-like structure rooted at `Object`.
- `class C extends D`: Class `C` inherits from superclass `D`.
- Default superclass: `Object` (if no `extends` clause).
- Examples: `Object`, `Thread`, `String`, `GreenThread` (subclass of `Thread`).

10.4 Arrays in Java

10.4.1 Arrays as Objects

- Arrays in Java are objects, residing on the heap and garbage collected.
- Type: `int[] a = new int[27];` - `int[]` is the array type (size not part of type).
- Dynamic Size: Array size can be determined at runtime (not compile-time constant).
- Fixed Size After Allocation: Array size cannot be changed after creation.

10.4.2 Arrays vs. C++ Arrays

- **C++ (local array):** `int a[27];` - Stack-allocated, lifetime tied to function scope, size must be compile-time constant (or variable-length array in C99 and later).
- **Java (similar local syntax, but heap allocation):** `int[] a = new int[27];` - Heap-allocated, garbage collected, dynamic size.

10.4.3 Escape Analysis Optimization

- Java compiler can perform escape analysis to determine if an array is only used locally within a method.
- If array doesn't "escape" (e.g., not returned, not passed to other methods), compiler can allocate it on the stack for performance optimization (like C++ local arrays).

10.5 Abstract Classes and Interfaces

10.5.1 Abstract Classes

- Classes with abstract methods (methods declared without implementation).
- Keyword `abstract` (for class and methods).
- Cannot be instantiated directly (no `new AbstractClass()`).
- Purpose: Define a common interface and partial implementation for subclasses.
- Subclasses **must** implement abstract methods to become concrete (instantiable).
- Example:

```
abstract class List {
  abstract void append(Object obj); // Abstract method
  int length() { ... }           // Concrete method
}

class LinkedList extends List {
  void append(Object obj) { ... } // Concrete implementation of append
}
```

10.5.2 Interfaces

- Define contracts (APIs) without implementation.
- Keyword `interface`.
- Contains only method declarations (no method bodies) and constants.
- Interfaces can extend other interfaces (interface inheritance).
- Classes can **implement** multiple interfaces (implements keyword).
- Example:

```
interface X {
  int length();
  void append(Object obj);
}

abstract class List implements X { // List class implementing interface X
  // ... (can have abstract or concrete methods)
}
```

10.5.3 Abstract Classes vs. Interfaces

- **Abstract Class:** Can have both abstract and concrete methods, single inheritance hierarchy.
- **Interface:** Purely abstract methods (no implementation), multiple inheritance of interfaces.
- Java uses interfaces as a form of **multiple inheritance** of types (but not implementation).
- Abstract classes can bundle concrete and interface-like behavior.

10.6 Final Classes and Methods

10.6.1 Final Classes

- Keyword `final` for classes.
- Cannot be subclassed (extended).
- Purpose: Prevent inheritance, for security or performance reasons.

10.6.2 Final Methods

- Keyword `final` for methods.
- Cannot be overridden in subclasses.
- Purpose: Prevent method overriding, for security or performance reasons.

10.6.3 Performance Optimization with `final`

- **Inlining:** Compiler can inline `final` methods because it knows the method implementation will not be overridden.
- Reduces method call overhead, improves performance.
- Example:

```
final class MyClass {
  final int f(int a, int b) {
    return a * a + b; // Simple method
  }
}
```

- JIT compiler can replace calls to `f()` with the actual computation `a * a + b`.

10.6.4 Trust and Security with `final`

- `final` methods and classes can enhance security by preventing untrusted subclasses from altering critical behavior.
- Ensures that certain methods or classes behave exactly as designed, without risk of being overridden maliciously or accidentally.

10.7 `Object` Class Methods (Plumbing of Java)

10.7.1 `Object()` - Constructor

- Default constructor for all Java objects.
- Creates a basic, featureless object.
- Use case: Sentinel objects, test objects, very basic objects when needed.

10.7.2 `boolean equals(Object obj)`

- Checks for semantic equality between objects (not just reference equality).
- Default implementation: Reference equality (same as `==`).
- Can be overridden to define custom equality based on object content.
- Consistency with `hashCode()` is crucial: If `equals()` returns `true`, `hashCode()` should return the same value for both objects.

10.7.3 `final Class<?> getClass()`

- Returns the runtime class of an object as a `Class` object.
- `Class` objects are runtime representations of classes.
- `final` method: Cannot be overridden, ensures reliable class information.
- Use case: Runtime type introspection, debugging, generic programming.
- Generic return type `Class<?> extends Object>}` (or more precisely, `Class<? extends |O's type|>`) reflects the class of the object.

10.7.4 `int hashCode()`

- Returns a hash code value for the object (integer).
- Used in hash-based collections (e.g., `HashMap`, `HashSet`).
- Default implementation: Hash based on object's memory address.
- Should be consistent with `equals()` method: Equal objects must have the same `hashCode()`.

10.7.5 `String toString()`

- Returns a string representation of the object.
- Default implementation: Class name + hexadecimal representation of hash code.
- Can be overridden to provide more informative string representations (for debugging, logging, etc.).

10.7.6 `protected Object clone() throws CloneNotSupportedException`

- Creates a shallow copy of the object.
- Protected method (not for general use directly).
- Can throw `CloneNotSupportedException` if cloning is not supported for the class.
- `Cloneable` interface: Marker interface to indicate that a class supports cloning.
- Originally intended for general cloning, but now considered problematic and less used.

10.7.7 `protected void finalize() throws Throwable`

- "Destructor" in Java (called by garbage collector before object reclamation).
- Protected method (not for general use directly).
- Can throw any `Throwable` exception.
- Purpose: Perform cleanup operations (releasing resources, closing files) before garbage collection.
- **Obsolete and Discouraged:** Finalization is unreliable, can cause performance issues and delays garbage collection. Modern Java recommends alternatives for resource management (e.g., `try-with-resources`).

10.8 Concurrency in Java: Threads and Synchronization

10.8.1 `Thread` Class

- Represents a thread of execution.
- Subclass of `Object`.
- Each `Thread` object has its own program counter, stack, and CPU (virtual or real core).
- Two ways to define thread code:
  - **Extending `Thread` class:** Override `run()` method. (Less common, less flexible).

```
class PrimeThread extends Thread {
  long minPrime;
  PrimeThread(long minPrime) {
    this.minPrime = minPrime;
  }
  public void run() {
```

```
    // compute primes larger than minPrime . . .
  }
}
// Then:
PrimeThread p = new PrimeThread(143);
p.start();
```

- **Implementing `Runnable` interface:** Implement `run()` method in a separate class, pass `Runnable` object to `Thread` constructor. (More common, more flexible).

```
class PrimeRun implements Runnable {
  long minPrime;
  PrimeRun(long minPrime) {
    this.minPrime = minPrime;
  }
  public void run() {
    // compute primes larger than minPrime . . .
  }
}
// Then:
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

10.8.2 `Runnable` Interface

```
public interface Runnable {
  void run();
}
```

- Single method interface: `void run()`.
- Defines the task to be executed by a thread.
- Allows implementing multithreading without being restricted to the `Thread` class hierarchy.

10.8.3 Thread Lifecycle States

1. **New:** Thread object created but not started yet.
2. **Runnable:** Thread is ready to run, waiting for CPU time. (Can be running or ready to run).
   - **Yielding:** `Thread.yield()` - Voluntarily gives up CPU time to other runnable threads (polite but not guaranteed effect).
3. **Timed Waiting:** Thread is paused for a specified time period (e.g., `Thread.sleep(millis)`).
4. **Waiting:** Thread is waiting for an event or condition (e.g., `Object.wait()`, waiting for a lock release).
5. **Blocked:** Thread is waiting to acquire a lock (e.g., entering a `synchronized` block when lock is held by another thread).
   - **I/O Blocking:** Waiting for I/O operation to complete (e.g., `read()`, `write()`).
6. **Terminated:** Thread has finished execution (e.g., `run()` method completes). Thread object becomes a "tombstone."

10.8.4 Race Conditions

- Occur when multiple threads access shared resources concurrently, and the outcome depends on the timing of thread execution.
- Example: Thread A writes to variable `V` while Thread B reads `V` simultaneously.
- Can lead to data corruption, inconsistent state, and unpredictable program behavior.
- Difficult to debug due to timing-dependent nature.

10.8.5 `synchronized` Keyword and Locks (Monitors)

- Java's primary mechanism for thread synchronization and preventing race conditions.

1. Synchronized Methods

```
class Counter {
  private int count = 0;
  public synchronized int next() { // Synchronized method
    return count++;
  }
}
```

   - `synchronized` keyword on a method makes it **mutually exclusive** with respect to the object instance.
   - When a thread enters a `synchronized` method of an object, it acquires a **lock** (monitor) on that object.
   - Only one thread can hold the lock for a given object at a time.
   - Other threads trying to enter any `synchronized` method of the same object will be **blocked** until the lock is released.
   - Implicit lock acquisition and release at method entry and exit.

2. Implementation of `synchronized` Methods (Simplified)

```
synchronized int next() {
  lock(this);     // Acquire lock on 'this' object
  try {
    int r = count++; // Critical section (protected by lock)
    return r;
  } finally {
    unlock(this);   // Release lock (even if exception occurs)
  }
}
```

   - Every Java object has an associated lock (monitor).
   - `synchronized` methods use this lock to ensure mutual exclusion.
   - `finally` block ensures lock release even if exceptions occur in the critical section.

10.8.6 Performance Bottleneck with `synchronized` (Spin Locks)

- `synchronized` methods use **spin locks** (simplified view) or more sophisticated locking mechanisms internally.
- **Spin lock:** Thread continuously checks if the lock is available in a loop ("spinning") until it acquires the lock.
- **Performance issue:** In high-contention scenarios (many threads trying to access the same lock), spin locks can waste CPU cycles as threads busy-wait for the lock.
- Contention and scalability issues with excessive use of `synchronized`.

10.8.7 `wait()`, `notify()`, `notifyAll()` Methods (Object Class)

- Low-level synchronization primitives for inter-thread communication and more efficient waiting.
- `wait()`:
  - Called from within a `synchronized` block/method.
  - Releases the object's lock.
  - Puts the calling thread in the **waiting** state (thread is no longer runnable, doesn't consume CPU).
  - Waits until `notify()` or `notifyAll()` is called on the **same** object by another thread.
- `notify()`:
  - Called from within a `synchronized` block/method of the **same** object on which `wait()` was called.
  - Wakes up **one** of the waiting threads (if any). Which thread is woken up is implementation-dependent (often the longest waiting thread).
  - Woken-up thread becomes **runnable** again (re-acquires the lock when it gets CPU time).
- `notifyAll()`:
  - Similar to `notify()`, but wakes up **all** threads waiting on the object.
  - All woken-up threads become **runnable** and compete to re-acquire the lock.

10.8.8 Higher-Level Concurrency Utilities (Java Standard Library)

- Built upon `wait()`, `notify()`, `notifyAll()` for more convenient and efficient concurrency management.
- Examples:
  - **`Semaphore`:** Controls access to a limited number of resources (permits). `acquire()`, `tryAcquire()`, `release()`.
  - **`Exchanger`:** Rendezvous point for two threads to exchange objects. e.g.:

```
Exchanger<DataBuffer> exchanger = new Exchanger<DataBuffer>();
// In Thread A:
currentBuffer = exchanger.exchange(currentBuffer);
// In Thread B:
currentBuffer = exchanger.exchange(currentBuffer);
```

- **`CountDownLatch`:** Synchronization aid for one or more threads to wait until a count reaches zero (one-time event). `countDown()`, `await()`.
- **`CyclicBarrier`:** Reusable synchronization barrier for a group of threads to wait for each other to reach a common point (cyclic, reusable barrier). `await()`.

10.9 Java Memory Model and Optimization Challenges (Reordering)

10.9.1 Out-of-Order Execution and Compiler/Hardware Optimizations

- Compilers and modern processors may reorder instructions for performance optimization.
- **As-if-Serial Semantics (Single-Thread Rule):** Reordering is allowed as long as the behavior is **as if** the code was executed in the source code order **within a single thread**. Single-threaded observation must be unchanged.

10.9.2 Example of Instruction Reordering (Potential Issue in Multithreading)

```
// Thread 1:
v++; // Increment v
w++; // Increment w

// Potential machine code (reordered by compiler/hardware):
load W into register2
load V into register1
add 1 to register2
add 1 to register1
store register2 into W
store register1 into V
```

- If Thread 2 reads `v` and `w` concurrently, it might observe `w` being incremented before `v` due to reordering, even though in source code `v++` comes first.

10.9.3 Data Races and Inconsistent Observations

- Reordering can lead to data races and inconsistent observations by other threads in multithreaded programs.
- Thread 2 might see an intermediate, inconsistent state where `w` is updated but `v` is not yet, violating program logic.

10.9.4 `synchronized` and Memory Ordering

- `synchronized` blocks and methods provide **memory barriers** (in addition to mutual exclusion).
- Memory barriers enforce ordering constraints on memory operations, preventing certain reorderings.
- Entering a `synchronized` block typically implies acquiring a memory barrier (e.g., **acquire barrier**).
- Exiting a `synchronized` block typically implies releasing a memory barrier (e.g., **release barrier**).
- Memory barriers ensure that operations **within** a `synchronized` block are not reordered **out** of the block, and that changes made inside the block become visible to other threads **after** exiting the block.

10.9.5 Critical Sections and Synchronization Guarantees

- `synchronized` methods and blocks define **critical sections** - code regions that must be executed atomically with respect to other synchronized code on the same object.
- Synchronization ensures that operations within critical sections are not interleaved or reordered in a way that violates program correctness in multithreaded contexts.

11 *<2025-02-13 Thu>* Java Memory Model and Logic Programming (Prolog)

11.1 Special Relativity and Java Memory Model Analogy

11.1.1 Minkowski Diagrams (Physics)

- One spatial dimension (x-axis) and time dimension (t-axis).
- Speed of light (c) is set to 1 for convenience.
- **Future:** Region of spacetime that can be influenced by an event at the origin.
- **Past:** Region of spacetime that can influence an event at the origin.
- **Elsewhere/Elsewhen:** Region of spacetime causally disconnected from the origin.

11.1.2 Race Conditions (Java/Concurrency)

- Analogy to special relativity: Threads sending "messages" via shared objects.
- Race conditions: Concurrent access to shared objects, where timing affects results.
  - Similar to the "elsewhere" region in Minkowski diagrams: Unreliable message passing.
- Goal: A simple model to understand and avoid race conditions.
- **Java Memory Model (JMM):** Specifies how threads interact through shared memory.
  - More complex than Minkowski diagrams.
  - Defines rules for when memory operations are visible to other threads.

11.2 Java Memory Model (JMM) - Key Concepts

11.2.1 Single-Threaded Execution and "As-If" Rule

- Assume single thread executes correctly (follows language rules).
- Compilers and hardware can **reorder** operations **within** a single thread as long as the **observable** behavior (from that thread's perspective) is **as if** the original order was preserved.
- Reordering is for **performance** (caching in registers, instruction scheduling).

11.2.2 Exceptions to Reordering (for Multithreaded Safety)

- **`volatile` Keyword:**
  - Variables declared `volatile` have special visibility and ordering guarantees.
  - Reads and writes to `volatile` variables **must** occur in program order (no reordering).
  - `volatile`: "Evaporates" - Value can change unexpectedly from another thread.
  - Compiler cannot cache or optimize `volatile` accesses in the same way as non-volatile variables.
  - Allows simple message passing between threads.
- **`synchronized` Keyword (Enter/Exit Monitor):**
  - Provides mutual exclusion (locks) and memory barriers.
  - **Enter Monitor:** Acquiring a lock (entering a `synchronized` block/method).
  - **Exit Monitor:** Releasing a lock (exiting a `synchronized` block/method).
  - Prevents reordering of operations into or out of the critical section (code protected by the lock).
  - Enforces memory consistency: Writes inside a `synchronized` block become visible to other threads **after** exiting the block.

11.2.3 Java Memory Model Reordering Table (Simplified)

- A table defines when operations can be reordered.
- **Operations Categories:**
  - **A:** Normal load and store (non-volatile variables).
  - **B:** Volatile load and Enter Monitor.
  - **C:** Volatile store and Exit Monitor.
- Table (simplified):

1st Op	2nd Op	Reorderable?
A	A	Yes
A	B	Yes*
A	C	No
B	A	No
B	B	No
B	C	No
C	A	Yes*
C	B	No
C	C	No

*Can expand critical section.

- **Key Points:**
  - Normal loads/stores (A) can be reordered freely within a single thread.
  - Entering a critical section (B) can cause normal operations before the lock to be moved into the critical section (A before B can become B, then A).
  - Exiting a critical section (C) can cause normal operations after the lock to be moved into critical section.
  - Operations involving `volatile` or locks (B, C) have stricter ordering constraints.
  - The critical sections can expand but not shrink past the enter/exit points
- **Goal of JMM Table:**
  - Allow compiler and hardware optimizations while preserving essential ordering for thread safety.
  - Provide a framework for understanding when reordering is allowed and when it is not.

11.3 Implementing Synchronization with `volatile` (Bakery Algorithm)

- It's possible to implement mutual exclusion (like `synchronized`) using only `volatile` variables, but:
  - It's complex (e.g., Bakery Algorithm).
  - It's often inefficient (requires multiple `volatile` variables and careful synchronization logic).
- Hardware support (atomic operations, compare-and-swap) is generally more efficient for synchronization than pure `volatile` implementations.
- `synchronized` in Java leverages hardware support for atomic operations to provide more efficient synchronization than pure `volatile` solutions.

11.4 Logic Programming (Prolog) - Introduction

11.4.1 Third Major Programming Paradigm

- **Imperative:** Commands, sequential execution, side effects.
- **Functional:** Functions, function calls, ideally no side effects.
- **Logic:** Predicates, logical relationships (and, or, not), no functions, no side effects (in pure logic programming).

11.4.2 Key Ideas

- **Predicates:** Relations or propositions that can be true or false (or unknown). Not functions returning Booleans.
- **Declarative Programming:** Specify **what** result is desired, not **how** to compute it.
- **Logic and Control:** Separate the logical specification of a problem from the control strategy for solving it.
  - **Logic:** What makes an answer correct. (Specification)
  - **Control:** How to efficiently find the answer. (Implementation)
- **Goal:** Divide and conquer: Separate logic (programmer A) and control (programmer B) for easier development and maintenance.

11.4.3 Prolog Example: Sorting a List

```
sort(L, S) :-  % S is the sorted version of L
  perm(L, S),  % S is a permutation of L
  sorted(S).    % S is sorted

perm([], []).            % Empty list is a permutation of itself
perm([X|L], R) :-       % X|L is a list with head X, tail L
  perm(L, PL),          % PL is a permutation of L
  append(P1, P2, PL),    % PL is split into P1 and P2
  append(P1, [X|P2], R).  % R is formed by inserting X in PL

sorted([]).               % Empty list is sorted
sorted([_]).              % Singleton list is sorted
sorted([X,Y|Z]) :-         % List with at least two elements
  X =< Y,                % X <= Y
  sorted([Y|Z]).           % Tail of list is sorted
  %sorted([Y|Z]). Original, incorrect definition

append([], L, L).
append([X|L], M, [X|LM]) :-
  append(L, M, LM).
```

- **`sort(L, S)`:** Predicate: `S` is the sorted version of list `L`.
- **`perm(L, S)`:** Predicate: `S` is a permutation of list `L`.
- **`sorted(S)`:** Predicate: List `S` is sorted.
- **`append(L1, L2, L3)`:** Predicate: `L3` is the concatenation of `L1` and `L2`.
- **`:-`:** Means "if" (implication).
- **`,`:** Means "and" (conjunction) at top level.
- **`[]`:** Empty list.
- **`[X|L]`:** List with head element `X` and tail list `L`.
- **`_`:** Anonymous variable (don't care about its value).
- **`=<`:** Less than or equal to operator.
- **Capital letters:** Logical variables (e.g., `L`, `S`, `X`, `Y`).
- **Lowercase letters:** Atoms (e.g., `sort`, `perm`, `sorted`, `append`).

11.4.4 Prolog Syntax and Terminology

- **Term:** Basic syntactic unit.
  - **Number:** Integer or floating-point number.
  - **Atom:** Symbolic constant (like an identifier), starts with lowercase letter or enclosed in single quotes.
  - **Variable:** Logical variable, starts with uppercase letter or underscore.
  - **Structure:** Compound term (like a data structure), has a **functor** (atom) and **arguments** (terms). `f(t1, t2, ..., tn)`.
- **Clause:** Statement in Prolog (fact or rule).
  - **Fact:** Unconditional statement (e.g., `sorted([]).`).
  - **Rule:** Conditional statement (e.g., `sort(L, S) :- perm(L, S), sorted(S).`).
- **Predicate:** Relation defined by a set of clauses (e.g., `sort/2`, `perm/2`, `sorted/1`, `append/3`).
- **Functor:** The name of a structure (e.g., `pr` in `pr(3, 9, XZ(27))`).
- **Arity:** The number of arguments of a functor (e.g., `pr/3` has arity 3).
- **Logical Variable:** Variable that can be bound to a term.
  - Scope: Limited to the clause in which it appears.
  - Binding: Variable can be bound to a value during execution.
  - Unbinding: On backtracking (failure), variables can become unbound.

11.4.5 Syntactic Sugar in Prolog

- **Operators:**
  - Arithmetic operators (`+`, `-`, `*`, `/`, etc.) are just syntactic sugar for structures.
  - `2 + 2` is equivalent to `'+'(2, 2)`.
- **Lists:**
  - `[]`: Empty list (atom).
  - `[a, b, c]`: List with elements `a`, `b`, and `c`.
  - Equivalent to: `'.'(a, '.'(b, '.'(c, [])))`.
  - `[X|L]`: List with head `X` and tail `L` (equivalent to `'.'(X, L)`).
- **Clauses:**
  - `head :- body.` is equivalent to `':-'(head, body)`.

11.4.6 Prolog Execution (Simplified)

- **Left-to-right, depth-first search:**
  - Tries to satisfy goals (predicates) from left to right within a clause.
  - Recursively tries to satisfy subgoals.
  - If a goal fails, backtracks to previous choice point and tries a different alternative.
- **Unification:** Process of matching terms and binding variables.
  - `X = 1`: Unifies `X` with `1`, binding `X` to `1`.

  - `X = 2 + 2`: Unifies `X` with the structure `'+'(2, 2)`, **not** with the number 4.
  - `[X|L] = [1, 2, 3]`: Unifies `X` with `1` and `L` with `[2, 3]`.

- **Generate-and-Test (Inefficient Sorting):**
  - `sort(L, S)` can be thought of as generating permutations of `L` and testing if they are sorted.
  - Highly inefficient (factorial time complexity).
  - Demonstrates the declarative nature: Specifies **what** is desired (a sorted permutation) without specifying **how** to efficiently find it.

11.4.7 Prolog Built-ins

- `append/3` is built in, making the definition redundant
- `sort/2` is also built in, though this uses a more efficient algorithm

12 *<2025-02-18 Tue>* More Logic Programming (Prolog) and Clause Specificity

12.1 Prolog Clauses and Specificity

12.1.1 Review: Prolog Clauses and Logical Statements

- **Clause:** Universally quantified logical statement.
- Example Clause: `a_b(X, f(Y) Z) :- b_c(Z, f(Y) X).`
- English Reading: "For all X, Y, Z, if `b_c(Z, f(Y) X)` is true, then `a_b(X, f(Y) Z)` is true."
- Prolog program: Set of clauses (facts and rules).

12.1.2 Clause Specificity and Generalization

- **Specificity:** One clause is more specific than another if it applies in fewer situations (more constrained).
- **Generalization:** One clause is more general than another if it applies in more situations (less constrained).
- **Example:**
  - **More General Clause:** `a_b(X, B, Z) :- b_c(Z, B, X).`
  - **More Specific Clause:** `a_b(X, f(Y) Z) :- b_c(Z, f(Y) X).`

12.1.3 Substitution Test for Specificity

- Clause C1 is more general than clause C2 if you can obtain C2 from C1 by applying a consistent substitution of terms for variables in C1.
- **Substitution:** Set of assignments of terms to logical variables (e.g., `{B = f(Y) Z, C = Z, A = X}`).

12.2 Prolog Facts, Rules, and Queries

12.2.1 Facts

- Unconditional clauses (always true).
- Example: `prerequisite(cs31, cs131).` (Fact: cs31 is a prerequisite for cs131)
- Ground Terms: Facts often use ground terms (terms without logical variables), but not required (e.g., `prerequisite(intro_one, _).` - `intro_one` is prerequisite for any course).

12.2.2 Rules

- Conditional clauses (true if conditions are met).
- Example: `prt(A, B) :- pr(A, B).` (Rule: A is a prerequisite of B if A is an immediate prerequisite of B)
- Example: `prt(A, Z) :- pr(A, B), prt(B, Z).` (Rule: A is a prerequisite of Z

if A is an immediate prerequisite of B AND B is a prerequisite of Z)

12.2.3 Queries

- Questions asked to the Prolog system to initiate computation and find answers.
- Example Query: `prt(cs31, R).` (Query: Find all courses R for which cs31 is a prerequisite)
- Prolog's Response: List of substitutions for variables that satisfy the query (e.g., `R = cs131.`, `R = cs132.`, ...).
- Proof by Contradiction: Prolog tries to prove the negation of the query is false. Finding substitutions that satisfy the query are counterexamples to the negation.

12.3 Prolog Execution Model: Proof Search

12.3.1 Depth-First, Left-to-Right Search

- Prolog interpreter searches for proofs in a depth-first, left-to-right manner.
- **Order of clauses and subgoals matters** for efficiency and termination.
- **OR Nodes:** When there are multiple clauses that can match a goal, Prolog explores them in order (OR choice).
- **AND Nodes:** When a rule has multiple subgoals (conjuncts), Prolog tries to prove them from left to right (AND sequence).

12.3.2 Backtracking

- If a subgoal fails, Prolog backtracks to the most recent choice point (OR node) and tries the next alternative.
- If all alternatives are exhausted, the query fails (no).
- Backtracking enables Prolog to explore different proof paths and find multiple solutions.

12.3.3 Member Predicate Example

```
member(X, [X|_]).       % Fact 1: X is a member of a list starting with X
member(X, [_|L]) :-     % Rule 2: X is a member of list if it's a member of the tail L
  member(X, L).
```

- **Query 1:** `member(a, [a, b, c]).` - Matches Fact 1 (success, `X = a`).
- **Query 2:** `member(q, [a, b, c]).` - No match with Fact 1, tries Rule 2, recursive calls, eventually fails (no).
- **Query 3:** `member(Q, [a, b, c]).` - Matches Fact 1 (success, `Q = a`), backtracking yields more solutions (`Q=b`, `Q=c`) by Rule 2.
- **Query 4:**

```
member(Q, R).
% We get:
R = [Q|_] ?
```

12.3.4 Append Predicate Example

```
append([], L, L).         % Fact 1: Appending [] to L gives L
append([X|L], M, [X|LM]) :- % Rule 2: Append [X|L] to M gives [X|LM] if append(L, M, LM)
  append(L, M, LM).
append([a, b], [c], R).
append(R, S, [a, b, c]).
append(R, S, T).
% We get:
R = []
T = S ? ;

R = [A]
T = [A|S] ? ;
% ....
```

- **Query 1:** `append([a, b], [c], R).` - Prolog finds `R = [a, b, c]` by applying Rule 2 recursively and then Fact 1.
- **Query 2:** `append(R, S, [a, b, c]).` - Prolog backtracks through different ways to split `[a, b, c]` into two lists `R` and `S` that append to it. Generates multiple solutions (`R=[], S=[a,b,c]; R=[a], S=[b,c]; R=[a,b], S=[c]; R=[a,b,c], S=[]`).
- **Query 3:** `append(R, S, T).` - Generates infinite solutions, Prolog will keep generating longer and longer lists R and S that append to form T (unbounded search space).

12.4 `reverse` Predicate

12.4.1 Inefficient Version (Quadratic Time)

```
reverse([], []).
reverse([X|L], R) :-
  reverse(L, BackwardsL),
  append(BackwardsL, [X], R).
```

- Inefficient because `append` is called repeatedly on growing lists.

12.4.2 Efficient Version (Linear Time - Accumulator)

```
rev([], A, A).
rev([H|T], A, R) :-
  rev(T, [H|A], R).

reverse(L, R) :-
  rev(L, [], R).  % Initial call with empty accumulator
```

- Uses an accumulator (`A`) to build the reversed list efficiently.
- Similar to efficient `reverse` implementation in functional languages.

12.4.3 `lips` (Logical Inferences Per Second)

- Performance measure for Prolog systems (analogous to `flops` for floating-point operations).
- Inefficient `reverse` can be used as a simple benchmark.

12.5 Prolog Predicates: `fail`, `true`, `loop`, `repeat`

12.5.1 `fail/0` Predicate

- Always fails (cannot be proven true).
- Source Code: (None - built-in failure).
- Query `fail.` always returns `no`.
- Used for forcing backtracking in Prolog programs.

12.5.2 `true/0` Predicate

- Always succeeds (unconditionally true).
- Source Code: `true.` (Fact: `true` is true).
- Query `true.` always returns `yes`.
- Used as a placeholder or for unconditional success.

12.5.3 `loop/0` Predicate (Infinite Loop)

```
loop :- loop.  % Rule: loop is true if loop is true (recursive call)
```

- Causes infinite recursion and stack overflow.
- Query `loop.` results in infinite loop or error.
- Example of unintended non-termination in Prolog.

12.5.4 `repeat/0` Predicate (Non-deterministic Success)

```
repeat.       % Fact 1: repeat is true (first way to succeed)
repeat :- repeat. % Rule 2: repeat is true if repeat is true (second way to succeed)
```

- Always succeeds (due to Fact 1).
- Backtracking into `repeat` always leads to another success (due to Rule 2 and recursion).
- Query `repeat.` returns `yes` (using Fact 1), backtracking (;) leads to infinite `yes` answers via Rule 2.
- Used to create loops or generate multiple solutions in Prolog.
- Example:

  ```
    repeat, write('ouch!'), fail.
  ```

  Prints “ouch!” infinitely because `repeat` always succeeds on backtracking, `fail` forces backtracking.

12.5.5 `repeat_e/0` Predicate (Infinite Loop Variant)

```
repeet :- repeet, repeet. % Rule: repeet is true if repeet AND repeat are true
```

- Causes infinite loop because Prolog gets stuck in recursive calls to `repeat_e` without reaching a base case.
- Never reaches the `repeat.` fact, leading to non-termination.

12.6 Clause Order and Proof Search Control

- Order of clauses in a Prolog program is significant because Prolog uses depth-first, left-to-right search.
- Clause order affects:
  - Efficiency: Order of clauses can impact how quickly Prolog finds a solution (or determines no solution exists).
  - Termination: Incorrect clause order can lead to infinite loops (as seen with `repeat_e`).
  - Solution Order: Prolog returns solutions in the order it finds them based on clause order and backtracking.
- Good Prolog programming style often involves:
  - Putting facts before rules.
  - Ordering rules and facts from more specific to more general (for efficiency).
  - Considering the search order when designing predicates to avoid infinite loops and ensure desired behavior.

13 *<2025-02-20 Thu>* Logic Programming (Prolog) - Unification, Control and Scheme

13.1 What Can Go Wrong in Prolog?

13.1.1 Wrong Code (Wrong Logic)

- Since Prolog focuses on logic, incorrect code means the logic is flawed.
- The program doesn't match the programmer's intent.

13.1.2 Debugging in Prolog

1. 4-Port Debugging Model
   - Visualized as a box with four ports: `call`, `success`, `backtrack` (redo), and `fail`.
     - `call`: Initiates execution of a goal.
     - `success`: A goal succeeds (finds a solution).
     - `backtrack`: Backtracks into a goal to find another solution.
     - `fail`: A goal fails (no more solutions).
   - The Prolog interpreter can put "spies" on these ports to trace execution.
   - It prints out information (variable bindings, etc.) when a port is crossed.
   - Possible execution paths for a goal Q:
     - `call` → `success`
     - `call` → `fail`
     - `backtrack` → `success`
     - `backtrack` → `fail`
   - Compared to other languages (C++, OCaml), Prolog has a more complex model due to backtracking. Other languages usually just have `call` and `return`.
2. Built-in Predicate: `trace`
   - Invokes the standard Prolog debugger.
   - Puts hooks in the interpreter to track execution.

- Uses the 4-port model.

3. Manual Debugging (Print Statements)
   - You can insert print statements (e.g., using `write` and `nl`) to inspect variable values.
   - Example:

     ```
     repeat, p(X, Y), write(Y), nl, fail.
     ```

     This prints the value of `Y` for every solution of `p(X, Y)`. The `fail` forces backtracking to find all solutions. The `repeat` makes sure you don't miss anything by using it with `fail`, however, it's better to remove it.

13.2 Unification in Detail

13.2.1 Definition

- The process of making two terms identical by finding a substitution (a set of variable bindings).
- Core to Prolog because it's how goals are matched against clause heads.
- **Example:** Goal: `member(X, [Z, A]).` Clause Head: `member(A, [L|M]) :- member(A, M).` Substitution: `{X = a, Y = _, L = [Z, A]}` (where `_` could be any term - represents an unbound variable). The textbook would use `{X/A, Y/_, L/[Z, A]}`
  - After applying the substitution, both the goal and the clause head become: `member(a, Y [Z, A]).`

13.2.2 Unification vs. OCaml Pattern Matching

- **Prolog Unification:** Two-way pattern matching. Variables can be bound in **both** the goal and the clause head.
- **OCaml Pattern Matching:** One-way pattern matching. Variables are bound only in the **pattern**, not in the **data**.
- **Example:**
  - In OCaml, you match a **value** against a **pattern** with variables. The data being matched **cannot** contain variables.
  - In Prolog, you unify a **goal** with a **clause head**, and **both** can contain variables.
- **Consequence:** Prolog unification is more powerful but can lead to problems (cyclic data structures).

13.2.3 Cyclic Data Structures (Infinite Terms)

- A problem that can arise from two-way unification.
- **Example:** Fact: `p(X, X).` (p is true if its two arguments are the same) Query: `p(Z, f(Z)).`
- Prolog will try to unify `Z` with `f(Z)`, creating a cyclic data structure where `Z` points to `f(Z)`, which contains `Z`, and so on infinitely. Internally represented as `Z = f(Z) = f(f(Z)) = ...`
- This can lead to infinite loops when printing or processing the term.
- Most Prolog systems detect infinite loops during printing and provide a diagnostic.

13.2.4 Implementing Peano Arithmetic (Example)

- Illustrates how to represent numbers and arithmetic operations using Prolog terms.
- **Representation:**
  - 0 is represented by the atom `z`.
  - The successor function `s(X)` represents `X + 1`.
  - Example: 2 is represented as `s(s(z))`.
- **Addition Predicate:**

```
plus(z, X, X).
plus(s(X), Y, s(Z)) :-
    plus(X, Y, Z).
```

  - This is analogous to the `append` predicate for lists.
- **Subtraction Predicate:** Defined in terms of addition. `x - y = z` if `y + z = x`.

```
minus(X, Y, Z) :- plus(Y, Z, X).
```

- **Less Than Predicate:**

```
lt(z, s(_)).
lt(X, s(X)).
lt(X, s(Y)) :- lt(X, Y).
```

- **Problem:** The `lt` predicate, as defined above, can lead to incorrect results due to cyclic data structures. For instance, the query `lt(X, X).` would incorrectly succeed by creating an infinite term. It would deduce the existence of infinity!

13.2.5 `unify_with_occurs_check/2`

- A built-in predicate that performs unification **with** an occurs check.
- **Occurs Check:** Prevents the creation of cyclic data structures by failing if a variable is unified with a term containing that variable.
- **Syntax:** `unify_with_occurs_check(X, Y).` (Equivalent to `X = Y`, but safer).

```
lt_safe(z, s(_)).
lt_safe(X, s(Y)) :-
    unify_with_occurs_check(X, Y).
lt_safe(X, s(Y)) :- lt_safe(X, Y).
```

- **Efficiency: Slower** than ordinary unification `=/2` because it needs to traverse the entire term to check for cycles.
  - Ordinary unification cost: $O(\min(|X|, |Y|))$
  - `unify_with_occurs_check` cost: $O(\max(|X|, |Y|))$
- **Usage:** Necessary for strict logical correctness, but often avoided by Prolog programmers for performance reasons. They manually make sure to avoid generating cyclic structures.

13.2.6 Prolog vs. Logic (the discrepancy)

- Prolog **doesn't perfectly match** mathematical logic because of the occurs check issue (and other control features like `cut`).
- Programmers rely on the rule of thumb of **never create loops**.

13.3 Control That Goes Wrong

13.3.1 Default Control: Depth-First, Left-to-Right

- Prolog's default execution strategy.
- Works well, but sometimes you need more control for efficiency.

13.3.2 `cut` (!) - The Control Primitive

- **Purpose:** To control backtracking and prune the search tree.
- **Syntax:** `!` (exclamation point).
- **Behavior:**
  - Always succeeds when first encountered.
  - **Side effect:** Removes all choice points (alternative clauses) for the **current predicate** in which the `cut` appears.
  - If you backtrack into a `cut`, the **calling predicate** fails immediately.
- **4-Port Model of Cut:** A box that succeeds immediately but causes the calling predicate to fail if backtracked into.
- **Example (membercheck):**

```
p(X, Z) :- q(X, Y), member(X, Z), r(X, Y, Z).
X = a.
Z = [a,b,r,a,c,a,d,a,b,r,a].
% Suppose q is for generating test cases with lots of successes.
% This would be inefficient if we had to check all the duplicates.
% Especially if r is expensive and fails on duplicates.

% We want to check for duplicates in Z and stop at the first one.
% We can use membercheck which is implemented as follows:
membercheck(X, [X|_]) :- !.
membercheck(X, [_|L]) :- membercheck(X, L).
```

  - The `cut` prevents `membercheck` from succeeding multiple times if there are duplicate elements in the list. Without the `cut`, backtracking would lead to redundant calls and inefficiency.
- **Scope of Cut:** The predicate in which it appears. It only affects backtracking within that predicate.

13.3.3 `once/1` (Meta-Predicate)

- A predicate that uses `cut` to ensure that a goal succeeds only once.

```
once(P) :-
    call(P), !.
% Once p succeeds, the cut prevents any further backtracking into P.
```

- Useful for avoiding redundant solutions from predicates like `member`.
- **Example:** `once(member(X, Z)).` would only find the first occurrence of `X` in `Z`.
- `once` can be defined in terms of a `cut`, showing how control predicates are built up.

13.3.4 Cut Placement and its Effects

1. Cut at the beginning of a clause

```
b_once(P):- !, call(P).
```

Placing the cut before P means that once b_{once} is called, only the first solution of P, will be considered, so that backtracking into `b_once` will **not** cut off alternatives to P.

2. Cut at the end

```
p(X, Y, Z) :- q(X, Y), member(X,Z), !, r(X, Y, Z).
```

This cut will commit and not try any other alternative

13.3.5 Negation as Failure (\backslash plus/1 or \+/1)

- Implemented using `cut` and `fail`.

```
\+(P) :- call(P), !, fail.
\+(P).
```

- `\+ P` succeeds if `P` fails, and fails if `P` succeeds.
- **Problems:**
  - Doesn't correspond to logical negation. It means "not provable," not "false."
  - $\vdash \equiv \models$ is assumed in prolog. (`\vdash` means provable, `\vDash` means true).
  - Can lead to unexpected results due to Prolog's closed-world assumption.
  - **Example:**

```
% Suppose we have all the facts about classes and their prerequisites.
% We could ask:
?- \+ prerequisite(dance10, cs131).
% It only shows that this is not provable, not that it is false.
% Logically, to prove it false, we need to have all the fact that says a class
% is not a prerequisite of another.
```

Closed-World Assumption (CWA): If something isn't provable, it's assumed to be false. This is not always valid in the real world.

  - Breaks commutativity of AND. `X = 5, \+ X = 6.` succeeds, but `\+ X = 6, X = 5.` fails. This implies that in prolog, the and (`,`) operator is not commutative.

13.4 Theory: Propositional Logic, First-Order Logic, and Prolog

13.4.1 Propositional Logic

- Simplest form of logic.
- **Propositions:** Statements about the world (e.g., "It's raining"). Represented by letters (P, Q, R...).
- **Connectives:** Combine propositions.
  - Negation ($\neg P, !P$), etc.): "not P"
  - Conjunction ($P \wedge Q, P.Q, P, Q$), etc): "P and Q"
  - Disjunction ($P \vee Q, P + Q$), etc): "P or Q"
  - Exclusive Or ($P \oplus Q$): "P or Q, but not both"
  - Implication ($P \rightarrow Q, P \supset Q$), etc): "If P, then Q" (In C/C++, $P \leq Q$)
  - Reverse Implication ($P \leftarrow Q$): "P if Q" (In C/C++, $P \geq Q$)
- **Truth Tables:** Define the truth value of compound propositions based on the truth values of their components.

- **Tautology:** A statement that is always true, regardless of the truth values of its propositions (e.g., $(P \to Q) \land (Q \to P) \to (P \leftrightarrow Q)$).
- **Limitation:** Propositional logic can't express relationships between objects or quantify over variables.

13.4.2 First-Order Logic (Predicate Calculus)

- What prolog is based on.
- Extends propositional logic with:
  - **Logical Variables:** Represent objects in the domain.
  - **Quantifiers:**
    - Universal Quantifier ($\forall$): "for all"
    - Existential Quantifier ($\exists$): "there exists"
  - **Predicates:** Propositions with arguments (e.g., `man(X)`, `mortal(X)`).
- **Example (Socrates Syllogism):**
  - $\forall X(man(X) \to mortal(X))$ (All men are mortal)
  - `man(Socrates)` (Socrates is a man)
  - Therefore, `mortal(Socrates)` (Socrates is mortal)
- In first-order logic, this syllogism becomes a tautology.

13.4.3 Prolog and Logic

- Prolog is an attempt to implement a subset of first-order logic for computation.
- We could not have a system that, given any first-order logic statement, could compute the truth value of that statement.
- This is due to the undecidability of first-order logic (Gödel's Incompleteness Theorem).
- Prolog is only doing as good as possible.
- Logicians have developed procedures for proving theorems in first-order logic.
- A standard approach is to convert the statement to be proven into **Conjunctive Normal Form (CNF)**.
- **CNF:** A conjunction (AND) of clauses.
- **Clause:** A disjunction (OR) of literals.
- **Literal:** An atomic formula or its negation. An atomic formula is a predicate symbol with terms as arguments (e.g., `p(x, f(y))`).
- **General Clause Form:** $(A_1 \land A_2 \land ... \land A_m) \to (C_1 \lor C_2 \lor ... \lor C_n)$
  - A_i: Antecedents (conditions).
  - C_i: Consequents (possible conclusions).
  - The implication means: "If all the antecedents are true, then **at least one** of the consequents must be true."
  - It can also be written in a fully disjunctive way:

  $(\neg A_1 \lor \neg A_2 \lor ... \lor \neg A_m \lor C_1 \lor C_2 \lor ... \lor C_n)$
- **Example:** `dog(X) | cat(X) | pig(X) :- in_santa_monica(X), licensed(X).`

- Antecedents: `in_santa_monica(X)`, `licensed(X)`.
- Consequents: `dog(X)`, `cat(X)`, `pig(X)`.
- Meaning: "If X is in Santa Monica and X is licensed, then X is a dog, a cat, or a pig."
- Equivalent disjunctive form: $\neg$ `in_santa_monica(X)` $\vee\neg$ `licensed(X)` $\vee$ `dog(X)` $\vee$ `cat(X)` $\vee$ `pig(X)`.

- **Conversion to CNF:** Any statement in first-order logic can be converted to an equivalent CNF representation through a series of logical transformations (e.g., eliminating implications, moving negations inwards, distributing AND over OR).

13.4.4 The Problem with General Clauses

- Proving theorems with general clauses (arbitrary numbers of antecedents and consequents) is computationally very difficult (in general, undecidable; in practice, often intractable).

13.4.5 Horn Clauses (The Prolog Restriction)

- Prolog uses a restricted form of clauses called **Horn clauses** to make theorem proving more efficient.
- **Horn Clause:** A clause with **at most one** positive literal (consequent). That is, $\mathbf{n} \leq 1$ in the general form above.
- **Why Horn Clauses?** They allow for a more efficient proof procedure (resolution with SLD-resolution, which is the basis for Prolog's execution).
- **The Example is NOT a Horn Clause:** The example above (`dog(X) | cat(X) | pig(X) :- ...`) is **not** a Horn clause because it has three consequents. Prolog cannot directly handle such clauses.

13.4.6 Types of Horn Clauses

- Since a Horn clause has at most one consequent (n $\leq$ 1), there are three possibilities:

1. **n = 1, m = 0 (Fact):**
   - Form: C (a single consequent, no antecedents).
   - Prolog: `c.` (e.g., `mortal(socrates).`)
   - Meaning: "C is unconditionally true."
2. **n = 1, m > 0 (Rule):**
   - Form: $(A_1 \wedge A_2 \wedge ... \wedge A_m) \rightarrow C$
   - Prolog: `c :- a1, a2, ..., am.` (e.g., `mortal(X) :- man(X).`)
   - Meaning: "If all the antecedents (A1, A2, ..., Am) are true, then the consequent (C) is true."
3. **n = 0, m $\geq$ 0 (Query/Goal):**
   - Form: $(A_1 \wedge A_2 \wedge ... \wedge A_m) \rightarrow false$ (or equivalently, $\neg(A_1 \wedge A_2 \wedge ... \wedge A_m)$)
   - Prolog: `?- a1, a2, ..., am.` (e.g., `?- man(X), mortal(X).`)
   - Meaning: "Is it possible for all the antecedents (A1, A2, ..., Am) to be true simultaneously?"
   - Prolog uses **proof by contradiction**: It tries to prove that the negation of the query is false. If it finds a substitution for the variables that makes the antecedents true, that's a counterexample to the negation, and thus a solution to the query. It is equivalent to $(\neg A_1 \vee \neg A_2 \vee ... \vee \neg A_m)$

13.5 Scheme

13.5.1 Introduction

- Scheme is a simple dialect of Lisp.
- Almost a subset of ML/OCaml, but with a very simple syntax.

13.5.2 Two Main Topics in Scheme

1. **Programs as Data:** Scheme programs can be easily represented as data structures, making meta-programming (writing programs that manipulate other programs) straightforward.
2. **Continuations:** A low-level control structure similar to `goto` in C/C++ or `cut` in Prolog, but more powerful and elegant.

13.5.3 Basic Scheme Syntax

- Function calls: `(function arg1 arg2 ... )`
- **Example:** `(+ 2 2)` (adds 2 and 2)
- Nested calls: `(f (g x) y)` (calls g with x, then calls f with the result and y)
- Or something weird like this: `((f 3) 5)`
- **Data as Code:** The same syntax is used for both code and data.

13.5.4 `cons`, Pairs, and Lists

- **`cons`:** The fundamental building block (short for "construct").
  - Creates a **pair** (a data structure with two fields, often called `car` and `cdr`).
  - Syntax: `(cons first second)`
  - Example: `(cons 10 20)` creates a pair where the first element is 10 and the second is 20.
- **Lists:** Built by chaining `cons` cells.
  - The `car` of each cell holds a list element.
  - The `cdr` of each cell points to the next cell, or to the **empty list** (`'()`) to mark the end of the list.
  - Example:

```
(cons 10 '()) ; Creates the list (10)
(cons "x" (cons 10 '())) ; Creates the list ("x" 10)
```

- **Empty List:** `'()` (or just `()`) represents the empty list.
- **Heterogeneous Lists:** Scheme lists can contain elements of different types (unlike OCaml's statically typed lists). Type checking happens at runtime.

13.5.5 Building Programs as Data (Example)

- We can create the data structure representing `((f 3) 4)` using `cons`:

```
(cons (cons 'f (cons 3 '())) (cons 4 '()))
```

  - `'f`: The **symbol** `f` (more on quoting below).
  - This creates the list structure, but it **doesn't** call the function `f`. It's just data.

13.5.6 `quote` (') and Symbols

- `quote` (**'**): Prevents evaluation. Returns the expression **literally**, without interpreting it as code.
- **Symbols:** Similar to atoms in Prolog. Represent names or identifiers.
  - Example: `'f` evaluates to the **symbol** `f`, not the **value** of a variable named `f`.
- **Quoting Lists:** `'(1 2 3)` creates the list containing 1, 2, and 3, without trying to evaluate 1, 2, or 3 as variables or functions. This is equivalent to: `(cons 1 (cons 2 (cons 3 '())))`
- **Syntactic Sugar:** `'x` is equivalent to `(quote x)`. The single quote is just a shorthand.
- **Nested Quoting:** `"a` (or `(quote (quote a))`) evaluates to the list `(quote a)`.
- Quoting is essential for representing code as data because it prevents the code from being executed prematurely.

13.5.7 Improper Lists

- Scheme allows the creation of "improper" lists, where the final `cdr` does **not** point to the empty list.
- **Example:** `(cons 3 (cons 4 5))`
- Scheme prints improper lists using a dot notation: `(3 4 . 5)`

14 *<2025-02-25 Tue>* More Scheme

14.1 Characteristics of Scheme-like Languages

14.1.1 Simple Syntax

- Programs are easily represented as data. This is unlike C++, where representing programs as strings is cumbersome.

14.1.2 Dynamic Memory Allocation with Garbage Collection

- Objects are allocated dynamically and never explicitly freed. A garbage collector reclaims unused memory.
  - Like: Java, JavaScript, Python, ML.
  - Unlike: C++, C.

14.1.3 Dynamic Type Checking

- All type checking is done at runtime.
  - Like: Python.
  - Unlike: C++, Java, ML (which have static type checking).

14.1.4 Static Scoping (Lexical Scoping)

- The scope of identifiers (where they are visible) can be determined at compile time by analyzing the program text.
  - Like: Most modern languages (Java, Python, ML, C++).
  - **Unlike:** Older versions of Lisp (dynamic scoping), and shell environment variables.
- **Dynamic Scoping Example (Lisp, Shell):**

```
(define f (x) (+ x y))          ; y is a free variable in f
(let ((y 12))
  (f 13))
(let ((y 19))
  (f 3))
```

  - In dynamic scoping, the meaning of a free variable (like `y` in `f`) depends on the **caller's** environment. The provided code illustrates this, with `f` accessing the dynamically bound `y`.

- **Advantages of Static Scoping:**
  - Easier to reason about code (you can see where variables are defined).
  - Enables more compiler optimizations.
- **Advantages of Dynamic Scoping:**
  - More flexible in some cases (e.g., passing environment information implicitly). Think about how shell scripts would work **without** environment variables.

14.1.5 Call by Value

- When a function is called, the arguments are evaluated, and **copies** of the values are passed to the function.
  - Like: ML, Java, Python, C++ (mostly).
  - Unlike: C++ (call by reference).
- Scheme uses object semantics, so "copies" are actually copies of pointers to objects.

14.1.6 First-Class Procedures (Functions)

- Procedures (functions) are treated as objects, just like numbers, strings, or lists.
- You can:
  - Pass functions as arguments to other functions.
  - Return functions as values from other functions.
  - Store functions in data structures.
- Includes **continuations** (a special, low-level type of procedure).

14.1.7 High-Level Arithmetic

- Supports complex numbers, integers of arbitrary size, and even floating-point numbers of arbitrary precision.

14.1.8 Tail Recursion Optimization (TRO) / Tail Call Optimization (TCO)

- A crucial optimization for functional programming.
- **Tail Call:** A function call that is the **last** operation performed in a function **before returning**. The function's result is **immediately** returned as the result of the caller.
- **Tail Recursion:** A special case of a tail call where a function calls **itself** in a tail position.
- **Optimization:** In a tail call, the caller's stack frame is **no longer needed**. The callee can reuse (or replace) the caller's stack frame.
- **Example:**

```
int factorial(int n) { return n == 0 ? 1 : n * factorial(n - 1); }
```

```
;; Incorrect Scheme version (non-tail recursive).  Note the misplaced 'n' in the multiplication.
(define (factorial n)
  (if (zero? n) 1
      (* (factorial (- n 1)) n)))  ; The call to * is the last operation, not the recursive call.
```

  - The call to `*` (or equivalent operator) is the **last** operation. The call to `factorial` is **not** a tail call because the result must be multiplied by `n` **after** the recursive call returns. This version will grow the stack.

- **Example (Tail Recursive Factorial):**

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

(define (fact n a)  ; Inner helper function with accumulator
  (if (zero? n)
      a
      (fact (- n 1)
            (* a n))))
```

  - The recursive call to `fact` is now a **tail call**. The multiplication is done **before** the recursive call.
  - The `a` parameter is an **accumulator** that accumulates the intermediate result.
  - This version **will not** grow the stack (thanks to TRO).

- **Corollary:** Tail-recursive functions are equivalent to loops. Scheme compilers often transform tail-recursive calls into iterative loops.

- **Named Let:** A special form in Scheme that provides a convenient syntax for writing tail-recursive functions. It looks like a `let`, but it defines an internal, named, recursive function.

```
(define (factorial n)
  (let fact ((n' n) (a 1))  ; Named let: defines a function called 'fact'
    (if (zero? n')
        a
        (fact (- n' 1)
              (* n' a))))) ; Corrected recursive call with accumulator update.
```

  - `fact`: The name of the internally defined recursive function.
  - `n'` and `a`: Parameters of the internal function. (Using `n'` to distinguish it from the outer `n`).
  - `n` and `1`: Initial values passed to the internal function when `factorial` is called.

14.2 Scheme Syntax

14.2.1 Identifiers

- More flexible than in many languages.
- Allowed characters: Letters, digits, +, -, ., ?, <, =, >, :, $, %, ^, &, _, ~, @, *.
- Cannot start with: Digit, `+`, `-` .
- **Exceptions:** `...` (literally 3 dots), single `+`, single `-`, identifiers starting with `->` (convention for type conversion functions, e.g., `integer->float`).

14.2.2 Comments

- Start with a semicolon (`;`).

14.2.3 Lists

- `(item1 item2 ... itemN)`
- Items are **not** separated by commas.

14.2.4 Improper Lists

- `(item1 item2 . last)`
- The final `cdr` does not point to the empty list.

14.2.5 Empty List

- `'()` or `()`

14.2.6 Booleans

- `#t` (true)
- `#f` (false) - **Only** `#f` is false; everything else is true (including 0, 0.0, empty list, etc.).

14.2.7 Vectors

- `#(item1 item2 ... itemN)`
- Fixed-size, contiguous sequences (like arrays). Different internal representation from lists.

14.2.8 Strings

- `"This is a string\nwith a newline."` (same as C/C++).

14.2.9 Characters

- `#\c` (character c)
- `#\newline`

14.2.10 Numbers

- `-19` (integer)
- `3.14` (floating-point)
- `2/3` (exact rational number)

14.2.11 `quote` (`'`) and Quasiquote (`` ` ``)

- `'expr`: Equivalent to `(quote expr)`. Prevents evaluation of `expr`. Returns `expr` literally.
- `` `expr ``: (backtick or grave accent) **Quasiquote**. Similar to `quote`, but allows selective **unquoting** within the expression.
- `,expr`: (comma) **Unquote**. Inside a quasiquoted expression, `,expr` **evaluates** `expr` and inserts the result.
- Example:

```
(define b 37)
`(a ,b c (d e))      ; => (a 37 c (d e))
`(a ,b c ,(append '(d) '(e))) ; => (a 37 c (d e))
```

  - Quasiquote is like a template, and unquote lets you fill in slots in the template.

14.2.12 Special Forms

- Expressions that **look** like function calls but have special evaluation rules. They are **not** functions.
- Examples: `define`, `if`, `lambda`, `let`, `quote`, `quasiquote`, `define-syntax`.
- **Why Special Forms?** They cannot be implemented as ordinary functions because of Scheme's call-by-value semantics.
  - Example: `if` **must** be a special form. If it were a function, both the "then" and "else" branches would always be evaluated, leading to incorrect behavior (and potentially infinite recursion or errors).

14.3 `lambda` Expressions (Nameless Functions)

- `(lambda (arg1 arg2 ...)  body)`
- Creates a function without giving it a name.
- Like anonymous functions in OCaml (`fun x y -> ...`), C++ (`[](int x, int y) { ... }`), Java, etc.
- Can be nested.
- Example (currying):

```
(lambda (x)
  (lambda (y)
    (+ x y)))
```

- Variadic functions (a function of indefinite arity):

```
(define printf (lambda (format . args) (...))) ; 'printf' for example purposes. Not the actual Scheme
↪  printf
(printf "%d hello %s" 19 "abc")
```

  - `format`: The first argument (mandatory).
  - `. args`: The **rest** of the arguments are collected into a **list** named `args`. This is how Scheme supports variable-arity functions.
  - The dot (`.`) before `args` is **crucial**. It indicates a "rest parameter."

```
(lambda args ...) ; Collects *all* arguments into a list named `args`.
```

- Shorthand for defining functions:

```
(define (f x y) (+ x (* y 3)))  ;; Equivalent to (define f (lambda (x y) (+ x (* y 3))))
(define (printf format . args) ...) ;; Equivalent to (define printf (lambda (format . args) ...))
(define (list . x) x)           ;; Equivalent to (define list (lambda x x)) ; The actual 'list' function.
```

14.4 `let` and Scoping

- `let` creates local bindings.
- **Scoping Rule:** The scope of variables introduced by `let` extends to the **body** of the `let`, but **not** to the initial value expressions.

```
(let ((x (+ a 3))
      (y (* x 2)))  ; This x refers to an outer x, NOT the x just defined.
  (+ (* a x) (- y x)))
```

  - C equivalent showing the scoping difference:

```
  {
    int a = /*some value*/;
    int x = a + 3;
    int y = x * 2; // This x *does* refer to the newly defined x.
    return a * x + (y - x);
  }
// Another, more dangerous C example
      : int x = sizeof(x); // Legal, but dangerous in C. Scope of x *does* extend
                           // to initializer.
  {
    int x = x + 5; // Accesses an uninitialized variable.
  }
```

- **`let` as Syntactic Sugar for `lambda`:** A `let` expression is equivalent to a call to a nameless function (lambda expression).

```
(let ((x (+ a 3))
      (y (* x 2)))
  (+ (* a x) (- y x)))

;; Is equivalent to:

((lambda (x y)
   (+ (* a x) (- y x)))
 (+ a 3)  ; Initial value for x
 (* x 2))  ; Initial value for y (uses the *outer* x)
```

- Understanding this equivalence clarifies the scoping rules of `let`.

14.5 `and` and `or` as Special Forms

- `and` and `or` are special forms, **not** functions. They provide **short-circuit evaluation**.
- `(and e1 e2 ... en)`: Evaluates `e1`, `e2`, ..., `en` from left to right. If any expression evaluates to `#f`, the `and` immediately returns `#f`. Otherwise, it returns the value of `en`.
- `(or e1 e2 ... en)`: Evaluates `e1`, `e2`, ..., `en` from left to right. If any expression evaluates to a true value (anything other than `#f`), the `or` immediately returns that value. Otherwise, it returns `#f`.

14.6 Tail Call Contexts

- A tail call is not simply the **textually** last expression in a function. It's the last expression **evaluated** before returning.
- Tail call contexts:
  - The last expression in a `lambda` body.
  - The "then" and "else" branches of an `if`, if the `if` itself is in a tail call context.
  - The last expression in an `and` or `or`, if the `and` or `or` itself is in a tail call context.
  - The last expression in a `let` body, if the `let` itself is in a tail call context.
  - There are some others as well (`cond`, for example).
- **Example:**

```
(define (f n)
  (if (zero? n)
      (f x)  ; Tail call, even though it's not textually last.
      37))
```

14.7 `define-syntax` (Macros)

- Allows you to define your own special forms (macros).
- Similar to macros in C/C++, but **hygienic** (avoids accidental variable capture).
- **Example (Defining `and`):**

```
(define-syntax and
  (syntax-rules ()
    ((and) #t)                          ; (and) => #t
    ((and x) x)                         ; (and x) => x
    ((and x y ...)                      ; (and x y ...) => (if x (and y ...) #f)
     (if x (and y ...) #f))))
```

  - `syntax-rules`: Keyword indicating a macro definition.
  - `()`: List of keywords used in the macro (none in this case).
  - `((pattern) replacement)`: Defines a pattern and its corresponding replacement.
  - `...`: Matches zero or more occurrences of the preceding pattern.
  - **Expansion Example**:

```
(and (< x 1) (< y 3) (p x))
;; Expands to (using the rules above):
(if (< x 1) (if (< y 3) (p x) #f) #f)
```

- **Example (Defining `or`):** (More complex due to the need to avoid double evaluation).

```
(define-syntax or
  (syntax-rules ()
    ((or) #f)
    ((or x) x)
    ((or x y ...)
     (let ((xc x))      ; introduce a temporary variable xc to store the value of x.
       (if xc xc (or y ...)))))))
```

- **Incorrect `or` (Double Evaluation):**

```
;;INCORRECT
(define-syntax or
  (syntax-rules ()
    ((or) #f)
    ((or x) x)
    ((or x y ...)
     (if x x (or y ...))))) ; Incorrect - evaluates x twice!
```

  ;; Example that demonstrates the problem:

```
(if (getenv "PATH")  ; If PATH exists, return its value.
    (getenv "PATH")  ; Incorrect: Calls getenv *twice*.
    "/bin:/usr/bin") ; Could be expensive or have side effects.
```

- **Hygienic Macros:** Scheme macros avoid accidental variable capture, a common problem with C/C++ macros. The Scheme compiler renames variables internally to prevent collisions.

```
(let ((xc 93)) ; Simulate an existing variable
  (or (getenv "q") xc))

(let ((xc (getenv "PATH")))  ; The macro introduces its *own* xc.
  (if xc xc xc))             ; No collision with the outer xc.
```

  - Scheme uses static scoping before expanding the macro, not afterwards.
  - The macro expander effectively renames the introduced `xc` to avoid conflict with existing variables. This ensures hygiene.

15 <2025-02-27 Thu> More Scheme, Implementation, and Continuations

15.1 Review: Scheme Characteristics and Design Principles

15.1.1 Simplicity

- Scheme aims for a small, core set of **primitives** and builds everything else in a **library**.
  - **Primitives:** Built-in features that cannot be defined in terms of anything simpler (e.g., `if`, `lambda`).
  - **Library:** Functions and macros defined in terms of the primitives (e.g., `not`, `and`, `or`).

15.1.2 Standardized vs. Implementation-Specific

- **Required:** Features that every Scheme implementation **must** provide (e.g. int).
- **Optional:** Features that implementations **may** provide (e.g., multiple-precision floating-point).
- **Extensions:** Implementation-specific features not covered by the standard. Using extensions makes code less portable.

15.2 Categories of Mistakes/Bugs in Scheme

15.2.1 Implementation Restrictions

- Valid Scheme code that fails due to limitations of a specific implementation (e.g., running out of memory).

15.2.2 Unspecified Behavior

- The Scheme standard does not specify the exact behavior in certain situations. Implementations can choose, but **portable** code should not rely on any particular choice.
- **Example:** `(eq? 0 0)` ; Might return `#t` or `#f`, depending on whether the implementation uses the same object for small integers.
  - **Equality Predicates:**
    - `eq?`: Object identity (pointer comparison). $O(1)$
    - `eqv?`: "Contents" comparison (non-recursive). Checks if values are the **same type** and have the **same content**. $O(N)$, where N is the size of the value in the worst case (comparing 2 different strings).
    - `equal?`: Recursive comparison (works for lists, etc.). Potentially infinite loop for cyclic structures.
    - `=`: Numeric comparison. $O(N)$ where N is the number of bits for integers.

```
(eq? "a" "a") ; might be #f
(eqv? "a" "a") ; #t. same type and same content.
(equal? '(a b . c) '(a b . c)); might loop forever if the lists contain circles.

;; NaN examples
(let ((x (/ 0.0 0.0)))
  (eq? x x))    ; => #t (likely, NaN is often a single object)
(= (/ 0.0 0.0) (/0.0 0.0)) ; #f
```

15.2.3 An Error is Signaled

- The implementation **must** report an error and stop execution (or enter an error handling mechanism).
- **Example:** `(open-input-file "foo")` if "foo" does not exist.
- **Example:** `(car 39)` ; In Racket, this will signal an error. Older Scheme implementations might have undefined behavior.

15.2.4 Undefined Behavior

- The **worst** kind of error. The standard says the implementation can do **anything**. No constraints whatsoever.
- Common in low-level languages like C/C++ (e.g., dereferencing a null pointer, array bounds errors).
- Scheme strives to avoid undefined behavior where reasonable, but it still exists in certain situations (often related to type errors or resource exhaustion).

15.3 Continuations: "The Essence of Scheme" (and Controversial)

15.3.1 What is a Continuation?

- A **data structure** that represents the **future** of a computation - "everything that needs to be done from this point until the program finishes." A "bucket list" of computations.
- **Components of a Continuation:**
  1. **Instruction Pointer (IP):** What to execute **next** (within the current function).
  2. **Environment Pointer (EP):** The context for execution (includes the stack frame, variable bindings, etc.). Also includes what happens upon **return** from the function.
- **Relationship to Stack Frames:** A continuation conceptually represents the **entire call stack** (plus the current instruction pointer).
- **Implementation Note:** Continuations don't **copy** the entire stack. They typically store a **pointer** to the relevant stack frame (the EP) and a pointer to the next instruction (the IP).
- **Every** programming language has continuations (implicitly). It's how function calls and returns are implemented. Scheme makes them **explicit** and accessible to the programmer.

15.3.2 `call-with-current-continuation` (aka `call/cc`)

- The core primitive for working with continuations.
- **Syntax:** `(call/cc proc)`
  - `proc`: A **procedure** (function) that takes **one** argument.
- **Behavior:**
  1. Creates a new continuation object (let's call it `k`) pointing to the current execution state (IP and EP pointer pair).
  2. Calls `proc` with `k` as its argument.
  3. Returns whatever `proc` returns.

15.3.3 Using Continuations

- A continuation (`k`) is itself a **procedure** (function) that takes **one** argument.
- **Calling a Continuation:** `(k value)`
  - **Effect: Abandons** the **current** continuation (the current "future") and **restores** the continuation represented by `k`. It's a **non-local jump**.
    1. Sets the return value register (e.g., `%rax` on x86-64) to `value`.
    2. Sets the environment pointer (EP) to the EP stored in `k` (effectively restoring the stack).
    3. Sets the instruction pointer (IP) to the IP stored in `k` (jumping to the saved location).

- It's like a `goto`, but you can jump **out of** functions, or even **back into** functions that have already returned!
- Calling `(k, v)` is like doing the following:

```
%rax = v %rbp = k's ep %rip = k's ip,
```

15.3.4 Example: `prod` (Product of a List) with Early Exit

```
(define (prod ls)
  (call/cc
   (lambda (break) ; 'break' will be the continuation.
     (let pr ((ls ls))
       (if (null? ls)
           1
           (if (zero? (car ls))
               (break 0) ; Immediately return 0 from 'prod'.
               (* (car ls) (pr (cdr ls)))))))))
```

- `call/cc`: Captures the current continuation (the point **after** the `call/cc` call) and passes it to the lambda as the `break` function.
- `break`: If called, immediately returns from the **outer** `prod` function (not just the inner `pr` function). This is the non-local jump.
- This version avoids unnecessary multiplications if a 0 is encountered.

15.3.5 Green Threads Example (Simplified Concurrency)

- Green threads are a form of concurrency where multiple threads of execution are managed within a single operating system process (and typically on a single CPU core). They are **not** true parallel threads.
- Advantages: Simpler than true threads (fewer race conditions).
- Disadvantages: No true parallelism (cannot utilize multiple cores).
- Implemented using continuations to save and restore execution states.

```
(define gt-list '())  ; List of green threads (initially empty).

;; Constructor for green threads.
(define (gt-cons thunk)
  (set! gt-list (append gt-list (list thunk)))) ; Add the 'thunk' to the end of the list.

;; Start running the green threads.
(define (start)
  (let ((next-gt (car gt-list)))   ; Get the first thread.
    (set! gt-list (cdr gt-list))  ; Remove it from the list.
    (next-gt)))                   ; Call the thunk (execute the thread).

;; Yield control to another green thread.
(define (yield)
  (call/cc
   (lambda (k)
     (gt-cons k)    ; Add *current* continuation to the end of the list.
     (start))))       ; Start the next thread.

;; Example usage
(gt-cons (lambda ()
           (let f ()
             (display "h")
             (yield)
             (f))))
(gt-cons (lambda ()
           (let f ()
```

```
            (display "i")
            (yield)
            (f))))
(gt-cons (lambda ()
           (let f ()
            (newline)
            (yield)
            (f))))
(start)
;; Output will be:
;; hi
;; hi
;; hi
;; (forever).
```

- **`gt-list`:** A global variable holding a list of "runnable" green threads (represented as continuations or thunks).
- **`gt-cons`:** "Constructor" - adds a new **thunk** (a function of no arguments) to the end of `gt-list`.
- **`start`:** Removes the first thunk from `gt-list` and executes it.
- **`yield`:** This is the key.
  1. Captures the **current** continuation (`k`) using `call/cc`.
  2. Adds `k` to the **end** of `gt-list` (making the current thread runnable again later).
  3. Calls `start` to run the next thread in the list. The call to `start` **transfers control** and does **not** return to the `yield` function until `k` is invoked.
- The `hi\n` example demonstrates how `yield` allows the threads to interleave their execution.

15.3.6 Continuation-Passing Style (CPS)

- An alternative way to program that makes continuations **explicit without** using `call/cc`.
- **Key Idea:** Every function takes an **extra** argument, which is its continuation (represented as a function).
- Instead of returning a value directly, a function **calls its continuation** with the result.

```
;; CPS version of 'prod'
(define (prod ls break) ; 'break' is now an *explicit* continuation.
  (let pr ((ls ls) (k break))  ; Inner function also takes a continuation 'k'.
    (if (null? ls)
        (k 1)          ; Call the continuation with the result (1).
        (if (zero? (car ls))
            (break 0)  ; Call the *top-level* continuation with 0.
            (pr (cdr ls)
                (lambda (n) (k (* (car ls) n))))))))) ; Create a *new* continuation.
;; This new continuation represents "what to do after the recursive call to pr completes."
(prod '(3 -9 912389712 0 57) (lambda (x) x)) ; Initial call with the identity function as continuation.
```

- The `break` argument to the top-level `prod` function is the initial continuation. It's the identity function `(lambda (x) x)`, which simply returns its argument.
- The inner `pr` function also takes a continuation argument, `k`.
- When a 0 is found, `break` (the initial continuation) is called directly with 0.
- Otherwise, a **new** continuation is created using `lambda`. This new continuation represents the "rest of the computation" **after** the recursive call to `pr`. It takes the result of the recursive call (`n`), multiplies it by the current element `(car ls)`, and then calls the **previous** continuation (`k`) with the final result. This chains the continuations.

- **Advantages of CPS:**
  - Works in languages that don't have `call/cc`.
  - Can make control flow very explicit.
  - Can be used for compiler optimizations.
- **Disadvantages of CPS:**
  - Code can be more complex and harder to read (extra continuation argument everywhere).
  - Can be less efficient than direct `call/cc` (creating lots of small lambda functions).

15.4 Memory Management

15.4.1 What Needs to be Stored in Memory?

- Contents of variables (especially large ones like arrays).
- Return addresses (where to jump back to after a function call).
- Saved environment pointers (to restore the caller's context).
- Instructions (the program itself).
- I/O buffers (and other system-level data).

15.4.2 Traditional Memory Layout (C, etc.)

- **Text Segment:** Instructions (read-only). Also, read-only data.
- **Data Segment:** Initialized static data (global variables with initial values).
- **BSS Segment:** Uninitialized static data (usually zeroed out).
- **Heap:** Dynamically allocated memory (grows upwards). `malloc`, `new`.
- **Stack:** Function call frames (grows downwards). Local variables, return addresses, saved registers.

15.4.3 Simplest Model: Fortran (1958)

- **Everything** is allocated statically (at compile time).
- No stack, no heap, no dynamic memory allocation.
- **Advantages:** Simple, fast, no memory exhaustion errors.
- **Disadvantages:** No recursion, inflexible (array sizes must be fixed at compile time), security issues.

15.4.4 C Model (1975)

- **Stack:** Used for function call frames (activation records). Supports recursion. Local variables live on the stack. Allocation/deallocation is very fast (adjusting the stack pointer). LIFO (Last-In, First-Out) allocation.
- **Heap:** Used for dynamic memory allocation (`malloc`, `free`). More flexible (allocation/deallocation in any order), but slower.

15.4.5 Algol 60 Model

- Like C, but allows **dynamically sized** local arrays. This makes stack frame management more complex.

The key takeaway is that Scheme's continuations, although a unique feature in terms of direct programmer access, are fundamentally related to how **all** programming languages manage execution flow and memory. The `call/cc` primitive just exposes this underlying mechanism. Continuation-passing style is a way to make continuations explicit **without** needing `call/cc`, and it highlights the connection to concepts like callbacks and acceptors.

16 *<2025-03-04 Tue>* Memory Management, Heap Management, Garbage Collection

16.1 Activation Records (Frames) - Review

- Store information about the currently executing function, its caller, etc.
- Traditionally, the stack and heap were distinct. Stack: LIFO. Heap: "Last in, anything out" (objects can be freed in any order).

16.2 Blurring the Stack/Heap Distinction

- **Currying** and similar techniques break the strict separation.
- **Example (Scheme):**

```
(define (f x)
  (lambda (y) (+ x y)))

(f 12)
```

- `(f 12)` returns a **function object**. This object needs:
  - **Instruction Pointer (IP):** Points to the code for `(+ x y)`.
  - **Environment Pointer (EP):** Points to `f`'s activation record (where `x`'s value, 12, is stored).
- **Fat Function Pointers:** Function pointers now need two parts (IP and EP).
- **Consequence:** The inner `lambda` (that adds `x` and `y`) needs access to the value of `x`, which is a local variable of the **outer** function `f`. While the activation record of `f` is destroyed, the variables it captures must live beyond `f`'s execution, requiring heap allocation.
- **Machine Code (Conceptual):** The machine code for the inner `lambda` might contain instructions like:

```
; Get the environment pointer (from the function object).
; Access the value of 'x' at a known offset within the activation record.
movq 24(%rbp), %rax  ; Example (offset might vary)
; ... (add %rax to y) ...
```

- A function pointer contains 2 pointers, making them fatter.

16.3 Dynamic vs. Static Chains

- **Continuation:** An IP/EP pair. The EP chain leads to caller's function activation record.
  - **Dynamic Chain:** Linked list of environment pointers, representing the **call history** (who called whom). Used for backtraces.

- **Static Chain:** Linked list representing the **definition history** (who defined whom). Used for accessing non-local variables.

```
EP -> current function activation record
|
v
EP -> caller's function activation record
|
v
EP -> caller's caller's function activation record
i.e. who called us
```

- The static chain is to find the definer of a function.

```
(Current Function)
EP -> [Definer's EP | ... ]
|
V
[Definer's Definer's EP | ... ]
|
V
...
```

- **Example:** In the currying example, when the inner lambda needs `x`, it looks up the **static** chain (to `f`'s activation record), not the dynamic chain.
- The length of a static chain is limited by how nested a function is.
- **Scheme's Activation Records:** Often on the **heap**, not the stack, because they can outlive the function's return (due to currying). The compiler can optimize to use the stack when possible.
- **Object Allocation without `new` / `malloc`:** Currying can be used to allocate objects (the activation record becomes the object). The local variables of the defining function act like the "slots" of the object. This is how object-oriented programming was originally conceived.

16.4 Heap Management

- **Last-In, Anything-Out:** Objects can be freed in any order.
- **Heap:** A large block of memory containing objects and free space.

```
Heap:
[Object] [Free] [Object] [Object] [Free] ...
^        ^        ^
|        |        |
Stack/Registers can point into the heap
```

- **Roots:** Pointers from outside the heap (stack, global registers, etc.) into the heap. The heap manager needs to know where the roots are.

16.5 Roots

- Pointers into the heap from outside the heap (e.g., stack, global variables, registers). These are the starting points for determining which objects are reachable.
- An object is **reachable** if it can be reached by following a chain of pointers starting from a root.
- Unreachable objects can be reclaimed (garbage collected).

16.6 Keeping Track of Roots

1. **Program Helps:** The program explicitly tells the heap manager when roots are created/destroyed (e.g., calls to special functions). Error-prone.
2. **Compiler Records Root Locations:** The compiler generates tables (read-only) that tell the heap manager where roots are in global variables and activation records. Used in languages like Scheme and ML.
3. **No Help (C/C++):** The programmer explicitly manages allocation (`malloc` / `new`) and deallocation (`free` / `delete`). The heap manager doesn't track roots.
4. **C/C++ Issues:**
   - **Dangling Pointers:** Pointers to freed memory. Undefined behavior even to **compare** a dangling pointer to `NULL`.

     ```
     char *buffer = malloc(1000);
     free(buffer);
     return buffer != NULL; // Undefined behavior!
     *buffer = 'a';          // Definitely undefined behavior (and likely a crash).
     ```

   - Dangling pointers can also happen in stack.

     ```
     char *p;
     {
       char c;
       p = &c;
     } // variable goes out of the scope.
     return *p == 0; // undefined behavior.
     ```

   - It is faster and buggy.
   - Not really used in other languages anymore that aren't low-level.

16.7 Garbage Collection (Managed Heap)

- No explicit `free` or `delete`. The system automatically reclaims unreachable objects.
- **Requirements for Garbage Collection:**
  1. **Know where the roots are.**
  2. **Know the layout of objects** (where the pointers are within each object).
  3. **Keep track of free space.**

16.8 Keeping Track of Free Space: The Free List

- **Goal:** Make `malloc` / `new` fast.
- **Naive Approach (Too Slow):** Keep track of all objects and subtract to find free space.
- **Free List Approach:** A linked list of free blocks. Each node in the list contains a pointer to a free block and its size.

  ```
  Free List:
  [Ptr, 100] -> [Ptr, 50] -> [Ptr, 300] -> [Ptr, 500] -> NULL

  Heap:
  [Object] [Free (100)] [Object] [Free (50)] ... [Free(300)] ... [Free(500)]
  ```

- **Problem:** Where to store the free list itself, given that it needs to be dynamically allocated? Don't want a separate "meta-heap."

- **Solution:** Store the free list **within** the free blocks themselves! (As long as free blocks are large enough to hold a pointer and a size.)

```
Heap (with free list embedded):
[Object] [100, NextPtr] [Object] [50, NextPtr] ...
           ^----------------------------|
```

16.9 Efficiency of `malloc` and `free`

16.9.1 `malloc` and `free` Performance

- **Goal:** Make `malloc` and `free` as fast as possible (ideally O(1)).
- **`malloc` (Naive First-Fit):**
  - Walk the free list, looking for the **first** block that's large enough.
  - If found, split the block (if necessary) and return a pointer to the allocated portion.
  - **Worst Case:** O(length of free list). Can be very slow if the free list is long and fragmented.
- **`free` (Naive):**
  - Add the freed block to the free list.
  - **Problem:** Need to **coalesce** adjacent free blocks to prevent fragmentation.
  - **Naive Coalescing:** Scan the entire free list O(length of free list). Slow!
- **Fragmentation:**
  - **External Fragmentation:** Enough total free space, but it's scattered in small, non-contiguous blocks. Can lead to allocation failures even when there's enough total free space.
  - **Internal Fragmentation:** An allocated block is larger than the requested size (due to alignment requirements, minimum block sizes, or metadata overhead). The extra space within the block is wasted.

16.9.2 Improving `malloc` Performance

- **First-Fit:** Simple, but can lead to fragmentation (lots of small blocks at the beginning of the free list).
- **Best-Fit:** Find the **smallest** block that's large enough. Reduces external fragmentation, but **slower** (requires scanning the entire free list).
- **Roving Pointer (Circular Free List):** Keep a pointer to the **last** allocated block. Start the search from there (instead of always from the beginning). Can improve performance by distributing allocations more evenly.

16.9.3 Improving `free` Performance (and reducing fragmentation)

- The naive approach is also O(length of free list).
- **Boundary Tags:** Store metadata (size, free/allocated status) at **both** the beginning and end of **each** block (both free and allocated).

```
[Size, Free, ...Payload..., Size, Used]
```

- This allows for O(1) coalescing:
  1. When freeing a block, look at the boundary tags of the **adjacent** blocks (both before and after).
  2. If an adjacent block is free, merge them into a single, larger free block.
- **Tradeoff:** Increases space overhead (extra metadata for each block) but significantly improves `free` performance.

16.9.4 `malloc` Failure and External Fragmentation

- If `malloc` cannot find a large enough free block, it typically returns a null pointer (in C).
- This can happen even if there's **enough total** free space, but it's fragmented into small, non-contiguous blocks (external fragmentation).

16.10 Mark and Sweep Garbage Collection

- **Assumptions:**
  - System knows where the roots are.
  - System knows the layout of objects.
  - Each object has a **mark bit** (initially 0).
- **Phases:**
  1. **Mark:**
     - Start from the roots.
     - Recursively traverse all reachable objects, setting the mark bit of each object visited to 1.
  2. **Sweep:**
     - Scan the entire heap.
     - Add all **unmarked** objects to the free list (they are unreachable).
     - Coalesce adjacent free blocks.
  3. **Unmark:**
     - clear mark bit.
     - Reset the mark bit of all **marked** objects to 0 (prepare for the next garbage collection cycle).
- **Mark Bit Storage:** Can be in the object itself or in a separate "mark bit region" (for better caching).

16.11 Problems with Mark and Sweep

- **Cost:** Proportional to the number of **reachable** objects (Mark phase) and the **total** number of objects (Sweep phase). Can be very expensive for large heaps.
- **Real-Time Issues:** Traditional mark-and-sweep pauses the entire program during garbage collection which is unacceptable for real-time systems (e.g., robots).

16.11.1 Real-Time Garbage Collection

- Goal: Bound the **maximum pause time** caused by garbage collection.
- **Incremental Collection:** Interleave garbage collection work with program execution. Do a little bit of marking/sweeping at a time.
- **Complexity:** Requires careful synchronization to avoid race conditions between the garbage collector and the program (which might be modifying the object graph).
- **Overhead:** Real-time garbage collectors are typically less efficient overall than traditional stop-the-world collectors.

16.12 Dangling Pointers and Memory Leaks

- Garbage collection **eliminates** dangling pointers (because objects are only freed when they are unreachable).
- Garbage collection **reduces** memory leaks (unreachable objects are eventually reclaimed). However, unintentional references can still prevent objects from being collected.

16.13 Conservative Garbage Collection (for C/C++)

- A technique that allows garbage collection to be used in languages like C and C++, **without** requiring compiler support or precise knowledge of object layouts.
- **Basic Idea:**
  1. Treat **every** word in memory (stack, registers, global data) that **looks like** a pointer to the heap as a potential root.
  2. Perform mark-and-sweep based on these "potential roots."
- **Advantages:**
  - Can be added to existing C/C++ programs with minimal code changes (often just by redefining `free` to do nothing).
  - Eliminates dangling pointers.
- **Disadvantages:**
  - **Conservative:** May keep some unreachable objects alive (because a random bit pattern might happen to look like a pointer). This leads to memory leaks (but **not** dangling pointers).
  - **Performance:** Scanning the entire stack and registers can be expensive.
  - **No Compaction:** Cannot move objects in memory (because it doesn't know for sure which words are pointers).
- **Example:**

```c
#define free(p) ((void) 0)  // Disable freeing.

// ... rest of the C program ...

// The garbage collector will scan the stack, registers, and global data,
// treating any value that looks like a heap address as a root.
```

It looks at the stack and registers. If any number in the stack or registers has the prefix of 0xc (in hex), it assumes this could be a pointer to heap. This way might be wrong but safe.

16.14 Reference Counting (Python's Original Approach)

- An alternative garbage collection technique.
- **Basic Idea:**
  - Each object has a **reference count**: the number of pointers to that object.
  - When a pointer is created, increment the reference count.
  - When a pointer is destroyed, decrement the reference count.
  - When the reference count reaches 0, the object is unreachable and can be immediately freed.
- **Advantages:**
  - Simple to implement.
  - **Immediate** reclamation of unreachable objects (no long pauses).
  - Incremental (work is distributed over time).
- **Disadvantages:**
  - **Overhead:** Requires extra space for the reference count, and every pointer assignment requires updating reference counts.

- **Cycles:** Cannot collect cyclic data structures (where objects point to each other in a cycle). This is a major limitation.

- **Python's Use of Reference Counting:** Python uses reference counting as its **primary** garbage collection mechanism, but it also has a **cycle detector** (a form of mark-and-sweep) to handle cyclic garbage.

```
p = q  # p and q reference the same object, refcount of object increase by 1
r = q   # r now also references the same object, refcount increase by 1
p[1] = p  # p creates a reference to itself (assuming p is a list or similar mutable container), ref count
↪  increase by 1
p = q = r = None  # All external references are removed, but internal p[1] = p still exists.
# reference count drops to 1, but not 0.
# This object would leak without cycle detection.
```

16.15 Generational Garbage Collection (Modern Java)

- **Goal:** Make allocation very fast (as close as possible to stack allocation in C).

Basic Idea: An optimization based on the observation that **most objects have short lifetimes**.

- **Generations:** Partition the heap into generations (nursery, young, middle-aged, old).
- **Allocation:** Always allocate in the nursery.
- Garbage collect the young generation more frequently than the old generation.
- Objects that survive a garbage collection in the young generation are promoted to the older generation.
- **Heap Pointer (HP) and Limit Pointer (LP):** Allocation is just incrementing HP.

```
//Conceptual code
void *malloc(size_t n) {
  void *p = hp;
  hp += n;
  if (limit_ptr < hp){
    ouch(); // do something when run out of the memory
  }
  return p;
}
```

- **Ouch:** When HP reaches LP (nursery is full), either:
  - Allocate a new nursery and copy old to new.
  - Call the garbage collector (usually a minor collection, just the nursery).
- **Key Observation:** Most objects point to **older** objects (especially in functional programming).
- **Benefit:** Can garbage collect just the nursery (or a few young generations) without looking at the whole heap.

16.16 Copying Collectors (Java)

- **Idea:**
  - Instead of collecting in place, **copy** reachable objects to a new region.
  - Divide the heap (or a generation) into two spaces: "from-space" and "to-space."
  - Allocation happens in the "from-space."
  - When the "from-space" is full, the garbage collector:
    1. Start with roots pointing into the old nursery.
    2. Copy reachable objects to the new nursery.

3. Update roots and pointers within the copied objects to point to the new locations.
4. Discard the old nursery.

- **Advantages:**
- **Compaction:** Automatically compacts memory (no external fragmentation).
- **Fast Allocation:** Allocation is simply pointer increment (like stack allocation).
  - **Better Caching:** Live objects are packed together, improving cache locality. Free space is never traversed (doesn't pollute the cache).
  - **Cost is proportional to live objects only,** not the total number of objects (including garbage).
  - Avoids the issue of the sweep phase.
- **Disadvantages:**
  - Copying overhead.
  - Need to update roots and pointers.
- **Finalize Problem:** Java's `finalize` method (called before an object is reclaimed) doesn't work well with copying collectors, because the garbage objects are never examined. Java has deprecated `finalize`.

16.17 Multi-threading and Garbage Collection (Java)

- In a multi-threaded environment, garbage collection becomes more complex.
- **Problem:** If multiple threads are allocating objects concurrently, you can't simply have a single global heap pointer.
- **Solution (Java):** Give each thread its own **thread-local allocation buffer (TLAB)** within the nursery. Each thread can allocate from its own TLAB without synchronization. When a TLAB is full, the thread gets a new TLAB (which might trigger garbage collection).

16.18 Private Free Lists

- **Idea:** For frequently allocated objects (e.g., cons cells in Lisp), maintain a private free list.
- **Example (Lisp/Scheme `cons` cells):**

```
struct cons *cons_free_list = NULL; // Global free list for cons cells

struct cons *allocate_cons() {
  if (cons_free_list != NULL) {
    void *p = cons_free_list;
    cons_free_list = cons_free_list->next; // Or whatever the 'next' field is
    return p;
  } else {
    return malloc(sizeof(struct cons)); // Fallback to system allocator
  }
}

void free_cons(struct cons *p){
  p->next = cons_free_list;
  cons_free_list = p;
}
```

- **Advantages:**
  - Can be **much** faster than the general-purpose `malloc` and `free` (because it avoids the overhead of searching a free list, coalescing, etc.).

- Improves locality of reference (newly allocated objects are likely to be close in memory to recently freed objects).

- **Disadvantages:**
  - Only works for specific object types.
  - **Can be a performance problem with copying garbage collectors** (the garbage collector will waste time traversing the private free list, which contains only garbage).
  - This technique works well with C/C++, but badly with Java, because java uses copying garbage collector.

17 *<2025-03-06 Thu>* Names, Bindings, Scope, and Error Handling

17.1 Names (Identifiers)

17.1.1 Philosophical Principle Example

- Sir Walter Scott: the author of **Waverly**.
- The King did not know Walter Scott was the author of **Waverly**.
- **Substitution Principle:** We should be able to substitute a name with its value.
  - The King did not know the author of **Waverly** was the author of **Waverly**.
- **Problem:** The substitution leads to a nonsensical statement.
- **Reason:** Names can have multiple meanings. "Sir Walter Scott" is not **just** "the author of Waverly."
- **Programming Language Analogy:** A variable name can refer to value, type, address, etc.

17.1.2 Names and Bindings in C

```
long int i = 27;
return i + 3;      // Substitutes to: return 27 + 3;  (OK)
return sizeof i;   // Substitutes to: return sizeof 27; (NOT OK - different meaning)
return &i + 1;    // Substitutes to: return &27 + 1;  (NOT OK - invalid C)
```

- **Problem:** Substituting the **value** (27) for the **name** (`i`) doesn't always preserve meaning.
- **Reason:** The name `i` is bound to:
  - **Name:** `i`
  - **Value:** 27 (initially)
  - **Type:** `long int`
  - **Address:** The memory location of `i` (`&i`)
  - **Alignment:** Memory alignment requirements (`alignof(i)`)
- **Key Idea:** Every programming language has things that names are bound to.

17.1.3 Binding

- **Binding:** An association between a name and a "value" (property).
  - Example: `int i = 19;` binds `i` to: type (`int`), value (19), alignment (e.g., 4), address (e.g., `0x79c00040`).

17.1.4 Sets of Bindings (Namespaces)

- A set of bindings is a mapping: names → "values". Also called a **namespace** or **dictionary**.
- Example (Conceptual):

```
{
  x: (int, 1, address_of_x),
  y: (double, 3.0, address_of_y),
  z: (float, ...)
}
```

- **How Bindings are Determined:**
  - **Explicitly:** Programmer directly specifies the binding (e.g., `int i;`).
  - **Implicitly:** Compiler/interpreter infers the binding (e.g., type inference in ML, Fortran's I-N rule).
  - Example: in Fortran, variable start with I-N is implicitly integers, and others are floating point numbers
  - Python: all done dynamically

17.1.5 Binding Time

- **Binding Time:** The time at which a binding is created.
- Different bindings for the same name can have **different** binding times.
- Example (C):

```c
long int i = 27;
i++;
```

  - `i` to value 27: **Execution time** (assignment).
  - `i` to type `long int`: **Compile time** (static typing).
  - `i` to address `&i`: **Block entry time** (local variable) or **link/load time** (global/static).
  - `INTMAX` to $2^{31} - 1$ (on a typical 32-bit system): **Platform definition time**.
  - `static int j;` (address): **Link time** (traditionally), but can be delayed to **load time** with ASLR + PIE.

17.1.6 Separate Compilation

- Compiling modules separately.
- The compiler doesn't know all bindings at compile time (especially across modules).

17.2 Declarations vs. Definitions

- **Declaration:** Provides **partial** information about a name (e.g., type).
- **Definition:** Provides **complete** information.
- Example (C):

```c
double difftime(time_t a, time_t b); // Declaration (prototype)
double difftime(time_t a, time_t b) { // Definition
  return a - b;
}
```

- Declarations and definitions can have different context.
- Declarations provide **abstraction**.

17.3 Namespaces

- **Namespace:** A set of bindings.
- **Nested/Block-Structured Declarations:** Inner declarations **shadow** outer ones.

```
int x, y, z;
int f() {
  int y, z, w;  // These y and z shadow the outer ones.
  return y + w + x; // y, w are local, x is from outer scope.
}
int g() { return x; } // x refers to the global x.
```

- Computing namespaces: start with the innermost namespace and work outwards.

17.3.1 Scope (of a Name)

- **Scope:** The set of locations in a program where a name is **visible**.
- **Visibility:** A name is **visible** if it is written in the program and has the desired meaning (binding).
- Scope and visibility are closely related.

17.3.2 Primitive Namespaces (Built-in)

- Languages often have multiple, built-in namespaces.
- Example (C/C++):

```
int f() {
  struct f { float f; } f; // struct tag 'f', member 'f', variable 'f'
  enum f { h, i } g;     // enum tag 'f', enumerators 'h', 'i', variable 'g'
#include <f>            // File 'f' (from include path)
#define f g             // Macro 'f'
  f: goto f;            // Label 'f' (used with goto)
  g: goto g;
}
```

- C/C++ Namespaces:
  1. Ordinary Identifiers: Variables, function names.
  2. Struct Tags: After `struct` keyword.
  3. Enum Tags: After `enum` keyword.
  4. Member Names: Within a `struct` / `union` (accessed with `.` or `->`).
  5. File Names: In `#include`.
  6. Preprocessor Macros: `#define`.
  7. Labels: Used with `goto`.

17.4 Information Hiding (Modularity)

- **Goal:** Control visibility of names from outside a module/class/package.
- **Mechanisms:**
  1. **Declaration of Visibility:**
     - C: `static` (private to module), `extern` (visible to all).
     - Java: `public`, `protected`, `private`, default (package-private).
  2. **Explicit Namespace Operators:**

- C++: Scope resolution operator (`::`).
- Java: Dot operator (`.`).

3. **Signatures/Interfaces:** Define the public interface, hiding implementation (Ocaml is powerful in this).

Example: Java

```
//Assume those are defined in different files.
public class A {
  public int x;
  protected int y;
  int z; //default, package private
  private int w;
}

//In other file
import A; // Assume A is in the same package.

class B {

}
```

- `public`: accessible everywhere.
- `protected`: accessible within the same package, and by subclasses (even in different packages).
- `default` (no modifier): accessible only within the same package.
- `private`: accessible only within the same class.

17.5 Explicit namespace operators

- you edit the full namespace to make it a more abstract namespace
- in Ocaml, operators at compile time.
- Example, Ocaml:

```
module q = struct
  type 'u queue =
  | Empty
  | Node of int * 'a * 'a queue

  let enqueue = (* actual implementation *)
  ...
end
```

- Example, Ocaml, signature:

```
module type qi =
  sig
    type 'a queue
    enqueue: 'a * 'a queue -> b queue
  end
```

  - All we expose to the outside is signature, and modules can be combined to be functors.

17.6 Ways to Address Errors/Faults/Failures (Bugs)

1. **Compile-Time Checking (Static Checking):** Compiler detects errors.
   - Type checking, etc.
2. **Preconditions:** Specify conditions that **must** be true before a function call.
   - Example (Eiffel): `require n >= 0;`
3. **Total Definitions:** Design functions to be defined for all inputs.
   - e.g. Rust doesn't have exception.
4. **Fatal Exits (abort):** Terminate the program immediately.
5. **Exceptions:** Mechanism for handling errors at runtime.
   - `try-catch` blocks.
   - **Problem:** Can make code complex; easy to forget to handle exceptions.
   - Example, exception in python

```python
try:
  do_some_stuff() # heart of the algorithm
except (IOError e):
  fixup(e)
except (NumError e):
  fixup(e)
finally:
  print("done")
```

   - The actual code is probably 25%, and error handling takes 75%
6. **Checked Exceptions (Java):** Compiler **forces** handling of certain exceptions.
   - Improves reliability.
   - Can be cumbersome.
   - Java has `Throwable`, an object.
   - throw and catch is dynamic, not static.

18 *<2025-03-11 Tue>* Parameter Passing, Object Oriented Programming, Cost Models

18.1 Parameter Passing

18.1.1 Semantics vs. Efficiency

- Two main aspects of parameter passing:
  - Semantics: How the arguments and parameters correspond.
  - Efficiency: How quickly and with what resources parameters are passed at runtime.
- Efficiency is a primary motivator for different parameter passing conventions, as function calls are frequent.

18.1.2 Correspondence Models Call

- Different ways arguments in a function call can correspond to parameters in the function definition:
  - Positional: Arguments are matched to parameters based on their position in the list.
    - Example: `f(a, b+1)` with definition `def f(x, y)`. `a` corresponds to `x`, and `b+1` corresponds to `y`.
  - Varargs (Varying numbers of arguments): Allows a function to accept a variable number of arguments.
    - Example (Python-like): `f(a, b-1, c+5)` with definition `def f(x, *y)`. `a` corresponds to `x`, and `b-1`, `c+5` are packed into a tuple and correspond to `y`.
  - Keyword: Arguments are passed with keywords, allowing order-independent matching to parameters by name.
    - Example (Python-like): `f(y=5, x=10)` with definition `def f(x, y)`. `x` is matched with `10`, and `y` with `5` regardless of position.

18.1.3 Runtime Methods for Parameter Passing

1. Call by Value

   - Caller evaluates the argument expression to get a value.
   - A **copy** of this value is passed to the callee.
   - Modifications to parameters within the callee do not affect the original arguments in the caller.
   - Typically used for small values (<= 64 bits) for efficiency.
   - Advantages:
     - Simple to understand and implement.
     - Fast for small arguments due to efficient copying.
   - Problem: Inefficient for large arguments as copying large data structures (e.g., arrays) can be expensive.
     - Example: Passing a whole array by value, not just a pointer.

2. Call by Reference

   - Caller evaluates the **address** of the argument (or a reference to where the object is stored).
   - This address (or reference) is passed to the callee.
   - The callee operates directly on the original argument's memory location.
   - Modifications to parameters in the callee **do** affect the original arguments in the caller.
   - Advantage:
     - Efficient for large arguments as only the address (which is small) is passed, avoiding expensive copying.
   - Disadvantages:
     - Slower for smaller arguments as passing an address (e.g., 8 bytes) might be less efficient than passing the value directly (e.g., 4 bytes).
     - **Aliasing**: The bigger problem. Multiple names (in caller and callee) refer to the same memory location, making code harder to understand and reason about.
       - Compiler needs to be more careful and may generate slower code to handle potential aliasing.
   - Source Code Example (Conceptual C-like):

```
// Source code:
int a[1000];
f(a, b+7); // Call by reference conceptually

int f(int x[1000], int y) { // x is treated as reference
```

```
  x[10] = 3; // Modifies the original array 'a'
  return x[5] + y;
}

// Machine code (actually C-like to illustrate the concept):
int a[1000];
f(&a, &(int)(b+7)); // Pass addresses
int f(int(*x)[1000], int *y) { // x and y are pointers
  (**x)[10] = 3; // Dereference x to access array 'a'
  return (**x)[5] + *y; // Dereference x and y
}
```

  - In machine code, addresses are passed. Even for expressions like `b+7`, a temporary storage location is created to hold the value, and its address is passed.
  - Object-oriented languages often use references under the hood for objects.

- Aliasing Example: Undefined Behavior

```
int f(int &x, int &y) { // Call by reference in C++
  print(x);
  y = x++; // Undefined behavior if x and y are aliases for the same variable
  &x = &y; // Invalid operation - cannot change address of x, also undefined behaviour
  print(x);
}

int i = 12;
f(i, i); // Calling f with the same variable 'i' as both arguments
```

  - Issue: When `f(i, i)` is called, `x` and `y` become aliases for the same variable `i`.
  - `y = x++;` leads to undefined behavior because modifying `x` also immediately affects `y` (as they are the same memory location). The order of operations becomes ambiguous and compiler optimizations can lead to unexpected results.
  - Machine code might initially cache `x`'s value in a register, but since `x` and `y` are aliases, this caching becomes incorrect when `y` is modified.
  - Compiler cannot easily optimize call-by-reference code due to potential aliasing, leading to slower execution.
  - Call by value avoids aliasing issues because the callee works with a copy.

3. Call by Result

- Opposite of call by value in terms of data flow for result.
- Callee is given a storage location (parameter) to write the result of the function.
- Callee starts with an uninitialized parameter.
- During execution, the callee sets the value of the parameter.
- Just before the function returns, the value of the parameter is copied back to the caller.
- Used when a function needs to "return multiple values" by modifying parameters.
- Example: `read` function in Linux API.

```
int read(int fd, char *buf, int bufsize);
```

  - `buf` is conceptually a "call by result" parameter.
  - Initial content of `buf` doesn't matter. `read` fills it with bytes read from file descriptor `fd`.
  - `read` effectively returns two things: the number of bytes read (function return value) and the bytes themselves (in `buf`).
  - Used for efficiency in system calls where parameters are hardwired for performance.

4. Call by Value Result

- Combination of call by value and call by result.
- Caller evaluates and copies the argument value to the callee (like call by value).
- Callee operates on the copied value.
- Just before returning, the **modified** value of the parameter in the callee is copied back to the caller (like call by result).
- Involves **two** copies: one in, one out.
- Advantage: No aliasing problems, as callee works on a copy.
- Disadvantage: Overhead of two copies, especially for large arguments.

5. Call by Name

- Caller provides the callee with a way to evaluate the argument expression **whenever** the parameter's value is needed.
- This "way to evaluate" is often implemented as a **thunk** - a parameterless procedure (function pointer).
- Callee does not receive the value directly, but a function to get the value.
- Resembles call by reference in that no initial evaluation happens, but instead of an address, a function is passed.
- Example (Conceptual Scheme-like):

```
(f (lambda () a) (lambda () (+ b 7))) ; Call with thunks for a and b+7

(define (f x y) ; x and y are conceptually thunks
  (vector-set! (x) 10 3)  ; Evaluate thunk x to get array, then set index 10
  (+ (vector-ref (x) 5) (y))) ; Evaluate thunk x and y when needed
)
```

  - `x` and `y` inside `f` are treated as functions. Calling `x()` or `y()` evaluates the original expressions `a` and `b+7` respectively.
- Disadvantage: Slower due to repeated evaluation of thunks whenever the parameter is used.
- Advantage: Can improve reliability in certain cases and avoid unnecessary computations.
  - Example: Prevents crashes in cases where an argument's value is not always needed.
- Example: Reliability Benefit - Avoiding Division by Zero

```
printarg(int nitems, int avgitemval) { // Conceptual call by name for avgitemval
  if (nitems == 0) {
    print("no items");
  } else {
    print("average is", avgitemval); // avgitemval is only evaluated if nitems != 0
  }
}

// Caller
for (int i = 0; i < n; i++)
  sum += a[i];
printarg(n, sum/n); // sum/n argument passed by name conceptually
```

  - In call by value, `sum/n` is evaluated **before** calling `printarg`, leading to a crash if `n` is 0.
  - In call by name, `sum/n` is only evaluated **inside** `printarg`, and **only if** `nitems` (which is `n`) is not 0, thus avoiding the division by zero error.
  - Shows reliability by preventing crashes when arguments might cause errors if evaluated unconditionally.
- Eager Evaluation vs. Lazy Evaluation
  - Eager Evaluation: Evaluate arguments **before** function call (call by value, reference, result).

- Lazy Evaluation: Evaluate arguments **only when needed** inside the function (call by name).

6. Call by Need

   - Optimization of call by name.
   - Evaluate the thunk at most **once**.
   - Cache the result of the thunk's evaluation.
   - Subsequent uses of the parameter use the cached value, avoiding repeated evaluations.
   - Combines the reliability benefits of call by name with improved efficiency.
   - Used in lazy programming languages like Haskell.
   - Advantage: Allows for infinite data structures.
     - Example (Haskell):

       ```
       let pl = "list of all primes"  -- pl is a computation for prime numbers (not evaluated yet)
       print pl[7] -- Only computes primes up to the 7th prime and then stops
       ```

       * `pl` is not computed until its value is actually needed (when printing `pl[7]`). Only then is computation performed just enough to get the 7th prime.

7. Call by Unification

   - Used in Prolog.
   - More complex than other methods, related to call by value result but with added features.
   - Involves pattern matching and variable binding.
   - Can leave variables unbound for later instantiation.

8. Static Calling Methods - Macro Calls

   - Fundamentally different from runtime methods; operate at compile time (statically).
   - Macro arguments are treated as bits of program code (textual substitution).
   - Macro expansion happens during compilation.
   - Resembles textual substitution or code copying.
   - Look like function calls in source code but operate completely differently.
   - Distinction between call by value, reference etc., is not relevant in macro calls as they are compile-time operations.

18.2 Extra Hidden or Transformed Parameters

- Many programming language features involve passing hidden parameters that are not explicitly written in the source code.
- For OO Methods: Pointer to self (`this` or `self`) is implicitly passed to methods, allowing access to the object's state.
- For Static Chain: Pointer to definer's frame is passed to nested functions to access non-local variables in their defining scope.
- For Dynamic Chain: Pointer to the caller's frame (e.g., `%rbp` in x86-64) is maintained for stack management and returning to the caller.
- For Return Address: Return address is implicitly passed (e.g., top of stack in x86-64) to functions to know where to return after execution.
- For Address of Thread Local Storage (TLS): Pointer to thread-local storage is passed for efficient access to thread-specific variables.

18.3 Cost Models

18.3.1 What is a Cost Model?

- A mental model of computing resources needed for efficient execution of a program.
- Helps in understanding and optimizing performance.
- Includes:
  - Mental model of resource usage.
  - Big O notation for asymptotic efficiency: Focuses on how runtime scales with input size (e.g., $O(N)$, $O(NlogN)$).
    - $O(N)$ is asymptotically better than $O(NlogN)$.
  - Absolute costs (constant factors): Big O ignores constant factors, but in practice, absolute performance can matter significantly, especially for fixed problem sizes.

18.3.2 What are the Costs?

1. Real Time: Actual wall clock time to get an answer back.
2. Memory: Amount and type of memory used (registers, cache levels, RAM, secondary storage).
3. System Time: Time spent in operating system overhead (kernel operations).
4. CPU Time: Time spent actually executing program instructions on the CPU (excludes I/O wait).
5. Energy/Power: Battery drainage, power consumption.
6. I/O Access Time and Space: Time and storage related to input/output operations.
7. Network Access: Time and bandwidth used for network communication.
8. Economic Cost: Ultimately, all costs can translate to economic cost (money, resources).

18.3.3 Classic Model: Lisp Lists

- Assumes RAM (Random Access Memory): Access any memory location in constant time.
- Lisp List Representation (Singly Linked List):
  - Each list element is a node in memory.
  - Nodes are linked via pointers, potentially scattered in memory.
- Cost of `append` in Lisp:

```
(append '(foo) x) ; O(1) - Constant time, prepending to a list
(append x '(foo)) ; O(|x|) - Linear time in length of x, needs to copy x
```

  - `(append '(foo) x)` is $O(1)$ because it only creates a new node and points its tail to `x`.
  - `(append x '(foo))` is $O(|x|)$ because it needs to copy the entire list `x` to create a new list with '(foo) appended.

18.3.4 Python Lists

- Python lists are objects that hold a pointer to an underlying array.
- Python list object is a **description** of the array, containing:
  - Pointer to the actual array in memory.
  - Number of elements currently in the list.
  - Allocated size of the array (capacity).
- `ls.append(39)`:
  - Typically $O(1)$ - Just adds element to the next available slot in the array.
  - Occasionally $O(|ls|)$ - If the array is full, a new array is allocated (often doubled in size), elements are copied, and then the new element is appended.
- Amortized Cost of `append`: $O(1)$
  - While occasional appends are $O(|ls|)$, most are $O(1)$, averaging out to $O(1)$ over many appends.
  - Total cost of T appends is not $O(|ls| * T)$ but closer to $O(T)$ in the long run.

18.3.5 Prolog Unification Cost

- Unifying two Prolog terms (trees) X and Y: `X = Y`.
- Cost is $O(min(|X|, |Y|))$ - Proportional to the size of the smaller term.
- Unification process walks through both trees simultaneously and stops when it reaches the end of the smaller tree or finds a mismatch.

18.3.6 Array Access Costs

- Example: `double a[1000000];`
- Goal: Calculate `a[i]` fast.
- Assume: Address of `a` in `%rax`, index `i` in `%rdi`.
- Naive Calculation (Multiply and Add):

```
movq %rdi, %rbx  ; Copy i to %rbx
mulq %rbx, 8     ; Multiply i by 8 (sizeof(double))
addq %rbx, %rax  ; Add result to base address of 'a'
```

  - Integer multiplication is slow.
- Optimized Calculation (Arithmetic Shift and Add):

```
movq %rdi, %rbx  ; Copy i to %rbx
ashq 3, %rax     ; Arithmetic shift left by 3 bits (equivalent to multiply by 2^3 = 8)
addq %rbx, %rax  ; Add shifted index to base address of 'a'
```

  - Arithmetic shift is much faster than multiplication.
  - TLDR: Pick array element sizes that are powers of 2 to use shifts instead of multiplications for index calculations.
- Example: 2D array `double b[100000][16];`
  - Address of `&b[i][j] = b + i * 100000 * 16 + j * 16`

  - If inner dimension (16) is power of 2, inner index calculation (j*16) can be a shift.

- Complications:
  - Reliability: Subscript Checking

```
cmpq %rdi, 0    ; Compare index with 0
jl error        ; Jump if less than 0 (error - out of bounds)
cmpq %rdi, $16  ; Compare index with 16 (array bound)
jge error       ; Jump if greater or equal to 16 (error - out of bounds)
```

    * Subscript checking adds overhead (comparisons and branches).
    * Optimization: Unsigned comparison for single branch checking:

```
cmpq %rdi, $16
jge error ; Jump if above or equal (unsigned comparison) - checks both < 0 and >= 16 in one
↪  go if using unsigned index
```

  - Cache Effects:
    * RAM is cached in cache lines (e.g., 64 bytes).
    * Cache hashing can cause performance issues if memory access patterns are not cache-friendly.
    * Example: Accessing elements in the same column of a large array where row size is a power of 2 can lead to cache line thrashing if addresses hash to the same cache set.
    * Solution: For large arrays, consider making row sizes not powers of 2 (e.g., odd or prime numbers) to improve cache utilization and avoid cache line conflicts when accessing columns.
    * Trade-off: Power of 2 sizes optimize index calculations; non-power of 2 sizes can optimize cache behavior in certain access patterns.

19 *<2025-03-11 Tue>* Rust, Static Checking, Semantics, History of Programming Languages

19.1 Rust Programming Language

19.1.1 Ubuntu 25.10 and Rust Adoption

- Ubuntu 25.10 will rewrite core utilities like `cat` in Rust as an experiment.
- Rust is also being integrated into the Linux kernel.
- Motivation: Memory safety and performance.

19.1.2 Memory Safety in Rust

- Rust aims for memory safety, similar to OCaml, Scheme, and Python, but with **static** checking.
- Other languages achieve memory safety **dynamically** (runtime checks).
- Static checking in Rust happens at compile time.
- Rust's static memory safety is not absolute but covers most common cases.
- Dynamic checks are still present in languages like OCaml, though OCaml also performs static checks.
- Rust uses static checking to prevent errors that can lead to program crashes and debugging difficulties.

19.1.3 Aside: Static Checking of Encrypted Based Programming

- Motivation for static checking from encrypted-based programming.
- Goal: Run a program on a third-party machine (e.g., Amazon AWS) without revealing the computation details or data.
- Encryption approach:
  - Original program: `y = f(x)`
  - Encrypted version for Amazon: `E(y) = fE(E(x))`
  - `fE` is an encrypted program that operates on encrypted inputs and produces encrypted outputs.
- Challenges of debugging encrypted programs:
  - Inspecting running encrypted code reveals nothing useful.
  - Debugging becomes impractical or impossible due to encryption overhead and obscurity.
  - Dynamic analysis is not effective.
- Use cases: Blockchain and multi-party computation where trust in the execution environment is limited.
- Solution: Static checking of the **original** program **before** encryption.
  - Compiler and static analysis tools need to be reliable and statically verifiable.
  - Debugging in production is impossible after encryption, making static checking crucial.
- Static checking is increasingly important in:
  - Crypto programming.
  - Quantum programming: Debugging quantum programs by inspecting state destroys the state itself.

19.1.4 Rust: "Zero Cost Abstraction" for Memory Safety

- Rust's primary goal: Memory safety without runtime performance overhead.
- Innovation: Ownership and borrowing system.
- Ownership and Borrowing:
  - Statically track which reference **owns** an object and which references are **borrowing** it.
  - Static analysis ensures memory safety at compile time.
  - Prevents mutation of shared state to avoid data races.
- Benefits:
  - Memory safety without garbage collection overhead.
  - Efficient multi-threaded programming by preventing data races.
- Addressing Race Conditions:
  - Enforces rules at compile time:
    - Only one **mutable** reference to an object, OR
    - Multiple **immutable** references to an object.
  - Prevents race conditions in multi-threaded programs.
- Drop Method and Memory Management:

  - When a reference variable goes out of scope:
    * If it **owns** the object, the `drop` method is invoked for cleanup.
    * `drop` recursively cleans up owned objects (transitive closure).

- "Immutable" in Rust:
  - Similar caveats as `const` in C.
  - `const` in C means "cannot modify via **this** pointer," but the object itself might not be truly constant.
  - Immutable in Rust relates to shared access and preventing **current** modification, not necessarily permanent immutability.

- Data Races and Mutability:
  - Data races occur in C/C++ with **shared** and **mutable** objects.
  - Functional programming avoids data races by eliminating **mutability**.
  - Rust's approach: Eliminates the **combination** of shared and mutable state. Allows either shared immutable or mutable exclusive access.
  - Compile-time checks in Rust ensure these rules are followed.

19.1.5 True Memory Safety in Rust

- Runtime Bounds Checking: `a[i]`
  - Rust arrays track bounds and perform runtime checks to prevent out-of-bounds access.
  - `panic!` if index is out of bounds.
  - Fat Pointer: Rust array variable is not just a pointer but a "fat pointer" containing:
    * Pointer to the start of the array.
    * Upper bound (and potentially lower bound for slices).
  - Runtime subscript check: `L <= i <= U` or `0 <= i - L < U - L` or `(unsigned) (i - L) < U - L`.
  - Subscript checking adds runtime overhead.
  - Optimization: Loop subscript check: For loops with predictable index ranges, perform bounds check only once for the entire loop instead of in each iteration.

```
for (int i = 0; i < j-5; i++)
  a[i] = j;
```

    * Only need to check bounds for `0` and `j-6` before the loop, not in each iteration.

- No Pointer Arithmetic: Disallowed in safe Rust for memory safety, simplifies bounds checking.
- No `free/del`: Memory deallocation via automatic `drop` method calls when ownership ends.
- No Null Pointers:
  - Option type (`Option<T>`) is used to represent nullable values, similar to OCaml's option types.
  - Must explicitly check if an `Option` is `Some` or `None` before accessing its value.

19.1.6 Problems Attacked by Rust

- Use-after-free (dangling pointers): Prevented by ownership and borrowing.
- Double free: Prevented by ownership and borrowing.
- Buffer overrun: Subscript checking prevents out-of-bounds access.
- Data races: Ownership and borrowing rules prevent shared mutable state.
- Null pointer dereferencing: Option type and compile-time checks prevent dereferencing null pointers.

19.1.7 Rust Performance and `unsafe` Keyword

- Rust aims to be fast and competitive with C/C++.
- `unsafe` Keyword: Escape hatch to C/C++-like behavior for performance-critical sections.
  - Allows bypassing Rust's safety checks for manual memory management, pointer arithmetic, etc.
  - Goal: Minimize `unsafe` code to a small, performance-critical portion.
  - Most of the code should be in "safe" Rust.
  - `unsafe` code is where performance optimizations are needed but safety guarantees are relaxed.
  - Analogy to Linux kernel and applications: Kernel (unsafe C code) provides a safe interface (system calls) for applications (safe code).
- Strategy: Small `unsafe` portion for performance, large `safe` portion for reliability and security.

19.1.8 Rust Failure Handling

1. Panic (`panic!("ouch");`):
   - Unrecoverable error, program termination.
   - Prints backtrace for debugging.
   - Unwinds stack, calling `drop` methods along the way.
   - Exits program.
   - Configurable behavior (but default is recommended).
2. Result Type:
   - For recoverable errors.
   - Similar to a discriminant union in OCaml.

```
type ('t, 'e) result =
  | Ok of 't
  | Err of 'e
```

```
enum Result <T,E> {
  Ok(T),
  Err(E),
};
```

   - Functions return `Result` to indicate success (`Ok(value)`) or failure (`Err(error_value)`).
   - Requires explicit handling of `Result` (no automatic exception propagation like in Python/C++).
   - Enables efficient and reliable error handling.

19.1.9 Other "Goodies" in Rust

- Cargo Build System:
  - Dependency management.
  - Build automation.
  - Testing and benchmarking.
  - Documentation generation.
  - Lint checker and formatter.

19.1.10 Problems with Rust

- Subscript checking at runtime (performance overhead).
- Inertia and steep learning curve.
  - Rust compiler error messages can be challenging for newcomers.
  - Requires understanding of ownership, borrowing, and lifetimes.
  - Static checking in general can increase development complexity compared to dynamic languages.

19.2 Semantics

19.2.1 What Does a Program Mean?

- Semantics: The meaning of a program.
- Syntax vs. Semantics:
  - Syntax: Structure of the program (grammar).
  - Semantics: Meaning and behavior of the program.
- Dynamic Semantics: Program behavior during runtime execution.
  - Harder to specify formally.
- Static Semantics: Program properties that can be determined **before** runtime (compile time).
  - Traditionally easier to specify and analyze.

19.2.2 Attribute Grammars

- A method for defining static semantics.
- Used for static analysis like type checking and scope resolution.
- Components:
  - Parse tree: Syntactic structure of the program.
  - Attributes: Annotations attached to non-terminals in the parse tree to store semantic information (e.g., type, scope).
  - Attribute assignment rules: Define how to compute attributes based on grammar rules.
- Types and Scope as Attributes:
  - Example attributes: `type` and `scope`.
  - Attribute computation rules are defined for each grammar production.
- Synthesized Attributes: Information flows **upwards** from children to parent nodes in the parse tree.
  - Example: Type of an expression is synthesized from the types of its subexpressions.
  - Example rule: `float + int = float`.
  - Example: `Expr1 = expr2 + expr3`.

    ```
    type(expr1) = if type(expr2) = int & type(expr3) = int then int else float
    ```
- Inherited Attributes: Information flows **downwards** (or sideways) from parent to children.
  - Example: Scope information (symbol table) is inherited by child nodes from parent nodes.
  - Symbol Table: Maps variable names to their types and scope information.
  - Example: Symbol table for scope: `{'a': int, 'b': float}`.
  - Symbol table is computed and passed down the tree.

19.3 Dynamic Semantics

19.3.1 Basic Ideas and Disciplined Approaches

- Dynamic Semantics: Formal methods to describe the runtime behavior of programs.
- Three main approaches:
  1. Operational Semantics
  2. Axiomatic Semantics
  3. Denotational Semantics

1. Operational Semantics
   - Define the meaning of a program by describing how an **interpreter** would execute it.
   - "Read source code to an 'interpreter' for L."
   - Interpreter `I` for language `L` is written in a language `K` that is well-understood.
   - Observe program `P` (written in `L`) behavior by running it under interpreter `I`.
   - "Better: read source": Understand the interpreter code to deduce the language semantics without actually running programs.
   - Example: Operational Semantics for a subset of ML in Prolog.
     - Predicate `m(E, C, R)`: Meaning of ML expression `E` in context `C` is result `R`.
       - `E`: ML expression (represented in Prolog).
       - `C`: Context (variable-value bindings, represented as a Prolog list).
       - `R`: Result of evaluation.
     - ML Expression Representations in Prolog:
       - Integer constants: Represent themselves.
       - Atoms: Represent variables.
       - Function call: `call(F, E)` represents `F E`.
       - Function definition: `fun(I, E)` represents `fun I -> E`.
       - Let expression: `let(I, E, F)` represents `let I = E in F`.
     - Prolog Code for `m(E, C, R)`:

```
m(E, _, E):- integer(E).                    % Integer constant evaluates to itself
m(E, C, R):- atom(E), member(E=R, C).       % Variable lookup in context
m(let(I,E,F), C, R):-                       % let I = E in F
  m(E, C, V),                               % Evaluate E in context C to get value V
  m(F, [I=V|C], R).                         % Evaluate F in extended context [I=V|C]
m(fun(I,E), C, lambda(I,C,E)).              % Function definition: create a lambda closure
m(call(F, E), C, R):-                       % Function call F E
  m(F, C, lambda(I,C1,E1)),                 % Evaluate F to get a lambda function lambda(I,C1,E1)
  m(E, C, V),                               % Evaluate argument E to get value V
  m(call(lambda(I,C1,E1), V), R):-          % Auxiliary rule for lambda call
  m(let(I,V,E1), C1, R).                    % Apply lambda: evaluate let I = V in E1 in context
  ↪ C1
```

     - Context `C` is a list of `Var=Value` terms (e.g., `[x=12, y=13, z=fun(...)]`).
     - Auxiliary rule for lambda call:

```
m(call(lambda(I,C1,E1), ), R):-m(let(I,E,E1), C1, R).
```

     - Example of Lisp interpreter in Lisp 1.5 manual as a classic example of operational semantics.

2. Axiomatic Semantics
   - Define the meaning of a program using logical axioms and inference rules.

- "Write the logic for how to reason about a program."
- Hoare Triple notation: `{P} S {Q}`
  - `P`: Precondition (logical assertion about program state **before** statement `S`).
  - `S`: Statement to be executed.
  - `Q`: Postcondition (logical assertion about program state **after** statement `S`).
  - Meaning: "If precondition `P` is true, and then statement `S` runs and finishes, then postcondition `Q` is true afterwards."
- Example: Assignment Axiom
  - For assignment statement `V := E`:

    ```
    {Q[V/E]} V := E {Q}
    ```

    * `Q[V/E]`: Precondition is obtained by substituting all occurrences of variable `V` in postcondition `Q` with expression `E`.
  - Example usage: `{i < 0} i = i + 1; {i <= 0}`
    * Postcondition `Q`: `i <= 0`
    * Variable `V`: `i`
    * Expression `E`: `i + 1`
    * Precondition `Q[V/E]`: `(i+1) <= 0` which is equivalent to `i < 0`.
- If-Then-Else Rule of Inference:
  - For `if B then T else ET`:

    ```
        {P and B} T {Q}, {P and not B} ET {Q}
    ---------------------------
        {P} if B then T else ET {Q}
    ```

    * To prove `{P} if B then T else ET {Q}`, prove:
    (a) `{P and B} T {Q}` (if `B` is true, `T` leads to `Q`).
    (b) `{P and not B} ET {Q}` (if `B` is false, `ET` leads to `Q`).
- While Loop Rule (Partial Correctness):
  - For `while W do S`:

    ```
        {P and W} S {P}, {P and not W} => Q
    ---------------------------
        {P} while W do S {Q}
    ```

    * Invariant `P`: Condition that is true before and after each loop iteration (if the loop terminates).
    * To prove `{P} while W do S {Q}`, prove:
    (a) `{P and W} S {P}` (if `P` and `W` are true before `S`, then `P` is true after `S`).
    (b) `{P and not W} => Q` (if `P` is true and loop condition `W` is false, then `Q` is implied).
- Loop Termination: Axiomatic semantics as described above only guarantees **partial correctness** (if the program terminates, it's correct), not **total correctness** (program terminates and is correct).
  - To prove termination, need to show a **monotonically decreasing non-negative integer function** that decreases with each loop iteration.

3. Denotational Semantics

- Define the meaning of a program as a mathematical function.
- "semantics(P) function from I to O."
- Semantics of program `P` is a function that maps inputs `I` to outputs `O`.
- Example: Semantics of a C program is a function from byte sequences (standard input) to byte sequences (standard output) and exit status.
- Focus on **what** the program computes, not **how** it computes it.

19.3.2 Relationship to Programming Language Paradigms

- Imperative Languages (C, Java, Python): Natural fit with **operational semantics** (step-by-step execution).
- Logical Languages (Prolog): Natural fit with **axiomatic semantics** (logic and rules of inference).
- Functional Languages (OCaml, Haskell): Natural fit with **denotational semantics** (functions and mathematical mappings).

19.4 History of Programming Languages

19.4.1 Fortran (1957)

- First successful high-level programming language.
- Developed by IBM team led by John Backus.
- Key Features:
  - Arrays.
  - Loops.
  - Subroutines (functions).
- Revolutionized programming by enabling efficient translation of mathematical equations to machine code.
- Initially proprietary to IBM, but competitors soon developed their own Fortran compilers.

19.4.2 Algol (1960)

- Developed as a standardized alternative to Fortran.
- Aim: Fix Fortran's problems and create an open standard.
- Key Innovation: Formal syntax definition using Backus-Naur Form (BNF).
- Other features: Call by name, recursion.
- BNF for formal syntax definition was its most significant contribution.

19.4.3 Lisp (1959)

- List Processing Language, for Symbolic AI.
- Key Innovations:
  - Recursion.
  - Garbage Collection (GC).
  - S-expressions: Programs as data, homoiconicity.
- Programs and data are represented as lists, facilitating meta-programming and symbolic manipulation.

19.4.4 Cobol (1960)

- Common Business-Oriented Language.
- Developed by Grace Hopper and others.
- Designed for business applications, not scientific computing.
- Key Innovation: Records/structs (heterogeneous data structures).
- Aim for "readable language" for managers, using verbose English-like syntax (e.g., `ADD 1 TO N`).
- Readability goal for managers failed in practice for complex programs.
- Became the most popular language for business applications for decades.

19.4.5 PL/I (1964)

- Programming Language One.
- Developed by IBM as a "kitchen sink" language.
- Aim: Combine features of Fortran, Algol, and Cobol into a single, all-encompassing language.
- Overly complex and large; difficult to compile and use effectively.

19.4.6 C (1968-1972)

- Developed at Bell Labs, inspired by Algol 68 and PL/I, simplified approach.
- Key Features:
  - Machine level access (low-level control).
  - Simple and efficient.
  - Unix-connected: Designed alongside and for the Unix operating system.
- Became highly influential due to its portability and efficiency for system programming.

19.4.7 Simula 67 (1967)

- First object-oriented programming language.
- Descendant of Algol.
- Introduced concepts of classes, objects, and inheritance.

19.4.8 C++ (1976)

- Combo of C and Simula 67.
- Developed by Bjarne Stroustrup as "C with Classes," later evolving into C++.
- Extended C with object-oriented features.

19.4.9 Python and Rust (Modern Languages)

- Python: High-level, dynamically typed, interpreted language, popular in AI and scripting.
- Rust: Systems programming language focused on memory safety and performance, statically typed, compiled.
- History of programming languages shows evolution and adaptation to different needs and paradigms.
- Continuous development and innovation in programming languages driven by new challenges and goals.