Stanley Wei
Eggert - W25

CS 131: Programming Languages

1: Programming Languages

Language Design

Knuth taocp → tex

- Literate programming: instead of separate code & documentation, keep together
    - Ex: comments with functions (see: Python source modules)
- Cacm: gave example of literate programming

Approaches to programming

- Fast, efficient, elegant
    - Ex: coding languages
- Easy & fast to write
    - Ex: sh
- Meta-language

Language Design Issues

- Language wars
- Deciding between languages
    - Inertia - what people already know
    - Runtime efficiency: power, memory, time (real & CPU), utilization, network access, etc.
    - Language ecosystem: libraries, tools, etc.
    - Syntax: simplicity, readability
    - Compatibility/interoperability (between different ISAs, OSes, etc.)
    - Documentation & learnability
    - Orthogonality: using a certain language feature should not restrict other choices made
        - Ex: can't return arrays from functions in C/C++
            - To prevent expensive copying/returning of (potentially large) arrays
            - C/C++: wanted users to use pointers for handling large data structures
    - Safety
        - Prevalence of undefined behavior
    - Abstraction

4 notable types/classes of languages:

1. **Imperative** (e.g. C): Programs as an ordered sequence of statements
    a. Variables with state, modified via assignment
    b. Iteration via loops
    c. Order-dependence of statements
2. **Functional** (e.g. ML): Functions as the fundamental unit of computation
    a. Recursion
    b. Single-value, unmodifiable variables
3. **Logic** (e.g. Prolog): Programs as rules regarding logical inference
    a. Useful for certain problem domains (e.g. problem-solving)
4. **Object-oriented** (e.g. Java): Objects and methods as the main means of organization
    a. Subclass of imperative languages

Many languages: fall into 1+ categories

Application Programming Interface/API: large standardized libraries of predefined code

Functional Programming

Functional programming: A different way of thinking about/solving problems (as opposed to *imperative programming*)

- Imperative languages more common, but can still benefit from functional thinking

Imperative vs functional languages:

1. Imperative:
    a. Basic unit: command/statements S1, S2, …
    b. General syntax: S1; S2; S3; …
    c. Relies on:
        i. Variables with state, modified via assignment
        ii. Iteration via loops
        iii. Order-dependence of statements
2. Functional:
    a. Basic unit: functions F1, F2, F3,...
    b. General syntax: F1 (F2 (x), F3 (y, w), w)
    c. Relies on:
        i. Functional evaluation providing a partial order on computation

ii. Referential transparency
iii. Functional forms/*higher-order functions*: functions that take other functions as arguments, or return other functions as results
iv. Recursion

d. Give up on:
i. Assignment statements - no variables with state (functional: variables have values, but don't change)
ii. Side effects - statements don't affect machine state (not via assignment, not via I/O, etc.)

Motivation:

1. Clarity: May be easier to write & maintain
    a. Imperative: every solution is a long string of commands → may not be the most natural solution
        i. No ***referential transparency*** - referring to the same identifier/name twice may return different values, depending on when they are checked
2. Performance: Want our programs to run faster + be more easily optimizable
    a. Allow compiler to do heavy lifting

Basic properties of OCaml/ML

- Compile-time (static) type checking - for reliability
- Inferred types
- Automatic garbage collection; no need for storage management

OCaml - pattern matching and recursion

- Patterns: used to take apart data structures
- Recursive functions preferred over loops
- Functions don't need names - common to have nameless functions
- Common for functions to return other functions
- Compiler debugs type errors

Defining own types

- Discriminant union type: type mytype = | A | B of whatever | C of whatever * whaterev
    - Use pattern matching to examine value of discriminant union

    - Vs C++ unions (very error-prone)
- Generic types: None/Some (replacing null pointers)

Tail recursion: Recursive functions where the recursive call is the last statement executed by the function (where the recursive call is the value returned)

- Less prone to stack overflow: function can return when recursive call is being made, can simply return another function call
    - Caller function is not contingent on subroutine completing

(*) OCaml

OCaml constants: int, real (analogous to float), bool, string, char

- #“H” [char] vs “H” [string]

Basic OCaml operators:

- Integer: addition (+), subtraction (-), multiplication (*), division (div, mod), negation (~)
- Real: +, -, *, ~; real division (/)
- Strings: concatenation (^)
- Ordering: <, >, <=, >= (can be applied to [pairs of] strings, characters, integers, reals)
    - String comparison: alphabetical order
- Booleans: logical or (||), logical and (&&), logical complement (not)
    - *Short-circuiting*: or/||, and/&& don’t evaluate second operand if 1st is sufficient

Comparison operators: equality (=), inequality (<>)

- Some, but not all ML types can be tested for equality/inequality [***equality types***]
    - Equality types: all constants except for real numbers (due to limited precision)

Basic OCaml operators (cont.)

- All operators are left-associative
- Precedence: (not, ~) > (*, /, div, mod) > (+, -, ^) > (<, >, <=, >=, =, <>) > (&&) > (||)

Conditional expressions (OCaml): <u>if X then Y else Z</u>

- Type rules: X must be a bool; Y and Z must share the same type, short-circuit

Type conversion: <u>No automatic conversions</u>

- Need to explicitly call conversion functions to convert between types
    - Trying to add an int and a real without converting will fail, e.g.

Function	Parameter Type	Result Type	Notes
`real`	`int`	`real`	Converts integer to real.
`floor`	`real`	`int`	Rounds down.
`ceil`	`real`	`int`	Rounds up.
`round`	`real`	`int`	Rounds to the nearest integer.
`trunc`	`real`	`int`	Truncates after the decimal point, effectively rounding toward zero.
`ord`	`char`	`int`	Finds the ASCII code for the given character.
`chr`	`int`	`char`	Finds the character with the given ASCII code.
`str`	`char`	`string`	Converts a character to a one-character string.

ML Conversion Functions

Variable definition: val my_var = <value>

- Can use to define a new variable or redefine an existing variable (change value and/or type)
    - Note: does not change the value of/reassign a variable, so much as it defines a new variable "onto" an existing one; any part of the program will still use the old value, e.g.

Tuples: (var1, var2, …, varN)

- Can declare a new tuple anywhere
- Tuple: ordered collection of values (potentially of different types) with length >1
    - OCaml type output: Type1 * Type2 * … * TypeN
- **Indexing**: #i <tuple>

Lists: [item1, item2, …, itemM]

- Can contain any type of element, but every element in the list must have the same type
    - Type output: *elementType list*
- Empty list: nil, [] (Note: no exact type, simply given as *'a list*)
    - Use null <list> to check if a list is empty; can also compare with empty list
- *List operations:*
    - **List concatenation**: <list1> @ <list2>
    - **Prepending one element [*cons*]**: <element; NOT a list> :: <list>

Strings

- *explode*, *implode*: convert a *string* to *char list* (and vice versa)

OCaml Functions

Syntax:

1. **Defining a function**: let fun <args> = <value>
2. **Calling a function**: fun <args>

OCaml functions

- Invoking functions: no need to wrap arguments in parentheses; just separate via spaces
- Function application has very high precedence
    - To enforce a different order, can use parentheses for grouping
- Types are inferred from operations used inside function
- OCaml functions: think math functions
    - Maps arguments → outputs; treat as black box
    - Note: (fun x → [expression] x) ⇔ [expression] <- (both are equivalent)
- Recursion

Functions with multiple arguments:

- OCaml type output: typeOfArg1 -> typeOfArg2 -> … -> typeOfArgN -> returnType <fun>
    - Notice: can pass just arg 1, to create a new function (with N-1 parameters)

Generic types: 'a [any type] vs ''a [restricted to equality types]

Specifying types: fun myFunc(a:real, b:int) : real = <value> [params: real & int; output: real]

- Can place type annotations after any variable/expression

ML: underscore variable (for function parameter) represents a variable whose value is ignored

- *fun _ = "thing"* [ignores value of variable]

ML pattern matching

- *Let (varA, varB) = (1, 2)*
- *Let varA, varB = 1, 2*
- *Let {a; b; c; _} = myTup*

ML match statements: *let X = match Y with | case 1 -> <value> | case2 -> <value> …*

- Recommended (but not required) to match all cases; Y doesn't match any -> error
- List matching: *match L with | [] -> emptylistcase | hd::tl -> nonemptylistcase*

Local variable definitions: Can define local variables/functions, for use solely in evaluating another variable/function

- **Syntax**: let varOrFunc = … let myLocal <args, optional> = <value> in … <rest of varOrFunc>
    - Can use myLocal in subsequent lines [<rest of varOrFunc>]
    - Can define multiple local variables at once: *let val myLocal = 2 val myLocal2 = 3 … in*
        - Optional: put a semicolon after each definition
- **Nested functions**: can use the same syntax to define a helper function inside of another
    - Is only visible from within the larger function; cannot be called from outside of it

OCaml: function names are just bound variables (that happen to refer to functions)

- Can assign variables to functions (e.g. *let subtract = -*)
    - Functional language: function names are not unique, "intrinsic", or permanent (unlike imperative); just correspond to bindings at some point in time
- ***Anonymous functions***: Can create functions without a name (to pass as parameter/return from a function without binding a name, e.g. -> less clutter)
    - **Syntax**: fun <args> => <value>

Higher-order functions: functions that take other functions as parameters and/or produces functions as returned values

- Often used in functional languages (more than imperative)
- **Function order**: Say that a function that is not a higher-order function has order 0; any higher-order function has order n+1 (n: highest order of a function used as parameter/return)

User-defined types:

- Enumerations: type myType = Val1 | Val2 | … | ValN
- Data constructors with parameters:
    - Simple case: *type myType = Value of int | Plusinf | Minusinf;*
        - Initialization: *val myVal = Value 5;*
- Type constructors with parameters: (polymorphic type definition)
    - *type ('a, 'b) myType = None | Some of 'a | SomeElse of 'b list*;
        - Initialization: *SomeElse ["c", "b"];*
- Recursively defined type constructors are permitted
    - *type myType = None | Some of myType;*
        - Analogy: creating a list of lists is a recursive process

Other properties

- OCaml has a garbage collector
- OCaml treats linebreaks as normal whitespace

2: Syntax & Semantics

Syntax vs semantics:

1. **Syntax**: Specifies the form & structure of a programming language
2. **Semantics**: Lays out behavior & meaning of a program

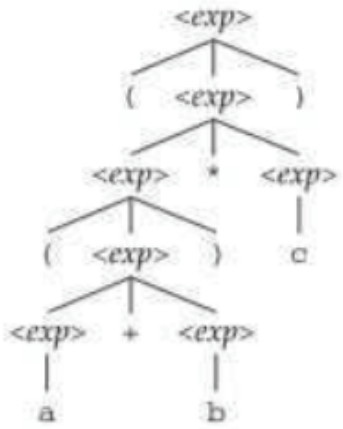
- *Analogy*: Specification (more abstract) vs implementation (more concrete)

Syntax without semantics / semantics without syntax

1. Syntax without semantics: can create English, C++ sentences that are syntactically valid, but meaningless (English)/fails to run (C++)
2. Semantics without syntax (malformed sentences that make sense):
    - English: special rules for "weird" syntax (ex: "galore": adjective, but follows a noun)
    - C++: does not allow this, crashes

Describe syntax using a ***grammar*** - a set of rules specifying how to "build up" a program

- Can use a grammar to generate a ***parse tree*** for a program - a tree data structure, where leaves correspond to the text of the program
- ***Parsing***: Finding the parse tree for a given string/program
    - Language systems must parse a program before being able to interpret and run it

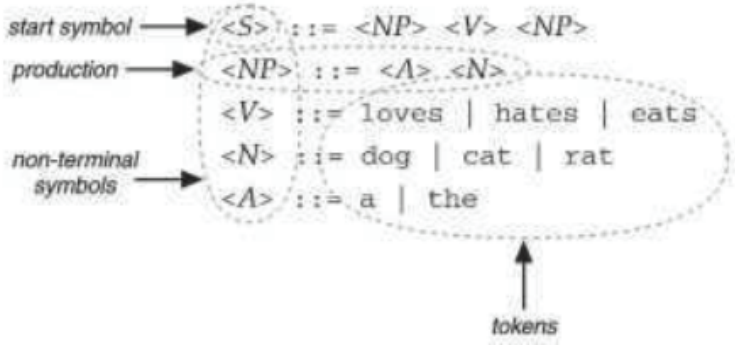


4 components of a grammar: a set of ***tokens***, a set of ***nonterminal symbols***, a set of ***productions***, and a single nonterminal symbol called the ***start symbol***.

1. **Tokens**: Smallest units of syntax; may correspond to a single keyword, name, etc.
2. **Nonterminal symbols**: correspond to different kinds of language constructs
3. **Productions**: A set of rules mapping a single nonterminal symbol (LHS) to a sequence of tokens and nonterminal symbols (RHS)

```
<exp> ::= <exp> + <exp> | <exp> * <exp> | ( <exp> )
        | a | b | c
```

4. **Start symbol**: The nonterminal symbol used as the root of the parse tree

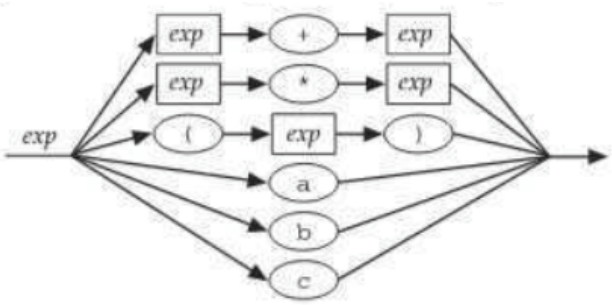


Backus-Naur Form for Grammars (BNF)

Grammars composed of ***nonterminals*** (parse tree internal nodes), ***terminals*** (parse tree leaves)

Syntax charts/diagrams: Another way to express a grammar graphically

- Circles (terminals) vs boxes (nonterminals)
- Branching, loops, shortcuts to represent multiple possible substitutions



Criteria for successful syntax

1. Fits what we know already
2. Simple
3. Unambiguous
4. Readable & writable
5. Redundant (some redundancy)

Criteria varies by audience:

- Automatic systems/parsers: Don't care about clutter, want unambiguity in the grammar
- Humans: Have difficulty reading cluttered grammars

Backus-Naur Form

Backus-Naur Form (BNF)

- BNF: No substitutions on RHS
- EBNF (Extended BNF): Substitutions (via |) on RHS
    - Equally powerful, just more concise

EBNF meta-notation:

- / (substitution)
- * (repetition)
- %dA-B (ASCII characters in the range A to B)
- 1*X (repeat at least once)
- Parentheses: grouping [ex: *(XY) = repeat XY]
- [X]: 0 or 1 Xs
- [A-B] = any symbol between A and B
- [^_] = negation of set
- {X}: 0 or more
- *Set operators*:
    - "-": exception [(sets) A-B = instance of A that is not instance of B]
    - ",": [set concatenation: A,B = instance of A or instance of B]

A grammar being expressible via BNF is "equivalent" to being a context-free grammar (CFG)

- BNF is "grep-able", can use pattern matching
- Some grammars are not pattern-matchable/not context-free; ex: pattern matching can't keep count & remember counts
    - CFGs, intuitively: RHS is generally simpler than LHS, no complicated recursion

BNF: No substitutions on RHS → BNF, EBNF can have multiple grammar rules for the same LHS; causes ambiguity

- EBNF:
- EBNF for EBNF:
    - Syntax = syntax rule, {syntax rule};
    - Syntax rule=meta id=defns list

- defnslist=defn,{“|”, defn};

Formally: BNF grammars described as ***context-free grammars***/**CFGs**: tuples

(Token set {T}, nonterminals {<N>}, start symbol N_0, rule set {LHS → RHS})

CFGs

- Token, nonterminal sets must be finite
- Start symbol must be a nonterminal
- Rules: mappings nonterminal [LHS] → finite sequence of symbols [RHS]
    - At least one rule must have the start symbol as the LHS

Grammar problems:

- **Useless rules/blind alleys**: rules where the LHS is not reachable from the start symbol
- Grammar may not properly express all (desired) constraints
- Grammar may express extra unintended constraints

Programming Language Syntax

Programming languages: Tokens (e.g. if, int) are built out of lower-level components (characters)

- Characters: a [valid] byte sequence (in ASCII, e.g.) represents a character; a character sequence represents a token sequence
- Not every byte sequence is a valid character sequence; similarly, not every character sequence is a valid token sequence (→ compile error, e.g.)

Programming language tokens: What can go wrong?

1. Character confusion: different characters that look similar/identical
2. Reserved words (e.g. programming language keywords)
3. Capitalization for alphabetic letters: accept or don't
4. Greedy tokenization: most tokenizers are greedy
5. Long tokens
6. Comments + whitespace

Lexical vs phrase structure of a language:

1. ***Phrase structure***: Shows how to construct parse trees (with tokens at the leaves)
2. ***Lexical structure***: How to divide a given set of text into tokens

- Often: may define separate token-level grammar for phrase structure, character-level grammar for lexical structure
    - Language systems: scanner/lexer for converting text into a token stream (removing whitespace, e.g.); parser for constructing parse tree

Programming language lexical structures

- Historical languages (punch card-era): *fixed-format lexical structure* (some columns in each line have special significance; may be reserved for statement label, e.g.)
- (Many) modern languages: *free-format* (column position doesn't matter; end-of-line marker is just treated as whitespace)

Operator Notation

Binary operations (2 operands):

- ***Infix notation***: a + b
- ***Postfix notation***: a b +

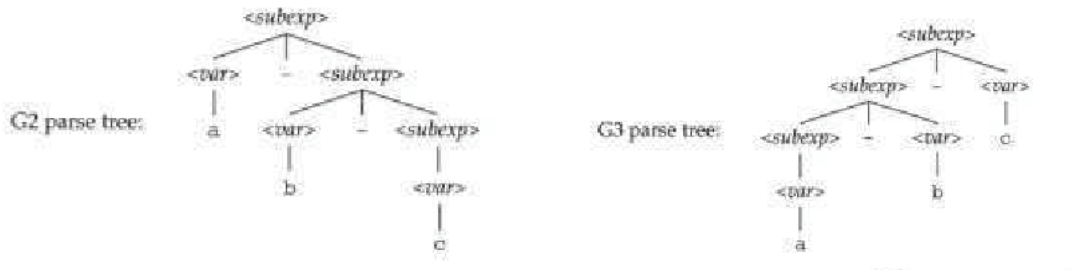
- ***Prefix notation***: + a b

Unary operations (1 operand): *prefix notation* (~a) vs. *postfix notation* (a++)

Programming Language Semantics

Syntax only looks at the leaves (fringes) of a parse tree; ignores internal structure

Can derive ***semantics*** from the internal structure of a parse tree:



a-(b-c) vs. *(a-b)-c*

Ideally: want parse tree structure to correspond to the semantics of the string

Ambiguity occurs when multiple distinct parse trees can be generated for the same sentence

- English: Some sentences have more than one meaning, understood by context
    - Ex. ("Time flies"): "time" noun, "flies" verb vs. "time" verb, "flies" noun
- C++: Some programs may be ambiguous (understood differently by different compilers)

Causes for ambiguity:

1. *(Lack of)* ***Precedence***: Occurs when no ordering is defined w.r.t. which operations should be generated [within the parse tree] before others
2. ***Associativity***: Occurs when [adjacent] equal-precedence operations can be expanded in more than one order

Solutions to ambiguity:

- Enforce ***precedence*** in applying rules
    - Most languages: various levels of precedence (ex: C has 15)
- Complicate the grammar (comes at a cost: more cluttered grammar, harder for humans)
- Design rules to enforce ordering of grammar
    - Previous example: change grammar s.t. <subexp> can only occur to the left of "-" (i.e. always expand right-to-left, or left-to-right; don't allow both)

Most languages: ***left-associative*** (i.e. grammar recurses on the left), with exceptions (e.g. "=", ML "::", etc.)

- Some languages: ***non-associative*** (require explicit ordering via parentheses)

For programming languages, want semantics to be ***unambiguous*** (for each string/program: have only one unique parse tree/derivation)

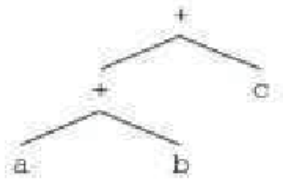
- Resolving ambiguity: language specs aim to answer questions about behavior, reduce ambiguity

Dangling else: common problem of distinguishing *if then (if else)*, vs. *if then (if) else* in programs

- Can be made unambiguous with a proper grammar, but still hard-to-read for a human
- Solutions: No *if*s within a *then* (Algol), explicit *end if* statements (Ada), indented *if*s (Python)

Note: During parsing, many language systems generate and store a simplified version of the parse tree, called an ***abstract syntax tree*** (**AST**)

- General format: AST will typically include a node for every operation, and a subtree for every operand; nonterminals may be left hidden/unspecified



Can use AST as an internal representation of a program (for type checking, compiling, interpreting, etc.)

Ex: Email

Internet RFC 5322: formal standard for what to put in an email

- Outlines what constitutes a valid email (invalid → may not be delivered)
- Email structure: header (nonempty lines) → one empty line → body (anything)

RFC: provides grammar rules for message IDs; header must match RFC format

- Ex: RFC specifies format for message IDs (a unique identifier for an email)
    - Email filter receives multiple messages with same ID → knows to only display one

XML: grammar for data (see also: JSON)

3: Programming Language Systems

Parsing: Given a sequence of tokens, generate a valid parse tree for that sequence

3 main parsing challenges:

1. **Recursion**: Being able to parse recursive rules
2. **Disjunction** (OR): Being able to handle multiple substitutions for a given LHS
3. **Concatenation** (,): Handle rules involving concatenation

Implementing parsing:

- **Recursion**: Easy if the underlying parser technology (e.g. language) is recursive
- **Disjunction & concatenation**: Via a combination of a ***matcher*** and an ***acceptor***
    - ***Acceptor***: Determines if a given string is "acceptable" (via some criteria)
    - ***Matcher***: Given a token string, partitions it into a combination of a matched prefix + an "acceptable" suffix; outputs either a partition (if one exists), or False

Ex. (Programming Languages): language is a set of sentences (sentence is a token sequence); matcher only accepts sentences in BNF grammar

- Matcher: from a start symbol, explore grammar
    - One symbol, multiple rules → must be able to backtrack, try a different rule if first one fails
    - Backtracking: use an acceptor (says if a match is good)
        - Acceptor: fragment → bool
        - Matcher takes token sequence + acceptor, finds suffix acceptable to acceptor
            - Idea: if S → AB, to find rule for A, can use acceptor that matches True only if B takes the output
- Matchers: (acceptor → (frag → (bool)))
    - Equivalently, matchers: acceptor → acceptor (higher-order function on acceptors)
    - Functional programming: can glue together simpler functions (& call them) to create higher-order, more complex functions

Matchers for 3 challenges

- Disjunction/OR: Let S → A, B; need only build matchers for A, B individually (may be simpler)
    - Define Matcher_S (acc, frag) := Matcher_A (acc, frag); if fails, Matcher_B (acc, frag)

- Concatenation
    - cal l matcher_A with special acceptor, only accepts strings s_ that can be matched by matcher_B and the original acceptor (matcher_B acceptor s_)
- Recursion

Compilers

Steps in a Compiler (e.g. GCC):

1. **Tokenization**: convert all instances of every word (e.g. "int") into integer tokens corresponding to that word (based on a symbol table)
    a. Lexene: (token, corresponding meaning/function address) pair
2. **Parsing**: determine if token sequence was valid, and return a parse tree if so
    a. Parse tree: rooted at "program", branches out
        i. Might branch out into "definition" and "sequence" (→ "definition" "sequence") children, e.g.
3. **Semantic analysis**: walks parse tree to determine if it makes sense within language
    a. Ex: check that all tokens are declared before usage, check types (in typed languages)
    b. Static checking: check everything that can be checked about the program before it's actually run
    c. Returns attributed parse tree - like a parse tree, but each node has additional information
        i. Ex: Each node has ID → has ID, type, etc.
4. **Code generation**: converts attributed parse tree into assembly code (foo.s)
5. **Assembly**: converts assembly code into machine instructions (foo.o, binary file)
    a. Binary function leaves blanks in parts of the file; are filled in with memory addresses later during later phases
6. **Linking**: combines machine code with other shared libraries (lib.so) to create an executable (foo)
7. **Loading**: loads & runs file (./foo)

Program optimization: Compilers often automatically perform certain optimizations during code generation for the sake of faster computation or lower memory usage

- Ex. (*Loop invariant removal*): If a given value is recomputed repeatedly in a loop, but is not expected to change in value, then the computation may be moved outside of the loop and only done once

Delayed linking: Instead of linking during the compilation process, can compile without including library code in the final executable

1. *Shared libraries* (Unix)/*Load-time dynamic linking* (Windows): Loader finds library functions and links to program at load time (i.e. right before running the program)

2. *Dynamically-loaded libraries* (Unix)/*Runtime dynamic linking* (Windows): The program makes explicit calls to find and link library functions during runtime

- Advantages of delayed linking:
    - Rather than duplicating a given library function for every program, can instead reuse the same library function (in memory) across multiple running programs to save memory
    - Can update the library function independently of the program without needing to recompile the program
    - Some implementations: unused functions can be skipped during the loading process, speeding up loading

Compilers often try to guess runtime behavior of code when making optimizations

- Ex: In the case of a branch, may try to optimize a more-used branch over a less-used one
- **Profiling**: Rather than guessing runtime behavior, the system can collect *profiling information* (statistics about the runtime behavior of a compiled program) and use it to inform future compilations of the program

IDEs & Hybrid Environments

Interpreters execute a program without prior translation into a language (e.g. machine code)

- Typically: worse runtime performance compared to compiled machine code

Integrated Development Environments (**IDEs**) provide a single interface for editing, running, and debugging programs

- Often have various other language-specific features (e.g. autocomplete) for convenience
- ***Automatic compilation***: In main memory, compiler maps strings to machine code [in RAM]
    - To run, jumps to machine code
    - Program being run is "the same" as IDE (machine code stored in RAM) → faster

Virtual machines simulate (via software) the execution of code on some set of hardware

- Analogous to an interpreter - virtual machines compile high-level code into a low-level language (***intermediate code***) and interprets it
- Advantage: is platform-independent; can run intermediate code on any machine with a compiler
    - Intermediate code sometimes, but not always, machine-level language (machine language simulators are difficult to write)
- Ex: Java programs compiled to, ran as ***bytecode*** by the Java virtual machine

Intermediate language approaches: spectrum from high-level intermediate code (interpreted languages) to machine-level (compiled languages)

Dynamic compilation/hybrid environments (e.g. Java, JS): composed of interpreter, just-in-time compiler

- Interpreter compiles into free bytecodes; executes in a safe environment
- Just-in-time compiler: subroutine called by interpreter; keeps track of how often each function is called, compiles often-used bytecodes into machine code and runs on machine code instead
    - Running on machine code → faster
- Other: OCaml, Emacs, Lisp

Binding and Debugging

Binding: The process of associating a name (e.g. void, foo, for, etc.) with a certain set of properties

Different binding times for different symbols:

1. ***Language-definition time***: Many keywords are included in the language definition
    a. Ex: void, if, for
2. ***Language-implementation time***: Some keywords may be left out of a language definition, implemented differently by different compilers
    a. Ex: the int type (C) has different bounds depending on machine
    b. Language implementations may also introduce limitations not present in the language definition (ex: max lengths on strings and arrays)
3. ***Compile-time***: Many properties (for variables, e.g.) determined at compile-time
    a. References to variables are also resolved at compile time
4. ***Link time***: Definitions for + references to library functions are resolved during linking
5. ***Load time***: Memory addresses are bound at load time, right before execution
6. ***Runtime***: Dynamically-loaded libraries or variable values, e.g.

Early binding (before runtime) vs. late binding (at runtime):

- Early binding: generally faster, more secure; less prone to errors
- Late binding: more runtime flexibility
- Binding times may be specified in language definitions (e.g. Java), or left unspecified
    - Lisp: some variables/variable types may be bound at compile-time, while others are left unbound until runtime
    - Interpreted languages: no early binding at all

Debuggers: Tools to help a programmer find errors in a program

- When programs hit a fatal defect, leave a ***core dump*** (a copy of its memory, stored in a file) → language tools can look at ***traceback*** (function calls), variable values, etc.
- ***Interactive debuggers*** give control over execution; can set breakpoints, look at & adjust variable values during runtime, etc.
- Challenge: debuggers work with executable program (in machine code, e.g.), but must express everything in terms of original source program (in the high-level language)

- Easy for IDEs; non-integrated systems: executable file contains dictionary of source program names for debugging

Runtime support: code included in an executable file, that is not explicitly mentioned/referred to in the original source program

- ***Startup processing***: runs before high-level program, sets up processor and memory for use by the high-level program
- ***Exception handling***: lays out what a program should do in the event of an exception (e.g. write a core dump)
- ***Memory management*** (allocation/deallocation)
- ***OS interface***: interfaces between the program and OS; for supporting I/O operations, e.g.
- ***Concurrent execution***: support multithreaded programming - synchronization, thread creation/destruction, interthread communication, etc.

4: Types

What is a type?

Various meanings across various languages:

1. Set of things a compiler knows about a value
    a. Issue: compiler may not know everything
    b. Issue: Interpreted languages (e.g. Python) checked dynamically
        i. From types POV: compilers = interpreters
2. A type is a set of values
    a. Set elements may share a *common representation in memory*
    b. SETL (language): any set can be a type
3. A type is a set of values + associated operations (popular definition)
    a. See: C++ classes, Python

Two classes of types: ***primitive types*** and ***constructed types***

1. **Primitive types**: "Any type that a program can use, but cannot define for itself"
    a. Ex: int, char, bool, float, etc.
    b. Primitive types are usually defined by languages, but may have differences between implementations (if not specified); ex: C numerical bounds on integers
        i. JavaScript ints are secretly floats under the hood
    c. Typically: built-in types are primitives
        i. Some exceptions; ex: ML - can define (in program) a type equivalent to "bool"
2. **Constructed types**: A type defined by a program (i.e. user-defined types)
    a. Defined in terms of primitive and other constructed types

Floats in programming languages

- IEEE-794 FP (32-bit): Sign bit (1 bit), exponent (8), significand/fraction f (23)
    - When 0 < e < 255:
        - $(-1)^{sign} * (1.significand)_2 * 2^{(exponent_2 - 127)}$
        - Significand 1: "hidden bit" in $(1.f)_2$
            - 1.f is a normalized number (leading digit nonzero)
        - Downsides:
            - Can't represent 0 exactly in normalized form

        - Underflow: normalizing a very small number (smallest possible exponent + leading digits 0) can cause a nonzero number to be rounded to 0 (closest representable answer)
    - For e = 0:
        - (-1)^sign * 2^(-127) * $(0.f)_2$
        - For representing "tiny numbers": nonzero numbers smaller (in absolute value) than the smallest normalized number
        - Can also use to represent zero
            - Issue: two representations of zero (positive & negative zero, depending on sign bit)
    - For e = 255, f = 0 → represents plus-minus infinity
        - Overflow → returns infinity
            - Historically: threw an exception
            - IEEE-794: overflow either traps to hardware (crashes/aborts program), or simply returns infinity
                - Choose based on option
        - 1/0: crashes for integer arithmetic, returns plus-minus infinity (same sign as XOR of 1, 0 sign bits) & continues in FP arithmetic
    - For e = 255, f != 0 → represents not-a-number/NaN
        - Denotes "bad computation"/no answer [not even infinity]: infinity - infinity, e.g.
            - Trap vs NaN choice, similar to infinity
        - Many different possible f's for NaN; no standard for what different sign/fraction values for NaN represent
            - Typically: treat all NaNs the same
        - Comparison (==) to a NaN will always fail (NaN x → x != x)
- Edge cases
    - Comparing two FP infinities x, y: may return different result from comparing representations (bits)
        - Comparing +0.0, -0.0 returns true; comparing bits returns false
- FP operations - strange behavior
    - C sqrtf (float square root) done at hardware level
        - sqrtf(-0) → behavior specified by hardware

Enumerations: constructed types defined by explicitly listing all of its elements

- Internal representation commonly done via associating each element with an integer; may be visible to the programmer or not
    - C: enum elements are integers; can add + subtract, specify which integer is used for each element, etc.
    - ML: only permitted operation is testing for equality

Another division of types: ***aggregate types*** (types containing component values/subelements) vs. ***scalar types*** (types that do not contain component values)

1. **Scalar types**: primitives and enums
2. **Aggregate types**: arrays, tuples, records, strings, etc.

Tuples: an aggregate type that is an ordered [finite] sequence of other types (ex: C pairs)

- Tuple implementations vary between languages
    - Memory representation may be contiguous or otherwise in memory, e.g.
        - Elements may be stored in forward or reverse order, may have gaps in between
    - Different languages: may or may not expose aspects of the memory interpretation
- ***Records/structures***: analogous to tuples, but with named components
    - Components identified by name, rather than index

Arrays: an aggregate type; is defined as the set of all tuples/vectors (of any length) that have the same type in every position

- Similar to tuples: usually supported by, but differ in implementation between, languages
    - Integer indexing only (C/Java) vs flexible counting & indexing (Pascal)
    - The/a array type includes only arrays of a specific length (Pascal), or of all sizes (Java)
    - Different sets of permitted operations; mutable vs immutable; multidimensional arrays

Unions: union of two sets A and B is the set of all elements in either A or B (C unions, ML types)

- How to represent the union of two types with different memory representations?
    - C: allocate enough space to hold the larger of the two types; use the memory representation of the given value (whichever type it is)
        - Issue: may not be able to tell what type a union variable is; same byte sequence may be interpretable as either type
        - C: can interpret a byte sequence (set as one type) as a member of the other type; can use to perform various tricks/exploits

- ML: each value carries its inner type with it, can distinguish at runtime
- ***Discriminated union*** (Ada, Modula-2): a union is a record containing value + enum; enum part indicates the type of the value

Subtypes: A type whose [set of] elements are the subset of another type [set]

- May use same representation as supertype, or make optimizations
  - Ex: int8 as subtype of int16 → can reduce memory allocation; subtypes of structures can eliminate unused fields
- Permitted operations: all operations permitted on supertype (usually)
  - May also define further operations on the subtype (not the supertype)
- General idea: a subset of the values, but a superset of the operations

Functions map objects from one set (its domain) to another set (its range)

- Most programming languages: specify the types constituting domain and range
  - Multiple arguments → domain is (implicitly or explicitly) a tuple
- Languages differ in support for supported operations for functions (beyond calls): storing functions as variables/array elements, passing functions as parameters, etc.
  - Functional languages tend to support a wide variety of function operations

Type Checking and Equivalence

Type annotations explicitly specify the type of a variable or function

- Provides info for humans & compiler; can affect execution behavior
- Type annotations may be required (Java), optional (ML)
    - Prolog: No type annotations at all

Types for inference: Many language systems can use types of variables/function arguments to infer the types of [results from] downstream computations

- The type of an operation may be inferable just from the types of its operands (many languages), or, rarely, may rely on the actual values of its operands (Lisp)
- ML tries to infer a type for every expression

Type checking verifies (before performing an operation) that the type of its operands allows for/is compatible with the operation in question

- Two types of type checking: ***static*** (done at compile-time) vs ***dynamic*** (done at runtime)
    - Static checking (Java, C) provides behavior guarantees, prevents unexpected type errors
        - Many languages: use annotations/inferences to determine types of all variables, functions, operands, etc. during compilation
    - Dynamic checking (Lisp, Python, Prolog) is slower, but provides more flexibility
        - Requires storing type of all variables during runtime (expensive)
            - Small advantage: can explicitly check/test for variable types at runtime
        - Can write programs that run successfully under dynamic checking, but would fail under static checking
- Some languages mix static, dynamic checking
    - Java uses mostly static checking; certain statements (e.g. casts) use dynamic checking
    - Subtypes may require additional dynamic checking due to modifying program behavior
    - Some dynamically-typed languages (e.g. Lisp) may still allow optional type annotations; can perform type checking on annotated variables for efficiency
- ***Strongly-typed languages*** do not allow “cheating” with types - ensures no type-incorrect operation can be performed (without raising an error)
    - Strongly-typed languages may either perform type checking statically (e.g. OCaml), dynamically (e.g. Python), or as a mix (Java)

- Not strongly-typed: C/C++ allow for casting variable addresses to other types to reinterpret bit patterns as different types

Type equivalence: In many cases, the language system may need to decide if two types are "the same"

- Equivalence of two types: ***name*** vs ***structural equivalence***
  - **Name equivalence**: Two types are equivalent iff they share the same name (ex: C classes)
  - **Structural equivalence**: Two types are equivalent if they are constructed from the same primitive types, using the same type constructors, in the same order
    - Ex: C typedefs, ML
  - Many other combinations/variations for defining type equivalence
- *Subtypes*: May also need to check if a given type is a subtype of another

Abstract vs exposed types:

1. **Abstract types**: no knowledge of how type is implemented, represented
   a. Treat as a black box; implementer can change without breaking
2. **Exposed types**: type implementation is known; explicitly included in type definition, typically
   a. Compiler can see & optimize around implementation → can gain some efficiency
   b. Issue: users are bound to a specific implementation; lose flexibility/modularity

- Can also have types that are **partly exposed**
  - Ex (IEEE float): Implementations of operations are explicitly defined (exposed), but representation (knowledge of where bits are, ordering, etc.) is hidden/abstract

Polymorphism

Polymorphism *(for typing)*: A single expression may have multiple interpretations/implementations depending on the types involved

Overloading: An ***overloaded*** function/operator is a function/operator that has at least 2 definitions with differing types (domain; optionally: codomain)

- Many languages: function names can be overloaded
    - C: virtually all functions (even built-in ones, array subscripting, etc.) can be overloaded with user-defined implementations
    - Most implementations for overloading: create separate definitions with unique names (internally), then redirect references at either compile-time or runtime
- Functions may differ in argument type/number, e.g.
    - Ex: operator+ has different implementations for int, float, etc.
    - Language system chooses which definition to use based on context (typically: type and number of operands in the function call)
    - Ex (FORTRAN): cosine has different machine-level implementations (single- vs double-precision) depending on whether input is float or double

[***Parameter***] ***Coercion***: automatic/implicit type conversions done on a function call or operation

- Coercion of parameters for a function/operation call gives polymorphism
- Can cause trouble with overloading if conversions cause overloaded functions to collide
    - Unsigned integers given negative values are converted into UINT_MAX, e.g.
    - Adding a new overloaded function can cause existing calls to fail
        - Many languages: ambiguous conversion → error
- Ex: int → float when performing arithmetic between an int and a float

Parametric polymorphism/generic programming: a type that contains one or more type variables/parameters; can use to write functions/data types to work without depending on input type

- In contrast to *ad-hoc polymorphism* (as seen previously), for built-in types
    - Ex (OCaml): ‘a list → the ‘a field is populated on use, but fixed once set
- Implementing parametric polymorphism:
    - One approach: add a copy of the generic function for each parameter type used in the program, at compile time

        - Allows compiler to optimize each definition for the specific types involved, but requires storing many similar copies of the same piece of code
    - Alternative approach: create only a single generic copy of the function, used by all callers; no duplication needed, but more generic code → less chance to optimize

Subtype polymorphism: If a function parameter's type have subtypes, most languages automatically permit passing in a subtype instance in place of the original type

- Formally: a function or operator has subtype polymorphism if one or more of its parameter types have subtypes
- Ex (traditional Java): all types are subtypes of type Object
    - All objects can be assigned to type Object; to use later, can cast to appropriate type
    - When referencing a generic type, but attempting to access a specific member function: have a separate function for checking if that member exists (hasNext() to check if there is next(), e.g.)
    - Templates (C++, Ada, etc.)
        - Used in lower-level
    - Generic types (OCaml, Java)
        - Used in higher-level

Ad hoc polymorphism (at least two, but only finitely many possible types) vs universal polymorphism (infinitely many possible types)

- Ad hoc: overloading, parameter coercion
- Universal: parametric polymorphism

(*) Java

Java: object-oriented language

- Object: something with its own data, not shared with other objects
- Fields: a component within an object
- Methods: “the things that objects know how to do”

Java primitives: int, char, double, boolean, void, null (+ other number sizes: byte, short, long, float)

- Java specification lays out implementations of primitives; does not leave up to implementation
- *int*: a 32-bit, twos-complement binary number (-2^{31} to $2^{31}-1$)
    - Note: “-” for negation
- Char: denoted with single quotes; 16-bit unsigned binary number, Unicode character set
    - Java supports escape sequences: \t, \n, ‘\’, etc.
    - Is an integral type: can assign an integer value to a char, use as an integer in an expression, etc. (not often used)
- Double: IEEE 64-bit double-precision float-point number
    - Constants: can define with either standard decimal notation, or scientific notation (e/E before exponent)
- Boolean: true or false [case-sensitive]
- void: type is empty set (no values of type void), used as return type
- null: type contains only special constant null (cannot declare a null variable)
    - Can be assigned to any variable of any reference type

Java: any variable that is not a primitive (i.e. any constructed variable) is a reference to an object (call these types, ***reference types***)

- Three types of reference types: any class name, any interface name, any array type

Java String class: no primitive string type; has predefined String class (object/reference type)

- String constant denoted via double quotes (double-quoted char seq -> instance of String)

Java integer arithmetic: +, - (both addition/subtraction and unary), /, *, %

- /: integer division for integer operands

Real arithmetic: +, -, *, /

String concatenation: "+" operator concatenates two String values (is overloaded)

- Generally: "+" means concatenation if either operand is a string
    - One operand is a non-String -> is coerced to a String before concatenation
        - Any primitives can be coerced to String; any reference coerced to a String via its toString() method
            - toString() method: inherited from Object, may be overloaded

Comparison operators: <, <=, >=, > (can be used on any numeric type)

Equality comparisons: ==, != (can be used on any type)

Boolean operators: &&, ||, ! (short-circuiting)

- Ternary/conditional expressions: a ? b : c

Operators with side effects: many operators have side effects (change something about environment)

- Assignment operator: =

Java assignment:

- Almost all Java expressions have a value, can appear on RHS of an assignment expression (exception: call of a method with return type void)
- LHS: expression must have a memory location; anything with a memory location can appear
    - Variables have memory locations; constants (e.g. 1), expressions (e.g. a+1) do not
    - Things with memory locations: local variables, parameters, array components, fields
- Things with memory locations:
    - Appear on LHS -> refers to memory location (*lvalue*)
    - Appear on RHS -> refers to value (*rvalue*)

Most languages (incl. Java): infer lvalues, rvalues via context

- Exception: Bliss (variable names are always memory locations; must explicitly dereference memory location to access value), ML (similar, for type ref)

Java shorthand:

- a+=b, a-=b, a*=b, etc.
- Assignment statement within an expression -> sets assignment (right-associative)
    - a+(x=b)+c -> returns a+b+c; sets x value to b
    - a=b=c -> returns value of c; sets a, b value to c
- ++a (add one to a, return new value) vs a++ (returns old value); similar for —a, a–
    - "Pre-increment vs post-increment"

Method calls: similar to other languages

- Instance methods [obj.method()]: require an object to operate on
    - Reference to object passed as an implicit parameter at start of function call
- Some methods (class methods): do not require an object to operate on/call from
    - Ex: *String.valueof(1==2) = "false"*
- Special case: within an instance method of type Class, can exclude "Class" when calling class methods from type Class (String.valueof -> just valueof, e.g.)

Object creation: *myClass myObj = new myClass(args)*

Other details:

- Java: all operators left-associative, besides assignment operators
- Precedence levels exist; just use parentheses
- Java performs coercions
    - Coerces null to any reference type
    - Coerces any value to String for concatenation
    - Coerces in numeric expressions
    - Coerce between most primitive types, and corresponding predefined reference types (e.g. int -> Integer)

Expression statements: a phrase that gives a command; no value, only executed for its side effects

- Java: every method is a sequence of statements (a "compound statement")
- Expression statement: *someExpression;* [no LHS]
    - Java: requires someExpressoin to actually have a side effect
        - Does not allow (x==y) as a standalone expression, e.g.; must be an expression using an operator with side effects, a method call, or an object creation

Compound statements: an opening brace + any number of statements + closing brace

- Expression statements: all end with a semicolon (not true for all Java statements)

Declaration statements: create a local variable

- Block scope rule: scope extends from point of definition to end of block where it was declared
- Ex: int temp = 5;
- Same syntax for declaring fields of a class

Java conditionals: *if (expr) <statement>* [*else <statement>*, optional]

- Parentheses non-optional, expr must be boolean

Java while: while (expr) <statement>

Java return: *return <expr>;* void methods: no return at all, or just *return;*

Java classes:

- Class declaration: public class MyClass { member variables, methods }
    - Variables, methods: may be public or private
        - Methods: can access fields of object using their simple names
    - Constructor: public ClassName(params) { <constructor body> }
        - No return type
- Static methods: are class methods, not instance methods

Java references & pointers: Java has only references, no pointers

- Java: every local variable is a reference to an object

Java main method: aa static, public method (called main) that takes an array of string values as parameter

- String values: from command line running program
- Java language systeM: calls main method to start program when run
- Any class can have a main method

Java memory model - model of how JVM (virtual machine) works with memory

- Specifies how & when threads can see values written to shared variables by other threads, synchronization
- JMM does not guarantee that instructions cannot be reordered
    - Synchronized: only one synchronized method can read/write to variables at the same time
    - Volatile: a write to a variable is guaranteed to be seen by all other threads; does not prevent potential reordering

Two ways for a Java program to run: as an application or applet

- Application: run by executing main method of a class
- Applet: Java program that runs under a Web browser

- Powers and limited to ensure security; no main method, but have methods for responding to particular events
- Java API: predefined Applet class to provide applet interface

Generics vs templates

- Generics: static checking at definition of method + type (i.e. when type is defined)
- Templates: static checking at type instantiation (when type is used)

Java: all types inherit from type Object

- Note: cannot say List <child> x; List <parent> y = x;
    - If X subtype Y, then every value of type X should be useful in context Y; every op on Y should work on X values
        - Consequently: List<child> is not subtype of List<parent>
- Similarly (C++) char const * is a pointer to char that can't be modified via that pointer (note: only the pointer can't modify char; char itself can be modified elsewhere)
- Consequently:: char* subtype of char const *

Generic programming in Java

- Void (List <String> & ls) → can't grab Object
- Void (List<?> l_) -< represents wildcard
    - Can call general List<Object> functions on the list (e.g. add, get), but not specific subclass functions, grab specific Subclass objects from list, etc.
- Bounded wildcards: <? extends Class>
    - Passed-in argument must be a subclass of Class; can now call Class functions, grab Class variables from the list
- Alternate bound format: <? super Class>
    - Must be either Class, or a parent (somewhere) of Class
    - Might use to add Class objects into list, e.g.

Java templates

- Java - Templates: <T> void func(args)
    - Can bound wildcards based on T: <T> fun(List <? extends T> thing)
    - Multiple templates <T, U>
    - Bounds on templates: <T, U> fun (List<U extends/super T>)

Java type implementation

- Variables are pointers to data; at head of data, have type section
    - Type section (for Java): type field might be a pointer to a type descriptor
- Java: One type descriptor for List (List<Integer> vs List<String> share descriptors, e.g.) for simplicity/performance; use type erasure at runtime (in every case, program only sees List<?> type descriptor)
    - Consequently: must check List contents every time they are used (compiler has to pull out some content, generate a cast)

- Issue: Java type system is confusing
    - Something simpler: “duck typing” (type of an object is a bad notion; just have values/objects + methods)
        - Idea: as long as an calling a method with an object works, good enough
        - See: JS, Python
        - Issue: security/reliability issues

Java: strong for multithreaded; popular for backend/server applications

El Capitan @ LLNL: world’s fastest computer, used for simulating nuclear

- Performance measured via FLOPs on LINPACK (linear programming diagnostic)
    - Issue: LINPACK in FORTRAN, hard to use
- Hardware: MIMD CPUs; SIMD GPUs; matrix cores (matrix arithmetic) + stream processors (vector arithmetic)
    - Performance bottlenecked by memory, not computational power

Today’s phones: more powerful than historic supercomputers

Sun Microsystems (Oracle): 1993 - worksation/server model

- 1993: tried to embed computers in appliances, run workstation code on them
- Issues: embedded world wants cheap CPUs → need to contend with multiple architectures
    - Can try to compile program N times, distribute N copies for each version (ex: Apple x86 → ARM, 2 copies for every program) very awkward
    - 1993: slow Internet, big executables
    - Too many crashes (C/C++ - not safe)
    - C/C++ - long time to build & test

Xerox PARC

- Network: Ethernet, mouse, bitmapped display

- Smalltalk: IDE
    - Object-oriented, interpreted from bytecode (run on interpreter, written in assembly)
        - Different kinds of computers → interpreter for each type; have an interpreter → don't have to deal with portability issue
        - Subscript checker (have interpreter → taking performance hit already anyway; can go for some safety/reliability)
    - Garbage collector

Sun Microsystems (based on Smalltalk): OAK (C++ meets Smalltalk), renamed Java

- C++ syntax & basic ideas, Smalltalk feature set
- Used to write a browser
    - First browser: Mosaic (UIUC, Andreesen)
        - Written in C++: inflexible, crashed a lot
    - Sun: Hot Java (written in Java)
        - Easy copy-able bytecode applets → flexible, Java (more reliable) → less crashes

Java

- Nowadays: used for servers (due to initial design for/use on servers → good multithreading)

Simple Java

- Unlike C/C++: simple inheritance, no pointers (still: references, much less powerful, can't perform conversion shenanigans), garbage collector, primitive types [byte, short, int, lonog, float, double, bool, etc.] are portable (size is fixed, not machine/OS-dependent, behavior is according to standard [ex: IEEE float standard])
    - Goa: simple & reliable language

Java vs C++

- Java: higher level of abstraction [above machine]
    - More portable, reliable; less performant
- Java: bytecodes for abstract stack machines
    - Java: interpreter + JIT compiler (runtime)

Java: single inheritance (exactly one parent class for any object)

Syntax: class C [extends D]

Java arrays

- Java: live on the heap (even after going out of scope); garbage collected only if no references
    - Can return an array from a function without issues, e.g.
- Performance cost (stack [C++] is easier to manage than heap [Java]), but more reliability
    - Java interpreter: look at how an array is used in the program
        - If array is never passed to anyone else (will go out of scope; does not escape) -> interpreter can store array on stack instead of heap
        - Offsets part of performance cost
    - Additional advantage: size of array can be computed at runtime (from variables, e.g.); does not need to be known at compile time
- Java: array size is fixed once allocated (for easier memory management)
    - No specification of how array is stored in language standard, but stored contiguously by most Java systems in practice

Java: Subclass method can shadow a superclass's

- A subclass method can shadow superclass's method (should implement, not redefine it)
- Java: Abstract method (method without an implementation) -> abstract class: any class with at least one abstract method (denoted SomeClass*)
    - Any non-abstract ("concrete") subclass must implement all abstract methods
    - Cannot create an object of an abstract class (i.e. a class with an abstract method) directly; can create variables having the abstract class type, but the new Class(); initializer must invoke a concrete subclass
    - Can create abstract subclasses of an abstract class (same stipulations apply)

Java: Interfaces contain a set of methods, but **no implementations or member variables**

- Interface corresponds to a set of API calls
- Can have interface inheritance (interface X extends Y)
- "Inheriting" from an interface: *class MyClass implements X*
    - Java classes can implement arbitrarily many interfaces (unlike classes), even if it is already a subclass of another class
        - Java's version of multiple inheritance
- Abstract class: analogous to a combination of concrete class (concrete methods) + an interface (abstract methods)

Java: Final classes (cannot be subclassed/shadowed), final methods (cannot be overridden)

- Advantage (for a compiler): can inline the method
    - Instead of adding function calls within bytecode, can simply inline the entire function
- Another advantage: trust
    - Children cannot override final method -> won't break behavior

Java class Object

- Contains "very important"/universal methods:
    - Constructor with no arguments: *public Object()*
    - Comparator: *public boolean equals(Object obj)*
        - Distinct from == operator
        - "Semantic" equality (not address equality)
    - *public final Class getClass()*
        - Class: standard type in Java hierarchy (a subclass of Object)
            - Object: static, compile-time notion (not a real object at runtime) -> allows circular definition
        - Returns the class used to define an object (i.e. in the *new MyClass()*)
        - Final: more efficient, trust [debugging: objects can't lie about their type]
        - "True" definition: *public final Class<? extends X> getClass()*, where X is the class of the caller (i.e. o.getClass() -> X is o's type)
    - *public int hashCode()*
        - Should be consistent with *equals* (two objects equal -> same hash code)
    - *public int toString()* [for debugging, e.g.]
    - *protected Object clone() throws CloneNotSupportException*

        - Can copy/clone any object, but some objects can choose to throw an exception instead of being cloned
        - *throws* statement: static checker ensures caller functions willing to catch exceptions thrown by subroutines
        - *protected*: ordinary users shouldn't be calling this
            - Implement *ClonableInterface* for user-level cloning
- *protected void finalize() throws Throwable*
    - *Throwable*: any possible exception that can be thrown
    - For garbage collection: .finalize() called immediately before garbage collector deallocates an object
    - Nowadays: obsolete
- Other methods: wait, notify, notifyAll (multithreading)

Java Concurrency

Java threads: via Thread objects (direct subclass of Object)

- Each Thread: represents distinct unit of execution
- Can create subclasses of threads
- Most Java concurrency: via *interface Runnable { void run(); }*
    - Thread: has *Thread(Runnable r)* constructor, takes a Runnable
        - Thread state: NEW
    - Run a thread: *thread.start()* allocates OS resources (e.g. virtual CPU, instruction pointer, stack, etc.) and calls *run()*; Runnable computes as normal
        - Thread state: NEW -> RUNNABLE
    - Other actions: *yield()*, *sleep()*, *wait()*, I/O
        - Thread states: sleep() -> TIMEDWAITING, wait -> WAITING, I/O -> BLOCKED
    - Exit: return from run() method [simplest way] -> new state: TERMINATED
        - Thread object still exists, but no longer holds OS resources

Java race conditions: when results of a program differ depending on order of thread executions

- OS may pause threads partway through execution and switch to other threads
- Debugging race conditions: difficult (race conditions occur due to timing -> to replicate, need to reproduce timing; not always possible)
    - One approach: logging environment notates all timestamps (issue: running a logging environment already changes timing; race condition may now not reproduce)
- Java synchronized methods: can add keyword *synchronized* to an object method/block of code
    - If a single instance of an object has that synchronized method being called simultaneously by multiple threads, one thread cannot enter/start the block of code until the other thread completely finishes
        - Low-level implementation: one thread locks at start of function, unlocks at end
    - Java objects: in addition to type & value fields, also have a lock field
        - Used as spin lock: low-level locking function keeps checking the lock (spinning) until it is free (set back to 0, e.g.); changes its value to lock
- Issue: synchronized leads to performance problems
    - Spin locking: time spent spinning is time wasted by processor/CPU
        - More threads, more spin locks -> more waste
        - Doesn't scale well outside of lighter-duty applications

        - Thread trying to acquire spin lock is marked RUNNABLE -> will be scheduled
- Alternative approach: to synchronize within an object, use its *wait()* method
    - *wait()*: removes all locks held by the thread; wait until object becomes "available", then reacquire the locks
        - "Available": up to programmer; via *notify()* method on that instance
    - *notify()*: wake up 1 thread that is waiting on the object
        - Called from some other, non-waiting thread
        - Which thread is determined by OS (effectively, from Java POV: randomly)
    - *notifyAll()*: wakes up all waiting threads
        - Lets the threads decide for themselves who gets to work/be active; gives more control of scheduling to programmer
- Primitives (e.g. wait()) very low-level; Java standard libraries have higher-level ways to use
    - *Semaphore* class: has methods *[s.]acquire()* [waits if unsuccessful], *bool tryAcquire()*, *release()*
        - *size()*: specifies # of threads that can have acquired object at once
    - *Exchanger*: "Rendezvous point" between objects
        - A thread can call *exchange(Value v)* on an exchanger object (with some value) -> waits for another thread to call *exchange()*
        - Another thread calls *exchange(Value w)* -> each thread gets the value of the other
    - *CountDownLatch*: For synchronizing among a large number of threads
        - Holds a number of threads in wait while computation is being done (preventing interference); later, call *go()* to allow all threads to continue
        - Used in scientific programs: for large amounts of computation within/into a global data structure
    - *CyclicBarrier*: Similar to CountDownLatch, but repeating
        - Can use to ensure all threads communicate periodically

Optimizing Java code: Machine code may interleave order of Java statements when executing them

- Reordering statements done for low-level efficiency; may be done by the Java compiler, by the machine microcode, hardware, etc.
    - Java policy: "If a single thread can't tell the difference, compiler or machine can execute instructions in any order" [as-if rule]
    - Exceptions:
        - Locking/unlocking prevent reordering from "escaping" outside of locked section

- Issue: if another thread is executing at the same time -> may retrieve values “out-of-order”, e.g. (if a later statement has been run, but not an earlier one)

The Java Memory Model

??? - Minkowski diagram

- Three regions: future you can affect; past that can affect you; everywhere/when else

Java Memory Model (JMM)

- More complex, analogous structures in other languages (e.g. C++)
- Assumes: each thread is implemented correctly (bytecode compiles to machine code, performs the correct operations)
- **As-if rule**: Reordering is allowed (in arbitrary ways), so long as the observer (e.g. “print”) can’t tell
- Exceptions (for safety - avoiding race conditions)
    - Keyword *volatile*: Prevents reordering operations involving a volatile variable, not even at the machine level
        - Indicates “something that changes without you doing anything” [chemistry]
        - Used for multithreading, if one thread expects the variable’s value to change (and wants to see the update); otherwise, compiler optimizations may break the link between two threads’ variables
        - Similar keyword for C, C++
        - Issue: can’t use to implement locks -> need new keyword
    - Keyword *synchronized*:
        - Has two primitives: *lock()* and *unlock()*
            - JMM: *entermonitor* and *exitmonitor*
- JMM: table for when reordering is allowed

“Can reorder”:

		2nd op		
		Normal Load + Store	Volatile LD + EnterMon	Volatile STR + ExitMon
1st op	Normal Load + Store	1 (yes reorder)	1	0
	Volatile LD + EnterMon	0	0	0

	Volatile STR + ExitMon	1	0	0

- In general: compiler can expand critical sections (protect more than is needed), but can never shrink them

(*) Prolog

Logic programming: 3rd paradigm of programming

- Vs imperative (commands, sequences of commands), functional (functions, function composition)
    - Imperative: lose
    - Functional: lose side effects, variable assignment

Logic programing is built on ***predicates***: statements that are either true, or not

- Can use logical operators (and/&, or/|, not/~) to connect predicates
- No side effects/variable assignment, no functions
- Operate declaratively: user specifies which answers they want/properties wanted, but doesn't specify details on how to get them
    - "Algorithm": Logic (spec/what you want - tells system whether an answer is correct or not) vs control (how to get it - about efficiency)

Prolog Syntax

Prolog is built on *terms*: one of:

- Number
- Atom [a-z][a-zA-Z0-9]*
    - Just a name; only equal to another atom if they share the same name; are never equal to a number/boolean, unlike other languages
- Variable [A-Z_][a-zA-Z0-9]*
    - ? x=1: no
    - ? - x=1: x=1
    - Become bound to terms on success, unbound on failure
- Structure/compound term f(T1, …, Tn)
    - f an atom; T1, …, Tn terms (n>0)
        - n: arity of the functor
    - f used as name, called function symbol/functor [note: this is not a function call!]
    - f/n: function symbol f with arity n
    - Can be more complicated: pr(3, X, xz(X))

Prolog ***clause***: a single universally-quantified logical statement (X :- Y)

- Constrains what it means for X to be true -> gives Prolog system information about X and Y, allows Prolog to make inferences
    - More specific clauses [ab(X, f(Y), Z) :- bc(Z, f(Y), X)]
    - Vs more general [ab(A, B, C) :- bc(C, B, A)]
        - Prolog: can substitute A, B, C to obtain more specific clause

3 kinds of clauses: ***facts***, ***rules***, ***queries***

- **Facts**: correspond to ground terms (*Stmt.*)
    - No logical variables
- **Rules**: conditional statements (X :- Y.)
- **Queries**: (?- X.)
    - Prolog: tries to prove statement false (proof by contradiction)
        - Intuition: Prolog searches through space of possible proofs (depth-first, with backtracking)
            - Some nodes: OR; other nodes: AND
    - If multiple answers: Prolog will find first, ask user if good; user can continue, or stop
        - Will return no (if no/no more proofs); otherwise, yes
        - Prolog can keep continuing until RAM reached (in theory: infinitely)

"Syntactic sugar" (not needed/formally equivalent to what is previously written, just more convenient to use)

- Operators: =<, +, *, etc.
    - X=<Y equivalent to '=<(X, Y)', e.g.
    - =(X, Y) will return true iff the two parameters can be unified
- Empty list: [] equivalent to atom '[]'
- Lists: [*elements*], essentially implemented as cons pairs (like ML)
    - [X, Y, 3] equivalent to '.'(X, '.'(Y, '.'(3, '[]')))
    - [X | Y] equivalent to '.'(X, Y)

Unification: the process of unifying two terms via ***substitution*** (so they become the same)

- Ex: parent(adam, Child) and parent(adam, seth) -> can ***unify*** variable Child, atom seth
    - Afterward: can query "parent(P, seth)" -> output: "P = adam"
    - Multiple variables -> Prolog will attempt to find (and return) any set of values that matches the conditions

        - Can keep querying -> will keep outputting different results; will eventually return "false" once all results exhausted
- Important to Prolog: unify goal to head of clause
    - Ex: goal [member(X, [Y, Z, a])], clause [member(A, [_|L]) :- member(A, L)] -> head of clause and goal don't match, but unify via {X=A, Y=_, L=[Z,a]}
- Similar to OCaml, but more powerful: is 2-way instead of 1-way
    - OCaml: match Value(*variables*) with [Pattern(*variables*)]
        - Bind pattern variables to data values, but not vice versa
    - Prolog: Can bind values in pattern to variables in data
- Prolog language system maintains a collection of facts and rules of inference during runtime
- Issue with 2-way pattern matching:
    - Pattern matching: p(Z, f(Z)) -> creates cyclic data structure (Z is bound to f), creates infinite loop when trying to evaluate

Prolog

- Operates left-to-right, depth-first recursively
    - Runtime efficiency: O(N!)
    - To obtain better efficiency: use differently-structured logic to modify how Prolog interpreter runs through
- Terminology: rules (conditional clauses) vs facts (unconditional clauses)
- Like ML: _ as special symbol, indicating nameless variables

Prolog Examples:

sort(L, S) :- perm(L, S), sorted(S).

- L, S: logical variables (scope: within the statement [Prolog clause])
- :- is conditional clause
- Defining sorted(S) clause:
    - sorted([]). <- notice: unconditional clause
        - Equivalently: sorted([]) :- true.
        - Use as base case for recursion
    - sorted([X, Y]) :- X=<Y.
    - sorted([X, Y, Z]) :- X=<Y, Y=<Z.
    - sorted([X, Y | Z]) :- X=<Y, sorted([Y | Z])
        - Recursive definition

        - [X, Y | Z] matches any list with at least two members; Z: trailing part of the list

Permutation

- perm([X | L], R) :-
    - perm(L, PL)
    - append(P1, P2, PL),
    - append(P1, [X | P2], R).

Append

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

Obtaining results:

- ?- sort([some list], R).
- Returns: R=[]

Prolog: recursive functions

Prolog: simple predicates

- fail/0, true/0
- Loop/0: loop :- loop
- Repeat/0:
    - repeat.
    - repeat :- repeat.
    - Running into repeat -> will backtrack infinitely, keep trying again

Prolog: Writing the wrong code -> program doesn''t match the intent

- Prolog debugger (predicate trace)
    - Places “hooks” when crossing any ports (trying any predicate substitution) -> prints a line when each is reached
    - Each predicate (simply): four ways to “get in/out” [substitute in, substitute from; backtrack into, backtrack out of/fail)
        - Vs other languages: only two (call & return)
        - Reason: predicates can succeed more than once

Peano arithmetic (logical basis fro arithmetic): non-negative integers, +, *, , forall/thereexists

- Z represents 0
- s(X) represents X+1 [successor function]
- plus(z, X, X).
- plus(s(X), Y, s(XplusY)) :- plus(X, Y, XplusY)
- minus(X,Y,Z) :- plus(Y, Z, X)
- Lessthan, greaterthan, etc.
    - Exception: Infinite loops (see above) -> deduce infinity
        - Prolog creates cycles -> creates infinite terms

Prolog: unify_with_occurs_check/2 [built-in]

- Prolog developers: don't want infinite loop (note: default unification has no occurs check)
- uwoc(X, Y) -> unification works unless would create an infinite loop (then, fails)
    - Usage: f(X, Y) :- uwoc(X, Y)
- Slower: cost O(min |X|, |Y|)

Prolog: Control

- Cut predicate (!) -> prunes a branch of the proof tree, tells Prolog not to backtrack
    - Q1, q2, …, qj, 1. -> once Prolog reaches cut: cut succeeds, but tells Prolog not to backtrack to look for other solutions to q1, …, qj
- memberchk(X, [X|_]) :- !
    - memberchk(X, [|_]) :- memberchk(X, L)
- once(P) :- P, !.
    - Runs P once; further invocations fail
- Different meta-predicate: takes predicate as an argument (\+P :- P, !, fail.)
    - \+ _.
        - If call to P succeeds, fail; if call to P fails, succeed (negation)

Philosophy: vertical bar + dash (P provable) vs two dashes (P is true)

- Ideally (within good logic systems): these are equivalent
    - Provable, but not true -> obviously bad logic system; true, but not provable -> logic system is incomplete
    - Godel's incompleteness theorem: all arithmetic/number systems are incomplete
- With diagonal slash: not provable, false

- \+(P) meaning: P is not provable (doesn't mean P false)
- Logically: not provable does not imply not true
    - In practice (closed-world assumption/CWA): I'm telling you everything that's true; if something can't be proven from what I tell you, it's false
        - Issue: doesn't always hold, need to ensure it does

Theory:

- Propositional logic (type of philosophical logic): based on propositions (statement about what's true)
    - Also have connectives (negation, and, or, xor, equality, etc.) - use to build larger propositional statements out of smaller ones
    - An additional connective: p -> q [implies], p <- q [if]
        - Implies: True in all cases, except where p is true and q is false
            - Represent with C <=
- First-order logic: adds logical variables (quantifiers forall/exists), predicates [propositions with arguments]
    - Logicians - procedure/steps for tackling inference
        - Convert each question into a series of clauses (C1 | … | Cn <- A1 & … & Am)
            - Prolog: use Horn clauses (N <= 1)
                - n=1, m=0 -> fact
                - n=1, m>1 -> rule
                - n=0, m>=0 -> query (false :- A1 & … & Am)

Prolog as exploring a proof tree:

- Proof tree: contains nothing nodes (leaves), solve nodes (containing a list of terms)
    - List empty -> solve nod is a leaf; otherwise, one child for each clause
    - Given clause does not unify with head of list at solve node -> child is a nothing node; otherwise, child is a solve node containing list of terms formed from current list of terms + clause from applying resolution step
- Root: solve node containing list of query terms

(*) Scheme

Scheme: (almost) a simple subset of ML

- Very simple syntax
    - Scheme programs have very simple representations as data
        - Routine (in Scheme) to represent programs as data, compute on the fly
    - Continuations: low-level control structure
- Objects allocated dynamically, never freed (like: Java, ML)
    - Scheme - garbage collector
- Dynamic type checking (like Python)
- Static scoping (like most other languages)
    - Most languages use static scoping; except: Lisp, sh
        - Dynamic scoping (value of variables is not determined by program, may vary; shell environment variables, e.g.): harder to debug, but powerful
- Call by value
    - Like most other languages; except: C++ (call by reference)
        - Call by reference: pass a pointer; called dereferences pointer
- Objects: have the usual (numbers, cons, etc.) + procedures (functions)
    - Procedures: include ***continuations*** (very "low-level" functions)
- High-level arithmetic (complex numbers, infinite-width integers, floating-point, etc.)
    - Able to do scientific computing
    - Arithmetic operators are just binary functions
- Has TRO (tail recursion optimization)
    - Under the hood: Scheme will convert a tail-recursive function into a for loop
    - Shorthand: named "let"
        - Ordinary let: (let ((x (+9 3 -7))) (* x (+ x 3)))
        - Named let: (let func ((n 3) (a 5)) (* n a))
            - Defines new function within scope of current/caller function
            - 3, 5 can be replaced with other values, variables
    - Named let: same idea as a loop, but within a functional style (recursion)

Scheme Syntax

- Identifiers: a-zA-Z0-9+-.?*/⇔:$%^&_~@
    - Identifiers cannot start with 0-9+-.

        - Exceptions: “+”, “-”, “…”, “->”
- Comments: start with ;
- Numbers: 12, -19, 12e-9, 2/3 (without spaces), etc.
    - 2/3: exactly stored (not as floating point); multiply by 3 -> get 2 exactly
        - Stored as two numbers internally
- Booleans: #t, #f
    - #f: only value that counts as false; int(0) counts as true
- Vectors: #(v1 v2 … vn)
    - Internally: contiguous, header indicates vector type; after: elements
    - Unlike lists: not stored as pairs
- Strings: same syntax as C (e.g. “\n”)
- Characters: #\c

More Syntax

- Defining a function: (define (name arg1 arg2 …))
- Conditionals: (if (condition) (then) (else))
- Function calls: (+ 2 2)
    - Nested function calls via parentheses
    - Functions return other functions
    - Represented in data as cons(es): data structure consisting of pointers to two objects
        - Function call: first pointer is to function; second pointer to first argument
            - Output of another function as the function -> first pointer is a pointer to another cons
            - Multiple argument: second pointer is to another cons
                - Last argument: second pointer is a nullptr
        - Lists: represented via cons
            - (cons “x” (cons 10 ‘()))
                - ‘() is empty list (terminates list, second ptr is nullptr)

Special definitions:

- Quote, define, if, etc.
    - If: defined specially (not implementable via lower-level functions)
        - Reasoning: Scheme call-by-value => (if c t e) would evaluate both LHS and RHS when calling, could cause runtime error (divide 1 by 0, e.g.)

- Not special: not
    - Informally: anything that can be redefined in terms of a lower-level function (e.g. if)

Scheme quoting: apostrophe before an identifier/expression -> don't evaluate this, leave it as-is without evaluating

- Quote a function -> "simple" function [is an atom: equal only to itself]
- Quote '() is empty list
- (cons (cons 'f (cons 3 '())) equivalent to '((f 3) 4)
    - Formally: 'X := (quote X) [apostrophe as syntactic sugar]

Can create "non-terminated" list: (cons 3 (cons 4 5)) printed as (3 4 . 5)

- Dot indicates list was not terminated

Lambda expressions: (lambda (x) (+ x 1))

- Can nest lambda expressions
- Like ML, can curry (compose) functions:
    - (define f (lambda (x) (lambda (y) (+ x y))))
    - (define g (f 3))
    - (g 7) -> 10

Formatted arguments: (lambda (format . args) (some expression))

- (printf "%d hello %s" 19 "abc")

Quoting & dequoting: `(a ,b(+ c 5))

- Everything after , is evaluated; everything before is quoted

Let scoping

- Vs C: scope of variable extends into its initial value
    - Ex: int x = sizeof(x)
- "Let" - syntactic sugar for lambda (internally, within Scheme compiler)

Notes:

- And: (and E1 … En) returns either false, or value of En (string, int, etc.)
    - Or: (or E1 … En) returns value of first true expression Ei

- Note: and of nothing is true (#t)

Define syntax (similar to C/C++ macros): use “core” operators [special forms] to build up more complex ones

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

(define-syntax or
    (syntax-rules ()
        ((or) #f)
        ((or x) x)
        ((or x y ...) ((let ((xc x)) )if xc xc (or y ...)))))
```

- Weird implementation: to avoid calling x (if it is a function) twice [may result in side effects, e.g.]
- Avoid namespace collisions (someone else defines their own “xc”, e.g.) -> Scheme solution: static scoping occurs before expanding macro, not after
    - “Hygienic macros” (unlike C, C++)

Scheme (+ the Scheme standard): categorize scheme into parts

- Primitives (if, lambda, etc.) vs library (not, and, etc.)
- Required types (e.g. integers) vs optional (implementer can supply, e.g. floating point) vs extensions (supplied by implementer, but not standard)

Scheme mistakes, bugs, etc.

- Implementation restrictions: e.g. RAM, virtual memory
- Unspecified [“partially defined”/constrained] behavior
- Error is signaled (due to exception, e.g.) -> Scheme should indicate, not continue
- Undefined behavior -> implementation can do anything

Equivalence

- (eq? A b) [pointer comparison] vs (eqv? A b) [content comparison, non-recursive]
- (equal? A b) <- recursive comparison

- (= A B) <- numeric comparison (compare bits)

Scheme continuations - “essence of scheme”, controversial

- Scheme interpreter (software or hardware) needs 2 things:
    - ip (instruction pointer): what to do next in current function
        - Ex: %rip in RAM
    - ep (environment pointer): context for instructions
        - In particular: what to do when function returns (caller’s ip), what context to use after leaving (caller’s ep)
- Continuation: every caller function maintains its own ep, ip (maintains “bucket list” of what to do, until end of computation)
    - Every language: uses continuations
    - Scheme (more powerful): at any moment, can view current continuations

Scheme: *(call-with-current-continuation p)*; alt: (call/cc p)

- Takes one argument (a procedure/argument p); creates continuation k, calls procedure p with k as argument
    - A continuation is a function: pass continuation with a given value -> equivalent to returning with that value [(p k) analogous to “return k”]
- Ex (break:)
    - (define (prod ls) (call/cc (lambda (break) *code* (break return_val)) …))

Set values: (set! <name> <action/expression>)

Continuation passing style

<u>Continuations</u>

While evaluating a Scheme expression: implementation must keep track of (1) what to evaluate + (2) what do with the value (ex: if “what to evaluate” is the conditional clause of an if, “what to do” is use it to decide which branch to follow) -> a Scheme ***continuation*** is the “what to do with the value” portion

- At any point, during the evaluation of any expression, there exists a continuation

Scheme: can capture the continuation of any expression via ***call/cc***: takes a procedure p of one argument, constructs a concrete representation of the current continuation, and passes it to p

- Continuation itself is represented by a procedure k; each time k is applied to a value, returns the value to the continuation of the call/cc application
    - In essence: the returned value becomes the value of application of call/cc
    - P returns without invoking k -> value returned by procedure is value of call/cc
- (note: call/cc is only one syntax for continuations)

Ex: (call/cc
    (lambda (k)
        (* 5 4))) -> 20
(call/cc
    (lambda (k)
        (* 5 (k 4)))) -> 4
(+ 2
    (call/cc
        (lambda (k)
            (* 5 (k 4))))) -> 6

Effectively: continuations short-circuit/swap out current procedure, swap in a new one; can use to break from a function (multiplying a list, see a 0 -> call a break continuation [call (break return_val); break itself originally passed as argument to the lambda] to immediately return return_val)

Continuation Passing Style

Continuation passing style: Can make continuations explicit by giving an explicit procedural argument specifying "what to do" with each call

- Ex: [f (lambda (x) (cons 'a x))] becomes [f (lambda (x k) (k (cons 'a x)))]

Allows a procedure to pass more than one result to its continuation, since procedure implementing continuation can take any number of arguments
Ex: (define car&cdr
    (lambda (p k)
        (k (car p) (cdr p))))

5: Memory Management

In programs, commonly need to store various forms of information:

- Contents of variables (esp. large variables)
- Return addresses (ip)
- Environment pointers (ep)
- Instructions
- Language-dependent: I/O buffers, other overhead

Traditional picture (low-level): divide memory into:

- Text: instructions/constants, read-only data
- Static data: Data that is known before running
    - Data (preset/initialized data) + BSS (zero-ed data)
- Heap: dynamically-allocated data; “the part of the BSS that grows”
- Stack (stack, heap grow towards each other; collide -> program ends)

Text, data, BSS are set at compile-time, preallocated, never-freed [simple]; harder: heap, stack

Historical Approaches

- Fortran (1950s): all variables, frames, etc. are statically allocated (no heap/stack)
    - Advantages: very simple (no stack/heap pointers, management); no memory exhaustion; fast (no register-relative allocation, e.g.)
    - Downsides: must trust all code, no recursion, not flexible (restricted to preset size; need to modify source code, recompile to change amount of memory allocated)
- C (1970s): allocate activation records on stack, malloc/free allows for objects on the heap
    - Stack: very cheap to manage; only have to add/subtract from stack pointer
    - Heap: can free objects in any order (more flexible), but more expensive
        - Many more instructions used in memory management
    - Original version of C: activation records were fixed-size, had to be known at compile time (unlike malloc)
- Algol 60 (1960): Like C, but local array size can be determined dynamically
    - Have to store activation record size in memory -> more complicated
- Some languages: less distinction between stack, heap

Memory Locations for Variables

Q: Where are variables stored in memory?

- "Stored in memory" ⇔ "bound to a memory location"

Activation-specific variables: Are bound to a memory location only for a specific ***activation*** (from the start to end of a code block within a function)

- Activation: may be from call to corresponding return for a given function
- Most modern languages: local variables are activation-specific by default
- Lifetime separate from scope (can have local scope but static lifetime, e.g.)

Other lifetimes:

- In many imperative languages, can also declare special variables bound to one memory location for the entire runtime of a program ("global variables")
- Object-oriented languages (e.g. Java): object fields associated with object's lifetime
- *Persistent variables*: can persist over multiple executions of a function

Activation Records

Each activation of a function requires storing various application-specific data in an ***activation record***

- In particular: application-specific variables, return address, other data
- Multiple activations of a function (e.g. via recursion) require multiple activation records
- Sub-blocks within a function may have space preallocated in the function's record; otherwise, can extend & shrink dynamically during runtime
    - Space for local variables can be preallocated in an activation record if sizes are known

Activation record creation

- ***Static allocation*** (early languages): one activation record for every function, allocated statically
    - Efficient (no runtime allocation, variables' memory locations are predetermined), but prohibits more than one activation of a function being active at the same time
- ***Dynamic stacks***: dynamically allocate an activation record on function execution, and deallocate on return; store activation records as *stack frames* in a stack data structure
    - Addresses of activation records no longer known at compile-time -> typically, use a machine register to point to address of current activation record

- Each activation record stores two addresses: return address + address of activation record of previous function (caller)

Handling nested function definitions: many languages utilize references to variables in other/higher-up functions' activation records (ex: functions defined inside other functions in ML)

- Variable reference may not be in immediate caller's record; may be several layers up
- ***Nesting links***: in the current function's activation record, keep the address of the most recent activation record for the function where the current function's definition is nested
    - Multiple layers of nesting may require traversing multiple nesting links; compiler may choose to enforce a limit of maximum depth
- Other solutions: keep all nesting links in a single static array (a *display*), pass each function all the variables it needs as extra hidden parameters (*lambda lifting*)

Passing functions as parameters: nesting link cannot simply be the address of the caller (function may be nested in a different function than the caller) -> to pass a function as a parameter, language system passes both the function implementation and the nesting link to use when calling it

- Can always keep function implementation, nesting link together

Some languages allow function values to persist after the function that created them returns (ex: in ML, can return [from a function] another function that utilizes the returning function's variables)

- Consequence: those languages cannot deallocate an activation record as soon as the activation returns, since the activation record's local variables may be needed at a later point
    - May still be able to optimize and allocate on the stack in some cases
- Can use *garbage collection* to determine when an activation record can be freed

Calling curried lambda functions (define f (lambda (x) (lambda (y) (+ x y))))

- Call (f 12) -> pointer to a cons of instruction pointer, ep points to activation record (of f)
    - Activation record stores value 12
    - Second is "fat function pointer": points to f's activation record [object must survive f's exit]
        - Scheme continuation: ep contains pointer to caller's activation record, contains pointers to pointer to… (a ***dynamic chain***: sequence of caller eps)
            - Also have ***static chain***: who defined us? (see lambda example)
                - Shorter than dynamic, usually

- Scheme: activation records lived on heap (optimized to stack in many cases), garbage collector to clean up

Object oriented programming (initial idea): “tie down” objects via stored activation records from currying, rather than an explicit new/malloc/etc.

Heap Management

In order to handle dynamic memory allocation/deallocation, each language needs to have its own complex systems for runtime memory management

- Ex: imperative languages often involve operations with arrays -> involve allocation
- Naively: a *memory manager* provides methods for a running program to allocate/deallocate blocks of addresses (memory) from some OS-allocated region

Simplest case (**stacks**): due to LIFO structure, allocation/deallocation simply require decrementing and incrementing a stack pointer, respectively ("*bump-the-pointer*")

- *Stack overflow*: occurs if a program tries to allocate more activation records than there is room in memory for

Heaps: regions of memory used for unordered runtime allocation and deallocation

- Runtime allocation/deallocation may be explicit (e.g. C malloc/free) or implicit (e.g. I/O, dynamically-resizable arrays; ML cannot always deallocate activation records in stack order)
- Allocations is last-in, but deallocated can occur in any order

Methods for Heap Allocation

First-fit method: heap manager maintains a linked list of free blocks, (initially one big free block); to allocate a block, the heap manager searches the free list for the first sufficiently large free block and allocates it

- If block is larger than needed, can split the block and return unused portion to free list
- On block free, can return it to the free list (add to front, e.g.)
    - Can keep free list blocks in sorted address order -> allows for ***coalescing*** (merging two adjacent free list blocks into a single larger block)

One observation: in most programs, small blocks tend to be allocated & deallocated more frequently than large ones

- One optimization: can create a separate free list with same-size [small] blocks (a ***quick list***), dedicated to allocation/deallocation of popular small block sizes
    - On deallocation: prioritize sending blocks to quick list if size matches; allocation can see if size matches quick list first before searching the larger free list
- Issue: with a quick list, quick list blocks are no longer coalesced with free list blocks on free

- One solution (***delayed coalescing***): if an allocation fails (no sufficiently large block), can send blocks from quick list to free list, coalesce as usual, then try again

Fragmentation occurs when a heap manager has enough total free space to make an allocation, but fails because its blocks are non-adjacent (preventing coalescing)

Allocation issues

- External fragmentation: enough free space, but too scattered to use
- Internal fragmentation: More space than in an area than requested
    - Ex: user requests most of a free area, remaining space too small to store free list words -> just allocate entire area at a cost

Other approaches to placement

- Best fit: pick block with closest amt. space
    - Circular list (pointer at end points to s)
- Roving pointer: leave free list pointer pointing to last block that worked

Three major questions in heap management: placement, splitting, and coalescing

- Placement: which position in memory to allocate a given block from
    - Heap managers do not know sequence of allocations/deallocations in advance
    - In addition to free lists, have other data structures for tracking free memory (e.g. BST)
- Block splitting: how to split blocks (when requested size is smaller than actual block size)
    - In some cases: may be more efficient to allocate a slightly larger block than was requested, rather than splitting

Approaches to tracking free space

- Keep track of where all objects are, and subtract (slow; want malloc fast)
- Keep a linked list of every free area on the heap [***free list***]
    - Each item: has pointer to free area + size (+ pointer to next entry)
    - Issue: need to allocate space for free list entries -> auxiliary heap (meta heap); must keep track of free space in meta heap -> com
- Alternative approach: free list, but each free area has 3 pointer (containing list payload)
    - Deallocating space: just decrement -> cost: malloc O(1) average, free O(N)

Heap managers can use information about what the running program does with allocated heap addresses to make decisions

Ex: heap managers can use ***current heap links*** (memory locations where a value is stored that the running program will use as a heap address) to inform heap compaction, garbage collection

- Current heap links may correspond to locations on the stack, or within the heap itself

Approaches to tracking roots

- C/C++: don't record roots; programmer tells heap manager exactly where in memory to allocate/free space via malloc/free, new/del operators
    - C: using dangling pointer [pointer after freeing] (in any way whatsoever) -> undefined behavior (ex: comparing to NULL may work, but deallocating will not)
    - Better performance (simplifies heap manager a lot), but much buggier
- When using ***managed heap*** (i.e. having garbage collector):
    - Program tells heap manager (explicit calls) -> troublesome for programmer
    - Compiler records root locations [popular approach (used in Scheme, ML, etc.)]
        - Records in read-only tables, visible to heap manager

Tracing heap links

- Can use knowledge about the language and typing to determine which memory locations do/do not correspond to a current heap link
    - Ex: C integers may or may not be heap links (can cast int->pointer), but Java integers are definitely not heap links
- Set of current reachable heap links corresponds to a *root set* (the set of current heap links on the stack itself) plus all current heap links stored in the heap that can be reached from the root set
    - Can start with a larger root set (set of memory locations of all of the running program's variables; ex: statically allocated, or stored in activation records) and omit all values that are not heap addresses
        - Can (usually) omit variables whose values do not correspond to the address of an allocated heap block
    - From each root set, look at the allocated block it points to and add all heap addresses inside it to the search set; continue until no new memory locations are found

Errors in tracing heap links

- ***Exclusion errors***: a current heap link is accidentally excluded from the set
    - Critical to avoid (for garbage-collected languages, e.g.)
- *Unused inclusion errors*: a memory location is included in the set, but the program never actually uses the value stored there
- *Used inclusion errors*: a memory location is included in the set, but the program uses the value stored there as something other than a heap address (e.g. as an integer)
    - Is effectively unavoidable in C, C++

Heap managers can use current heap links to move allocated blocks around without disrupting the running program; can copy an allocated block to a different region in memory, then update all current heap links to the original block to instead point at the new block

- Can use to perform **heap compaction**: moving all allocated blocks together to one end of the heap, leaving all free space as a single block at the other end (combats fragmentation)
    - Is expensive – may be done as a last resort if an allocation fails, e.g.
- Is only safe if there are no exclusion errors, or used inclusion errors
    - Exclusion error: if a link is not properly updated, can point to an incorrect/invalid location after heap compaction
    - Used inclusion error: a value not meant to be an address may be accidentally interpreted as an address and changed accordingly, resulting in an incorrect value

Garbage Collection

Languages that require explicit heap deallocation (e.g. C) often have to contend with resulting issues

- *Dangling pointers*: a variable may point at a block that has been deallocated -> attempting to access it again will yield either a deallocated block, or reallocated value used elsewhere
- *Memory leaks*: a program may neglect to deallocate a heap block before removing reference(s) to it, resulting in a block that cannot be accessed, but is nevertheless allocated -> memory can't be used anywhere else in the future

Alternative approach (***garbage collection***): some languages only allow the program to allocate memory, not deallocate it; instead, heap manager can find unused blocks and reclaim them automatically

- Typically done by relying on current heap links: if an allocated block is the target of a current heap link, it is still in use; otherwise, it is garbage and can be freed (alleviates memory leaks)

Garbage collector: knows root, object positions

- From information: can deduce which objects are reachable by the program & free heap space

Methods of Garbage Collection

3 major techniques for garbage collection: ***mark-and-sweep***, ***copying***, and ***reference counting***

(i) Mark-and-Sweep: Uses a a 2-phase process:

1. ***Mark***: The garbage collector traces all current heap links and marks all allocated blocks that are targets of a heap link (each object may have an extra mark bit, e.g.)
2. ***Sweep***: Garbage collector makes a pass over the heap, adds all unmarked blocks to the free list

Mark-and-sweep: does not move allocated blocks -> inclusion errors are okay (may cause some garbage to be retained; not a major issue), but fragmentation may still remain

- In practice, don't need to store mark bits with an object; another implementation: Commonly have a concurrent array of mark bits before main part of heap (array size: heap size / object granularity [64 bytes, e.g.])
- Issues:
    - Mark phase is expensive, O(# reachable objects in use)
    - While marking: can't let other computations go on, otherwise program may create objects skipped in mark phase -> deleted unintentionally

(ii) Copying: Heap manager only uses half of its available memory at a time; when that half fills, copies all non-garbage blocks (determined by current heap links) to the other half, compacting as it copies

- Can resume normal allocation in the new half; old half is the new unused part
- Moves allocated blocks -> combats fragmentation, but inclusion errors are bad

(iii) Reference Counting (e.g. Python): Rather than tracking current heap links, each allocated heap block includes a counter/word that keeps track of how many copies of its address there are

- Language system maintains counters, increments/decrements as references are copied or discard; when a counter becomes zero, can immediately free that block
    - With every assignment to an object, update associated reference counts (more overhead)
- Upside: whenever a reference count drops to 0, can free it immediately; don't need to wait for a garbage collector
    - When reclaiming object, update reference counts associated with pointers in that object (and free downstream objects if needed)
- Benefits: simple, memory management cost is O(1), no mark-and-sweep overhead; safe
- Issues:
    - Poor performance: maintaining counters adds overhead even to simple operations + more storage used to hold reference counts
    - Cannot garbage-collect cycles of garbage (i.e. a set of garbage blocks that keep a cycle of pointers to each other) [Python: any cycles exist -> leaks memory]
        - Ex: "p[1] = p" -> have to increase reference count of p by 1; if p is ever reassigned, leaves reference count of p at 1 -> never freed
        - Python, later (hybrid approach): use reference counts in normal operation; if memory gets low, use mark-and-sweep

Incremental collectors: similar to mark-and-sweep, but only recover a little garbage at a time (rather than making large one-off passes, as in regular mark-and-sweep)

- Attempts to solve an issue of mark-and-sweep (performance is generally good, but requires occasional long pauses while garbage collection is happening); may be used in real-time systems where delays are unacceptable
- ***Real-time garbage collection***: do a small amount of GC whenever a new object is allocated (less disruptive, but less space is deleted)

"Faster G.C." (modern Java): want to make allocation fast; is based on two ideas:

- ***Generation-based collector***: partition heap into several "generations" (depending on time of creation); when allocating new storage, always allocate in the "youngest" generation
    - Youngest generation keeps two pointers: heap pointer and limit pointer [at end of generation]; all free space is in region between heap, limit pointer
        - When allocating: just increment heap pointer
        - When youngest generation fills up (heap pointer passes limit pointer), either allocate a new generation or run garbage collection [which one: depends on intuition]
    - General rule: objects will refer to older objects, but the reverse is much less likely (can only refer to older objects in functional programming)
        - When youngest generation fills up: only need to run garbage collection on the youngest generation(s), much smaller
    - G.C. on older generations: via various heuristics
- ***Copying collector***: don't garbage-collect in place (in youngest generation); instead, upon finding an object, copy that object into a new generation & update the root the point to copied object
    - Pointers within copied objects -> remap these too
    - Once done: can deallocate the entire old youngest generation and return to OS
    - Issues: have to copy objects; have to update roots
    - Benefits:
        - Once finished, all free space in G.C.'ed generation has been moved to the end of the new generation (no fragmentation + better caching: All copied objects placed adjacent to each other -> very small cache line [in CPU])
        - Cost of mark-and-sweep is proportional total # objects in system (incl. Objects that are being freed); cost of Java G.C.: only proportional to # objects in use/accessible (not total # objects)

Another issue: Java has a finalize() method; called whenever G.C. notices that an object is garbage, immediately before it is deleted

- With faster G.C.: don't look at freed objects -> doesn't work anymore
- Java: like Python, two G.C.s (regular mark & sweep, supports finalize; copy collector, preferred)
    - Class has finalize method -> allocated the slow way, otherwise copy collector
- Recently: Java deprecated finalize (changed language to provide performance improvements)

- finalize() quirk: in finalize(), may create a new pointer (global variable) to self -> Java G.C. has to know not to free that object

Another complication: Java is multithreaded -> how to handle heap pointer among multiple threads?

- Standard approach: every thread gets its own youngest generation (+ heap, limit pointers)
- Another approach: each thread maintains its own private free list (without putting them on the explicit free list); when allocating, can first check private free list for space before going to call malloc
    - Issue: works well in C/C++, traditional G.C.;s, but in Java: will copy private free list (list of garbage - don't need to keep if we're copying anyway)

Garbage collection varies between languages

- Some languages (e.g. Java, ML) explicitly require garbage collection; others make it optional
- Certain languages (C, C++) make implementing garbage collection difficult, albeit not impossible

"Two worlds": fast & dangerous (C, C++) vs slow & safe (garbage collectors)
-> Another approach (conservative G.C.):

- #define free(p) ((void) 0) [comment out all frees, essentially]
- Implement a garbage collector (mark & sweep, without root knowledge):
    - When running mark & sweep: look at entire stack (low level: starts at 0x7800000…, ends at stack pointer) and find all "potential pointers"
        - Ex: assuming heap operates a certain memory range, find all bytes that fall within that memory range (correspond to a heap address)
        - Check if that memory address corresponds to an object in the heap -> if so, say it is a pointer to that object
            - Idea: can have memory leaks (if mistaken), but never a dangling pointer
    - Do same process recursively within heap objects + with all registers

[Can be used in C/C++; often used in large C++ programs (used by GCC, Emacs, etc.)]

6: Aspects of Programming Language

Object-Oriented Programming

Object-oriented programming: programs are designed using objects – little bundles of data that know how to do things to themselves

- Can utilize object-oriented programming in virtually any language (even non-object-oriented languages); conversely, not every program in an object-oriented language is an object-oriented program

Object-oriented programs based on objects with methods – implementations of some function based on that specific object's characteristics

- Instead of needing to include lots of branching in outside functions based on an object's type, can instead call object's method and allow it to use the appropriate implementation
- Good for standardization and information hiding

Most object-oriented languages support ***classes***: groups together object fields/methods and allows for instantiation (i.e. creating as many objects of that class as needed)

- Can also use classes as a unit of inheritance (as in Java), as types (for static checking), or as namespaces (for scope control)
- Many languages allow static fields (only one instance, shared across all objects of that class) and static methods (are called without an instance/object; can only access static fields)

Prototypes: objects that are copied to make other, similar objects (alternative to classes, more flexible)

- Can use a prototype as a base-level object; copies may add/remove fields, methods as needed
- *Prototype-based languages* (e.g. Self) have constructs for prototypes; can be classless
    - Consequences of classlessness: no static fields/methods; difficult to do static checking; no framework for a static inheritance hierarchy (may instead do *delegation* dynamically)

Inheritance: A relationship between two classes (a base class and derived class), where the derived class "receives things" from the base class

- Implementation details vary between languages:
    - Can a derived class have multiple base classes?

        - Does a derived class have to inherit all methods/fields of the base class, or can it pick & choose which ones to inherit?
        - Is there a common root class (as in Java's Object hierarchy)?
        - What happens when a derived class has a method/field definition with the same name and type as an inherited one?
            - Java: Supports overloading, but not always; can declare non-overloadable functions (*final*), cannot overload a function with a function with more restricted visibility (e.g. overload public with private)
        - Does a derived class inherit specification (i.e. method obligations) from the base class?
        - Does a derived class inherit membership in types from the base class?
    - Prototype-based languages use ***delegation*** instead: when an object gets a method call that it cannot handle, can arrange to delegate it to another object
        - An object can delegate to the object from which it was cloned (simple case), or choose arbitrary objects to delegate to and change delegation dynamically (more complex)

Encapsulation: can use objects for encapsulation (i.e. limiting what parts of a program are visible to different parts of the program)

- Java: Objects can specify the visibility of its methods & fields
    - When creating a variable with an interface as its declared type, can only use those methods that are part of the interface

Subtype polymorphism: declared type may be a base class of/interface implemented by an actual object's class

- Requires ***dynamic dispatch***: language system has to store objects' exact class in memory + look up implementations at runtime

Names, Identifiers, and Scope

Issue: A single name/identifier may be associated with multiple things

- Ex: an int variable is associated both a value, and a type + size (in memory); will obtain different results for *sizeof i* vs *sizeof 27*, e.g.
    - C: names are bound to: value, type, address, alignment, etc.

Bindings: An association between a name and a value

- Assignments establish a binding between a name and associations
- More low-level: a set of bindings (a dictionary, namespace) binds byte sequences [names] to other byte sequences [values]
    - Set of bindings can be determined explicitly (via assignment, e.g.), or implicitly (in a lambda function, e.g.)
        - Lambda function (fun i -> i+1) -> implicit binding when called
- Python: No bindings of names to types (dynamic type checking)

Binding time: when a name becomes bound to its value

- Different aspects of a value may be bound at different times
    - Static binding (compile time): “int i = 27;” -> name is bound to value at compile time
    - “Int i = 27;” -> name is bound to an address at “block entry time”/declaration time [upon entering the relevant curly brace-enclosed code block]
        - Some compilers: bound earlier
- INTMAX: bound at platform definition time [hardcoded into machine architecture]
- Static variables: bound to address at link time

A ***definition*** is anything that establishes a possible binding for a name

- “Possible” binding: one name may have multiple bindings in different places
- In addition to explicit definitions, can have implicit ones; ex: Fortran treats unidentified names as either variables or integer variables, depending on the name

An occurrence of a name is ***in the scope of*** a given definition for that name whenever that definition governs the binding for the occurrence

- Scope (of a name): set of locations in program where name is visible

Most languages use ***blocks*** – a language construct containing definitions, as well as a program region where those definitions apply – to help solve the scoping problem

- Ex: Function definitions

Most modern languages are ***block-structured***: use blocks to delimit scopes

- Most languages also support nesting blocks, redefinitions
- "*Block scope rule*": the scope of a definition is the block containing that definition, from the point of definition to the end of the block, minus the scope of any redefinitions of the same name in an interior/nested block
    - ***Shadowing***: in C, can redefine a variable within a smaller scope -> within only the smaller scope, that variable will now map to a different value
        - Can distinguish between inner namespace, outer namespace

Most languages also have ***labeled namespaces***: language constructs that allow a definition of a name to be accessed, outside of the block where the definition is normally in-scope

- Advantages:
    - Help reduce namespace pollution, mitigating name collisions
        - Often: want to minimize size of a scope (information hiding)
    - Within a namespace, can decide which information is visible (e.g. public vs private)
- *Primitive namespaces*: part of a language definition; may define primitive types (e.g. int)
    - Ex [C/C++]: int f() { struct f{ float f; } f; #include <f> #define f int }
        - f has multiple meanings: the function, the struct tag, a member of the struct, a local variable of the function f(), a package name, a type definition
            - Curly braces mark a new namespace
            - Compiler knows string following <type> is an identifier
- Ex: ML structs, C++ namespaces, Java packages

Dynamic scoping: Scoping is not fully determined until runtime (as opposed to static scoping)

- "*Dynamic scope rule*": The scope of a definition is the function containing that definition, from the point of definition to the end of the function, along with any functions called (even indirectly) from that scope – minus the scope of any redefinitions in those functions
- Used in very few modern languages (some dialects of Lisp, e.g.)
- Issues:
    - Difficult to implement efficiently due to the need for runtime lookups

- Can result in large and complicated scopes

C: Distinguishes between full definitions vs ***declarations*** – definitions that give the function name and type, but not the body

- Both are "definitions" in the scoping sense; allow for separate compilation & downstream linking at a later step in the compilation process

Separately compiled modules

- Only some information shared between modules -> difficulties
- Names + terminology
    - Declaration vs definition
        - Definition: within a module, have a full specification of a (function, e.g.)'s implementation
        - Declaration: a stripped-down version of the definition, provided to other modules (has to match definition)
            - Gives compiler some information about implementation

Information hiding for modularity

- Can declare, for each identifier, whether it is externally visible:
    - C: static (private to this module) vs extern (visible to all)
    - Java methods: public, protected, private, [default] (different properties for to whom the method is visible); Java package hierarchy
- Explicit namespace
    - Full namespace -> edit -> (typically:) more abstract namespace
        - Python: namespaces are dictionaries -> can be dynamically modified
        - OCaml: operators at compile time
            - Signatures: static checking, more flexible than Java

Addressing Errors

Ways to address errors/faults/failures (bugs)

- Errors: humans
- Faults: programs
- Failures: program produces different output from intended (behaviors)

Addressing in a programming language

- Compile-time checking [static checking]: static type checking, e.g.
- Preconditions: assign conditions to check validity of values before running code
    - Special notation (visible to both programmer & compilation), corresponds to an obligation on the compiler to ensure input is valid
        - During compile-time: compiler checks that inputs will always be valid; breaks if there is a possibility of invalid inputs being passed-in
    - Used in high-reliability languages (e.g. Eiffel)
- Total definitions: make sure compiler does something reasonable, no matter what the definitions are
    - Ex: instead of sqrt() crashing on receiving negative number, just return a NaN
    - Ex: Rust - no exception handling, but provides a way to return a value indicating "something went wrong"
- Fatal exits (crash the program): abort()
- Throw an exception
    - Exception-handling: try-catch vs explicit error-checking (via if, e.g.)
        - Try-catch: more concise, cleaner; less granular [less reliable?]
        - Former more used in scripting languages (less reliable), latter reliable languages
- Checked exceptions (Java): every method specifies the set of exceptions it can throw
    - Built-in Throwable class

Parameter Passing

Notation: Actual parameters (parameter values passed in a function call) vs formal parameters (variables in the called method corresponding to the actual parameters)

- Parameters given, formally, as a *parameter list*

Correspondence: How does a language decide which formal parameters are associated with which actual parameters?

- ***Positional parameters*** (most common approach): based on the ordering of the arguments (parameters) in a function call
- ***Keyword parameters***: write out the name of the formal parameter next to the value
    - Can use with positional parameters, e.g. in Ada/Python: f(y=5, x=10)

Some languages offer *optional parameters* with default values; use the default value for a formal parameter if no corresponding actual parameter is provided

Some languages (e.g. C, JS, Python): allow actual parameter lists to be of any length

- *Varargs*: Python (x, *y) pass all arguments past 1st in a tuple
- C uses (formal params, …) to indicate allowing additional arguments
    - Issue: no static checking for trailing arguments

<u>Mechanisms for Parameter Passing</u>

Pass by value (simplest): The formal parameter is like a local variable, but is initialized using the value of the actual parameter before the function begins executing

- Changes to the formal parameter do not affect the actual parameter; can use mutators (i.e. external functions) to make changes outside
- References as values: any changes made to the object pointed to by the reference are visible to the caller, but changes to the reference variable itself are not
- Issue: copying large values is expensive, inefficient

Pass by result ("**copy-out**"): The formal parameter is like a local variable, but left uninitialized; after the called method finishes executing, the final value of the formal parameter is assigned to the corresponding actual parameter

- Actual parameter is never evaluated by the called method; is only assigned to

    - Must be a type that can be assigned to (not const, e.g.)
- Pass-by-result as the opposite of pass-by-value: acts as a way for the called method to send information back to the caller

Pass by value-result: Combination of pass-by-value and pass-by-result: formal parameter is initialized to the value of the actual parameter at start of execution; after the method finishes executing, the actual parameter is assigned the value of the formal parameter

Pass by reference: The address [lvalue] of the actual parameter is computed before the method executes, and used as the lvalue for the corresponding formal parameter

- Formal, actual parameter are (effectively) two names for the same variable/memory location
    - Changes to the formal parameter also affect the caller
    - Analogous to passing a pointer by value in C
    - Benefit: Previous methods require copying values into/out of method activation records -> can be expensive for large objects/values, e.g. arrays
- No action needs to occur when the method returns
- f(&(int) (b+7)) -> compiler creates temporary value for lifetime of function to represent copy of b+7; disappears once function ends
- Cheap to call references + lets caller see computed values without needing to copy and return
- Issues: slower for small arguments (need to dereference memory), aliasing
    - ***Aliasing***: The same variable having multiple *aliases* (e.g. the formal & actual parameter above) results in hard-to-understand programs and potentially unpredictable behavior
        - Two different formal parameters might end up having the same address in memory -> can cause issues [not an issue in by-value-result, e.g.]

Macro expansion: macros look like regular function calls, but rather than jumping to a different instructions, each instance of a macro is simply replaced with a complete copy of the macro body during the pre-processing process of compilation

- Is done via exact text replacement (replacing macro formal parameter names with the actual parameter names) -> may encounter issues if the macro declares a variable with the same name as one of the passed-in actual parameters (*capture*), e.g.
- Is done statically -> during runtime, has no associated activation record; is (often) simply a textual substitution, cannot usually be done recursively
    - Macro body has access to caller's local variables

Pass by name: Pass a function (thunk) telling the function how to evaluate the parameter; each actual parameter is evaluated in caller's context on every use of the corresponding formal parameter

- Pass two parts in the parameters: the code for the actual parameter (i.e. how to evaluate its value), and the nesting link to use with it
- Slower (need to call function), but avoids crashes/loops
    - Ex: pass a=i, b=i+1 as (formal)=(actual) -> upon adding 1 to a, consequent references to b will see a value incremented by
    - Ex: f((lambda () a)) -> argument gives an expression to compute a

Pass by need (Haskell): Similar to pass by name, but the actual parameter is only evaluated on first use of the corresponding formal parameter (with the obtained value stored and always used thereafter)

- Subsequent uses of the formal parameter do not reevaluate the value

Eager evaluation (actual parameters are always evaluated; most) vs *lazy evaluation* (pass-by-name/need: actual parameters are only evaluated if the corresponding formal parameter is actually used)

Most function calls also have extra hidden parameters (*transformed parameters*):

- Object-oriented methods: extra hidden parameter is pointer to current object (this/self)
- Static chain: Pointer to definer's frame
- Dynamic chain: pointer to caller's frame
- For return address: %rbp
- For address of TLS (thread local storage): top word on stack

Many objects have "extra slots":

- User slots
- Type-related: Prototype, representation of type (for dynamic type checking, e.g.)
- GC-related slots (e.g. for reference counting/marking)

Cost Models

Every programmer has a ***cost model*** – a mental model of the relative costs of various operations (in terms of resources needed for runtime execution) –for the languages they use

- "Relative costs": how the performance of one operation might compare to another, speed vs ease-of-use tradeoff (recall: Big O notation for asymptotic efficiency)
- Other costs: real time, memory, system time (overhead), CPU time (sum of threads' tiem), energy & power (= energy/time), I/O space & # OS operations, network access/latency, etc.
- HPC challenge: keep data near the registers where they'll be needed for better performance

Cost Models for Lists

Prolog uses *cons cells* (value, pointer pairs) to make its lists

- Writing E=[1|D] (where D is an existing list) creates a cons cell where the second entry points to D -> we know this can be done in O(1), no matter what the size of D is
    - Less efficient: E=[C|D] for two lists C, D requires copying C; O(|C|)

  Getting the length of a list is O(N)

Prolog may attempt to unify lists: have two lists with value [1, 2] stored separately in memory -> will check if they are the same and, if so, make them point to only a single instance of that list in memory

- Consequently: Prolog unification is O(min(|x|, |y|)) for 2 trees X, Y

Similar to Prolog: Lisp's lists represented in RAM as randomly-placed cons pairs (note: RAM -> random placement not an issue)

- (append a b) creates a copy of a; last element points to original b in memory
    - Cost: O(|a|) -> cost model: (append '(foo) x) [O(1)] <> (append x '(foo)) [O(|x|)]
- Multiple equality tests in Lisp: equal (checks value) vs eq (checks structure; faster), e.g.

A different approach – in Python, Lists are contiguous arrays

- List variable: holds the number of elements, the amount of space allocated, and a pointer to the array in memory
    - Cost model: list.append(foo) is normally cheap [O(1)], O(|list|) when the list variable is out of memory -> worst-case O(|list|)
        - In general: still more performant, O(1) amortized

Can use cost models to inform decisions in other parts of the program

- Can see that a add-front implementation for a Prolog list reverse would be faster than append-back, e.g. [O(N) instead of O(N^2)]

Cost Models for Function Calls

Traditional picture: each function call pushes a new activation record onto the stack; when a function returns, it writes its return value to the caller's activation record and pops its own activation record from the stack

- For recursive programs: each recursive call adds another activation record to the stack

Tail calls: Function calls where, once the function returns a value, the caller function does not additional work and simply returns the returned value as its own return value

- ***Tail-recursive functions***: recursive functions where all of their recursive calls are tail calls

Many language systems have ***tail-call optimization***: for tail-recursive functions, rather than allocating a new activation record for the recursive call, can simply reuse the caller's stack frame for the callee

- No cleanup needed for the original function; new callee can return directly to the calling function's caller
- Unlike regular recursive functions (keep nesting -> may eventually run out of memory), tail-call-optimized recursive functions can (functionally) recurse infinitely

Consequently: knowing a language system optimizes tail calls gives programmers a cost model for tail calls; know to gravitate towards approaches that utilize tail calls

A Cost Model for Prolog Search

Different logically-equivalent ways of specifying Prolog statements might nonetheless lead to different time-to-solve from the Prolog system (or may fail to be solved at all)

- Ex: Restricting early is generally best

A Cost Model for Arrays

Change array access order for temporal locality

- Accessing arrays sequentially is faster than non-sequentially
- Row-major (rows kept together) vs column-major (columns kept together) order -> affects what form of sequential access is better

Low-level:

- Use simpler operations to be faster (e.g. shifts instead of multiplications)
    - All primitives are size 2^k for this reason
- Reliability (e.g. subscript checking): Conditional branches (jl/jge, etc.) slow down programs

Cache effects: cache divided into cache lines -> want to fit data onto cache

- Need to consider locality, array access order
- Cache hashes: Given a virtual address, use middle subsequence of address as index to cache [cache hash]
    - Consequently: can increase array size to prevent hash collisions for better performance

Want to avoid ***spurious cost models***: beliefs/mental models that do not correspond to physical reality

- Ex: making optimizations that, in reality, are done by the compiler automatically anyway (and also clutter up the program, potentially)

(∗) Rust

Rust: coming to Ubuntu 25.10, Linux kernel

- Static memory safety (checked at compile-time)
    - Vs Python, Scheme, OCaml, etc. (checked dynamically/at runtime)
- Both static & dynamic checking: most of the time (not for all memory accesses)

Some PHD dude: static checking of encryption-based programming

- Goal: Want a program to be able to static-check E(y) = f(E(x)), where f is some unknown function for encryption

Rust's goal: "**Zero-cost abstraction**"

- Via an ***ownership & borrowing*** system (statically): at any point in program, can determine who owns the object
    - Can use static checking to prevent modification/mutation of shared state helps mitigate data races
    - Only one owner of memory at a time
        - Either: 1 reference to mutable, or >=1 reference to immutable

Ownership also combats garbage collectors: when reference variables drop out of scope, if it owns the object, invokes object's drop() method

- drop() cleans up object, goes into all slots and drops them too [transitive closure]
- Can overload drop() method for additional cleanup
- Rust: "immutable" ⇔ "shared access" [object is in a state where more than one thread can access it]
    - Can later become mutable (immutability is not a permanent state)
    - C/C++: data races can occur on data that is both shared and mutable
        - Functional programming: no mutability (no assignment)
        - Rust: data cannot be simultaneously shared and mutable

Other parts of memory safety:

- Runtime bounds-checking (if violated, panics)
    - Consequently: Rust arrays are not just pointers (unlike C), has to have associated size information (essentially: pointers with bounds)

        - No static bounds-checking
- No pointer arithmetic, no “free”/”delete” operations, no null pointers
- Rust tries to solve (in normal Rust): use-after-free (dangling pointers), double frees, buffer overrun, data races, null pointer dereferencing

Rust: provides ***unsafe*** keyword; unsafe code -> many C/C++

- May use unsafe for better performance; Rust programmer goal: maximize amount of program in safe Rust, restrict usage of unsafe to smaller sections
    - Can limit security checks to unsafe Rust, mostly
- Analogy (Linux): relatively small kernel [low-level C, unsafe games], much more applications [talk to kernel with system calls; kernel prevents interference -> don’t have to worry]

Rust exceptions - two forms of errors:

- panic(message); -> program immediately crashes/halts
    - Prints backtrace to help debugging, unwinds stack (drops objects), exits
        - Can configure this behavior: change build configuration
- enum Result<T, E> { Ok(T), Err(E) }; (similar to union) -> either holds Ok + object of type T, or Error + object of type E
    - Allows caller to figure out what to do with result; good for reliable code
        - In some sense: programmer writes their own exception handler

Rust has Cargo build system: handles dependency management, building, testing, benchmarking, documentation, lint checker, formatter, etc.

7: Semantics

Semantics: “What does a program mean?”

- Vs. syntax (easier)

Recall: Once a grammar has been used to construct a parse tree, most language systems only retain a simplified representation/structure (an abstract syntax tree/AST)
-> **Semantics**: Given an AST for a language, how do we know what the AST means? (i.e. What happens when it is evaluated)

Natural semantics: define a relation between ASTs and their results when evaluated

- Can write as a collection of rules E->v [evaluating an AST E produces the final value v]
    - Similar structure to mathematical logic
- Most abstract form of semantics

Two subcategories of semantics:

1. ***Static semantics***: How much of a program’s meaning can be determined before/without running the program?
    a. “Easier” for traditional software applications
    b. Important for static checking, e.g.
    c. “Parts of the definition that are neither part of the formal syntax, nor yet part of the runtime behavior”
2. ***Dynamic semantics***: What does the program mean as it is being run?
    a. May assume that the program is statically legal

<u>Static Semantics</u>

Depending on language: can figure out (some or all of) object types, scope

Attribute grammars: create a parse tree (as in syntax); in addition, assign attributes (e.g. types, scope) to the nonterminals that care

- **Synthesized attributes**: “Information flows up toward the root”
    - Attributes: “type of this subtree”, e.g.
    - Type deduction: from children to parents (e.g. type of “+” subtree is float if one of its children is of type float)

        - With grammar rules, may define a way to compute attributes of certain subexpressions/nonterminals from the others
            - Ex: “Expr1 = Expr2 + Expr3” -> find Expr1’s type from Expr2, Expr3
- **Inherited attributes**: information flows either direction/both ways
    - Scope: need a symbol table mapping identifier names to types
        - Declaration -> takes a symbol table from parent & builds a slightly larger/different symbol table
        - May flow downwards + left-to-right, e.g.

Dynamic Semantics

Dynamic semantics: various “disciplined” ways to perform dynamic semantics; ex:

1. **Operational semantics**
2. **Axiomatic semantics**
3. **Denotational semantics**

Operational semantics: oldest + simplest form

- To understand a P.L. L: read source code to a compiler/interpreter for L
    - In essence: “Ask somebody else” (interpreter is simpler)
    - Interpreter: written in a simpler language K that is understood
- Observe behavior of a program in L by running on interpreter (or debugger)
- Better: read source code, mentally understand source code (w/o running)

Ex (Operational semantics): Interpreter for OCaML in Prolog:

- m(E, C, R) [m/3]: takes ML expr E, context vars->values C, result of evaluating E in C R
- call/2: atoms represent variables, constants represent themselves, call(F, E) represents F E, fun(I, E) represents fun I->E, let(I, E, F) represents let I=E in F
    - m(E, _, E) :- integer(E).
    - m(E [variable], C, R) :- atom(E), member(E=R, C), !.
        - Context C: list of var=value terms
        - E an atom; member checks if E in R
    - m(let(I, E, F), C, R) :- m(E, C, V), m(F, [I=V|C], R)
        - Second m evaluates in bigger context [C with I added]
    - m(fun(I, E), C, lambda(I, C, E)).

        - Third term: local expression representing a runtime callable that, when called, evaluates to appropriate body; don't need to run the function to represent
    - m(call(lambda(I, C, E1), E), _, R) :- m(let(I, E, E1), C, R)

Lisp 1.5: defined a Lisp interpreter written in Lisp

Axiomatic semantics: write down logic (logical axioms + rules of inference) regarding how to reason about a program

- Classic definition [Pascal: {} -> comment]: {P} S {Q}
    - S a statement; P a precondition, Q a postcondition (P, Q booleans)
    - Interpretation: If P is true and S is executed (+ finishes), then Q is true afterward
    - Ex: v:=E (assignment) -> {Q [v/E] v:=E {Q} (assuming no side effects)
        - [v/E]: substitute v for every occurrence of E in Q
    - Ex: if B [boolean] then T else E -> {P} if B then T else E {Q}
        - Assuming no side effects in B: to prove, need to prove (i) {P^B} T {Q} and {P^!B} E {Q}
    - Ex: {P} while W do S {Q}: {P ^ W} S {P}, P^!W=>Q
        - Catch: no guarantee that loop will terminate -> may need additional clause
        - Verifying a loop terminates: loop condition is monotonically decreasing non-negative integer function
- More broadly: {true} Program {condition} to statically verify correctness properties
    - Don't need to run programs to verify

Denotational: semantics(P) = function from inputs to outputs

- Figure out what inputs, outputs are -> write down the function corresponding to the program P
    - Can write down function as a mathematical function -> apply mathematical theory

Analogy with 3 major classes of programming languages:

1. Operational semantics: imperative languages
2. Axiomatic semantics: logical languages
3. Denotational semantics: functional languages

(Connection between what we want our program to mean and how we write it)

(*) History of Programming Languages

First successful programming language: Fortran/FORTRAN (1956/1957)

- Traditionally written in all caps [1950s: only uppercase letters on keyboards]
- “Most successful language of all time”: within a year, used in half of software
- 1955 belief: impossible to have a programming language
- Fortran: IBM, J. Backus
    - IBM: successful due to having Fortran compiler
    - Implemented arrays, loops, subroutines
- No formal standard for Fortran [runs on IBM compiler ⇔ is Fortran]

Algol (1960) [J. Backus]

- First formal syntax via BNF

List Processing Language/Lisp (1959)

- New ideas: recursion, garbage collection
- S-expressions: representing a program as data [useful for symbolic AI]

Cobol (1960) [Grace Hopper & co.]

- Made for business: records [structs]
- Mistake: tried to make programs easier-to-read by managers (to verify correctness) -> in practice, didn’t work
    - Add 1 syntax: “ADD 1 TO N.”
    - Complex programs -> readability doesn’t scale anyway
- Very successful language

PL/I (1964)

- “Kitchen sink”: tried to come up with a single programming language that could do anything Fortran/Algol/Cobol could
    - Recent analogue: C# for Java/C++

C (1972): designed to be machine-level, simple, and connected to Unix

- Inspired by PL/I + Algol 68 [Algol successor]

- C directly tied to an OS (Unix), used to write Unix kernel [first successful example of a PL used to write a kernel]

Simula 67 (1967): descendant of Algol

C++ (1976): successor of C, Simula 67

Future: ???