

*A programming language is low level when its programs require attention to the irrelevant.*

*- Alan Perlis*

## CSE341 Programming Languages

Lecture 1.2 – September 2019

Programming Paradigms

© 2012-2019 Yakup Genç

Largely adapted from V. Shmatikov, J. Mitchell and R.W. Sebesta

## Duyuru

BIL101 ve BIL107 dersi için CSE102 ve CSE241 veya CSE244 derslerinden BB ve üstü alan tutorlara ihtiyaç var....

- Neden yapmak isteyebilirsiniz:
  - Not yükseltmek
  - Deneyim kazanmak
  - Öğreterek öğrenmek
  - ...

September 2019

CSE341 Lecture 1.2

2

## Today

- Evaluating Programming Languages
- Programming Paradigms
- Compilation Process

September 2019

CSE341 Lecture 1.2

3

## Evaluating Programming Languages

September 2019

CSE341 Lecture 1.2

4

## Language Evaluation Criteria

- Readability – easily understood?
- Writability – easy to write?
- Simplicity, orthogonality
- High expressive power, flexibility
- Reliability
- Safety
- Cost (influenced by above)
  - Creation
  - Execution
  - Maintenance

September 2019

CSE341 Lecture 1.2

5

## Language Evaluation Criteria

- Readability: Most important
  - Overall simplicity
    - Too many features is bad
    - Multiplicity of features is bad
  - Orthogonality
    - Makes the language easy to learn and read
    - Meaning is context independent
  - Control statements
  - Data type and structures
  - Syntax considerations

September 2019

CSE341 Lecture 1.2

6

## Language Evaluation Criteria

- Writability:
  - Simplicity and orthogonality
  - Support for abstraction
  - Expressivity
  - Development environments

September 2019

CSE341 Lecture 1.2

7

## Simplicity Improves Read/Writability

- A **large** language takes more time to learn
  - Programmers might learn only a subset
- **Feature multiplicity** is often confusing
  - E.g., decrementing a variable in four different ways
- **Operator overloading** can lead to confusion
- Some languages, can be "**too simple**"

September 2019

CSE341 Lecture 1.2

8

## Orthogonality

- A language is orthogonal if its features are built upon a small, mutually independent set of primitive operations.
- Fewer exceptional rules = conceptual simplicity
  - E.g., restricting types of arguments to a function
- Tradeoffs with efficiency

September 2019

CSE341 Lecture 1.2

9

## Orthogonality Improves Read/Writability

- Having fewer constructs and having few exceptions increases readability and writability
- Orthogonal languages are easier to learn
- Examples:
  - Pointers should be able to point to any type of variable or data structure
    - Exceptions (e.g. in C) are due to a lack of orthogonality
  - ADD op1 op2 op1 vs.:
  - ADDR Reg1 Reg2 Reg1 **and**
  - ADDRM Reg1 MemA Reg1
  - A **different ADD operation** depending on operand location in memory!
- However, if a language is **too orthogonal**, an inexperienced programmer might assume they can do something that makes no sense,
  - e.g. add two pointers together

September 2019

CSE341 Lecture 1.2

10

## Support for Abstraction

- Data
  - Programmer-defined types and classes
  - Class libraries
- Procedural
  - Programmer-defined functions
  - Standard function libraries

September 2019

CSE341 Lecture 1.2

11

## Reliability

- Program behavior is the same on different platforms
  - E.g., early versions of Fortran
- Type errors are detected
  - E.g., C vs. ML
- Semantic errors are properly trapped
  - E.g., C vs. C++
- Memory leaks are prevented
  - E.g., C vs. Java

September 2019

CSE341 Lecture 1.2

12

## What Does This Mean?

`*p++ = *q++`

Does this mean...

```
*p = *q;
++p;
++q;
```

... or

```
*p = *q;
++q;
++p;
```

... or

```
tp = p;
++p;
tq = q;
++q;
*tp = *tq;
```

September 2019

CSE341 Lecture 1.2

13

## Efficient Implementation

- Embedded systems
  - Real-time responsiveness (e.g., navigation)
  - Failures of early Ada implementations
- Web applications
  - Responsiveness to users (e.g., Google search)
- Corporate database applications
  - Efficient search and updating
- AI applications
  - Modeling human behaviors

September 2019

CSE341 Lecture 1.2

14

## Language Evaluation Criteria

- Reliability:
  - Type checking
  - Exception handling
  - Aliasing
  - Readability and writability

September 2019

CSE341 Lecture 1.2

15

## Language Evaluation Criteria

Characteristic	CRITERIA		
	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

September 2019

CSE341 Lecture 1.2

16

## Language Evaluation Criteria

---

- Cost:
  - Programmer training
  - Software creation
  - Compilation
  - Execution
  - Compiler cost
  - Poor reliability
  - Maintenance

September 2019

CSE341 Lecture 1.2

17

## Language Evaluation Criteria

---

- Others:
  - Portability
  - Generality
  - Well-definedness

September 2019

CSE341 Lecture 1.2

18

## Programming Paradigms

---

September 2019

CSE341 Lecture 1.2

19

## What Is a Programming Language?

---

- Formal notation for specifying computations, independent of a specific machine
  - Example: a factorial function takes a single non-negative integer argument and computes a positive integer result
  - Mathematically, written as factorial:  $\text{Int} \rightarrow \text{Int}$
- Set of imperative commands used to direct computer to do something useful
  - Print to an output device: `printf("merhaba\n");`
  - What mathematical function is "computed" by `printf`?

September 2019

CSE341 Lecture 1.2

20

## What is Computation

- Solving a problem  $\rightarrow$  function
- A decision problem: well-defined question about well-specified data that has a yes/no answer.
  - For example, primality problem

September 2019

CSE341 Lecture 1.2

21

## Partial and Total Functions

- Value of an expression may be undefined
  - Undefined operation, e.g., division by zero
    - $3/0$  has no value
    - Implementation may halt with error condition
  - Nontermination
    - $f(x) = \text{if } x=0 \text{ then } 1 \text{ else } f(x-2)$
    - This is a partial function: not defined on all arguments
    - Cannot be detected by inspecting expression (why?)
- These two cases are “mathematically” equivalent, but operationally different (why?)

September 2019

CSE341 Lecture 1.2

22

## Partial and Total Functions

- Total function  $f: A \rightarrow B$  is a subset  $f \subseteq A \times B$  with
  - $\forall x \in A$ , there is some  $y \in B$  with  $\langle x, y \rangle \in f$  (total)
  - $\langle x, y \rangle \in f$  and  $\langle x, z \rangle \in f$  then  $y = z$  (single-valued)
- Partial function  $f: A \rightarrow B$  is a subset  $f \subseteq A \times B$  with
  - If  $\langle x, y \rangle \in f$  and  $\langle x, z \rangle \in f$  then  $y = z$  (single-valued)
- Programs define partial functions for two reasons
  - What are these reasons?

September 2019

CSE341 Lecture 1.2

23

### Programs Define Partial Functions

In most programming languages, it is possible to define functions recursively. For example, here is a function  $f$  defined in terms of itself:

---

```
f(x:int) = if x = 0 then 0 else x + f(x-2);
```

---

If this were written as a program in some programming language, the declaration would associate the function name  $f$  with an algorithm that terminates on every even  $x \geq 0$ , but diverges (does not halt and return a value) if  $x$  is odd or negative. The algorithm for  $f$  defines the following mathematical function  $f$ , expressed here as a set of ordered pairs:

$$f = \{ \langle x, y \rangle \mid x \text{ is positive and even, } y = 0 + 2 + 4 + \dots + x \}.$$

This is a partial function on the integers. For every integer  $x$ , there is at most one  $y$  with  $f(x) = y$ . However, if  $x$  is an odd number, then there is no  $y$  with  $f(x) = y$ . Where the algorithm does not terminate, the value of the function is undefined. Because a function call may not terminate, this program defines a partial function.

September 2019

CSE341 Lecture 1.2

24

Computability theory establishes some important ground rules for programming language design and implementation. The following main concepts from this short overview should be remembered:

- *Partiality*: Recursively defined functions may be partial functions. They are not always total functions. A function may be partial because a basic operation is not defined on some argument or because a computation does not terminate.
- *Computability*: Some functions are computable and others are not. Programming languages can be used to define computable functions; we cannot write programs for functions that are not computable in principle.
- *Turing completeness*: All standard general-purpose programming languages give us the same class of computable functions.
- *Undecidability*: Many important properties of programs cannot be determined by any computable function. In particular, the halting problem is undecidable.

When the value of a function or the value of an expression is undefined because a basic operation such as division by zero does not make sense, a compiler or interpreter can cause the program to halt and report the error. However, the undecidability of the halting problem implies that there is no way to detect and report an error whenever a program is not going to halt.

September 2019

CSE341 Lecture 1.2

25

## Computability

- Function  $f$  is computable if some program  $P$  computes it
  - For any input  $x$ , the computation  $P(x)$  halts with output  $f(x)$
  - Partial recursive functions: partial functions (int to int) that are computable

September 2019

CSE341 Lecture 1.2

26

## Halting Function

- Decide whether program halts on input
  - Given program  $P$  and input  $x$  to  $P$ ,  
 $\text{Halt}(P,x) = \text{yes, if } P(x) \text{ halts}$   
 $\text{no, otherwise}$
- Clarifications
  - Assume program  $P$  requires one string input  $x$
  - Write  $P(x)$  for output of  $P$  when run in input  $x$
  - Program  $P$  is a string input to Halt
- Fact: There is no program for Halt

September 2019

CSE341 Lecture 1.2

27

## Unsolvability of the Halting Problem

- Suppose  $P$  solves variant of halting problem  
 $\text{On input } Q, \text{ assume } P(Q) = \text{yes, if } Q(Q) \text{ halts}$   
 $\text{no, otherwise}$
- Build program  $D$   
 $D(Q) = \text{run forever if } Q(Q) \text{ halts}$   
 $\text{halt if } Q(Q) \text{ runs forever}$
- If  $D(D)$  halts, then  $D(D)$  runs forever
- If  $D(D)$  runs forever, then  $D(D)$  halts
- Contradiction! Thus  $P$  cannot exist.

September 2019

CSE341 Lecture 1.2

28

## Main Points About Computability

- Some functions are computable, some are not
  - Example: halting problem
- Programming language implementation
  - Can report error if program result is undefined due to an undefined basic operation (e.g., division by zero)
  - Cannot report error if program will not terminate

September 2019

CSE341 Lecture 1.2

29

## Computation Rules

- The factorial function type declaration does not convey how the computation is to proceed
- We also need a computation rule
 
$$\text{fact}(0) = 1$$

$$\text{fact}(n) = n * \text{fact}(n-1)$$
- This notation is more computationally oriented and can almost be executed by a machine

September 2019

CSE341 Lecture 1.2

30

## Factorial Functions

- C, C++, Java:
  - `int fact (int n) { return (n == 0) ? 1 : n * fact (n-1); }`
- Scheme:
  - `(define fact (lambda (n) (if (= n 0) 1 (* n (fact (- n 1))))))`
- Haskell:
  - `fact :: Integer->Integer`
  - `fact 0 = 1`
  - `fact n = n*fact(n-1)`

September 2019

CSE341 Lecture 1.2

31

## Paradigms

September 2019

CSE341 Lecture 1.2

32



## Paradigms

- It describes an overall concept accepted by most people in an intellectual community, as those in one of the natural sciences, because of its effectiveness in explaining a complex process, idea, or set of data
- A scientific paradigm is a framework containing all the commonly accepted views about a subject, conventions about what direction research should take and how it should be performed

September 2019

CSE341 Lecture 1.2

33

## Paradigms

- Imperative
  - Procedural
  - Object-Oriented
- Declarative
  - Functional / Applicative
  - Logic
  - Mathematical
- In reality, very few languages are “pure”
  - Most combine features of different paradigms

September 2019

CSE341 Lecture 1.2

34

## Why Paradigms?

- Paradigms emerge as the result of social processes in which people develop ideas and create principles and practices that embody those ideas
  - Thomas Kuhn. “The Structure of Scientific Revolutions.”
- Programming paradigms are the result of people’s ideas about how programs should be constructed
  - ... and formal linguistic mechanisms for expressing them
  - ... and software engineering principles and practices for using the resulting programming language to solve problems

September 2019

CSE341 Lecture 1.2

35

## Imperative

- Imperative (procedural) programs consists of actions to effect **state change**, principally through assignment operations or side effects
  - Fortran, Algol, Cobol, PL/I, Pascal, Modula-2, Ada, C
  - Why does imperative programming dominate in practice?
- OO programming is not always imperative, but most OO languages have been imperative
  - Simula, Smalltalk, C++, Modula-3, Java
  - Notable exception: CLOS (Common Lisp Object System)

September 2019

CSE341 Lecture 1.2

36

## Functional and Logic

- Focuses on function evaluation; avoids updates, assignment, mutable state, side effects
- Not all functional languages are “pure”
  - In practice, rely on non-pure functions for input/output and some permit assignment-like operators
  - E.g., (set! x 1) in Scheme
- Logic programming is based on predicate logic
  - Targeted at theorem-proving languages, automated reasoning, database applications
  - Declarative programming (express the logic of a computation without describing its control flow)

September 2019

CSE341 Lecture 1.2

37

## Concurrent and Scripting

- Concurrent programming cuts across imperative, object-oriented, and functional paradigms
- Scripting is a very “high” level of programming
  - Rapid development; glue together different programs
  - Often dynamically typed, with only int, float, string, and array as the data types; no user-defined types
  - Weakly typed: a variable ‘x’ can be assigned a value of any type at any time during execution
- Very popular in Web development
  - Especially scripting active Web pages

September 2019

CSE341 Lecture 1.2

38

## Unifying Concepts

- Unifying language concepts
  - Types (both built-in and user-defined)
    - Specify constraints on functions and data
    - Static vs. dynamic typing
  - Expressions (e.g., arithmetic, boolean, strings)
  - Functions/procedures
  - Commands
- We will study how these are defined syntactically, used semantically, and implemented pragmatically

September 2019

CSE341 Lecture 1.2

39

## Design Choices

- C: Efficient imperative programming with static types
- C++: Object-oriented programming with static types and ad hoc, subtype and parametric polymorphism
- Java: Imperative, object-oriented, and concurrent programming with static types and garbage collection
- Scheme: Lexically scoped, applicative-style recursive programming with dynamic types
- Standard ML: Practical functional programming with strict (eager) evaluation and polymorphic type inference
- Haskell: Pure functional programming with non-strict (lazy) evaluation.

September 2019

CSE341 Lecture 1.2

40

## Abstraction and Modularization

- Re-use, sharing, extension of code are critically important in software engineering
- Big idea: **detect errors at compile-time**, not when program is executed
- **Type** definitions and declarations
  - Define intent for both functions/procedures and data
- **Abstract data types (ADT)**
  - Access to local data only via a well-defined interface
- Lexical **scope**

September 2019

CSE341 Lecture 1.2

41

## Static vs. Dynamic Typing

- Static typing
  - Common in compiled languages, considered “safer”
  - Type of each variable determined at compile-time; constrains the set of values it can hold at run-time
- Dynamic typing
  - Common in interpreted languages
  - Types are associated with a variable at run-time; may change dynamically to conform to the type of the value currently referenced by the variable
  - Type errors not detected until a piece of code is executed

September 2019

CSE341 Lecture 1.2

42

## Why is this so Important?



- Failed launch of Ariane 5 rocket (1996)
  - \$500 million payload; \$7 billion spent on development
- Cause: software error in inertial reference system
  - Re-used Ariane 4 code, but flight path was different
  - 64-bit floating point number related to horizontal velocity converted to 16-bit signed integer; the number was larger than 32,767; inertial guidance crashed

September 2019

CSE341 Lecture 1.2

43

## Program Correctness

- Assert formal correctness statements about critical parts of a program and reason effectively
  - A program is intended to carry out a specific computation, but a programmer can fail to adequately address all data value ranges, input conditions, system resource constraints, memory limitations, etc.
- Language features and their interaction should be clearly specified and understandable
  - If you do not or can not clearly understand the semantics of the language, your ability to accurately predict the behavior of your program is limited

September 2019

CSE341 Lecture 1.2

44

## Language Translation

- **Native-code compiler:** produces machine code
  - Compiled languages: Fortran, C, C++, SML ...
- **Interpreter:** translates into internal form and immediately executes (read-eval-print loop)
  - Interpreted languages: Scheme, Haskell, Python ...
- **Byte-code compiler:** produces portable bytecode, which is executed on virtual machine (e.g., Java)
- Hybrid approaches
  - Source-to-source translation (early C++ to C to compile)
  - Just-in-time Java compilers convert bytecode into native machine code when first executed

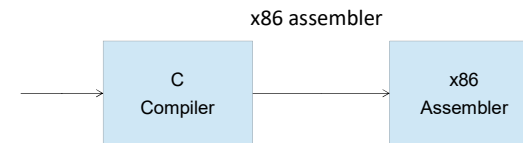
September 2019

CSE341 Lecture 1.2

45

## Language Compilation

- **Compiler:** program that translates a source language into a target language
  - Target language is often, but not always, the assembly language for a particular machine



September 2019

CSE341 Lecture 1.2

46

## Checks During Compilation

- Syntactically invalid constructs
  - Invalid type conversions
  - A value is used in the “wrong” context, e.g., assigning a float to an int
- Static determination of type information is also used to generate more efficient code
  - Know what kind of values will be stored in a given memory region during program execution
- Some programmer logic errors
  - Can be subtle: if (a = b) ... instead of if (a == b) ...

September 2019

CSE341 Lecture 1.2

47

## Compilation Process

- Compilation: source code → relocatable object code (binaries)
- Linking: many relocatable binaries (modules plus libraries) → one relocatable binary (with all external references satisfied)
- Loading: relocatable → absolute binary (with all code and data references bound to the addresses occupied in memory)
- Execution: control is transferred to the first instruction of the program

September 2019

CSE341 Lecture 1.2

48

## Compilation Process

- At compile time, absolute addresses of variables and statement labels are not known
- In static languages (such as Fortran), absolute addresses are bound at load time
- In block-structured languages, bindings can change at run time

September 2019

CSE341 Lecture 1.2

49

## Phases of Compilation

- Preprocessing: conditional macro text substitution
- Lexical analysis: convert keywords, identifiers, constants into a sequence of tokens
- Syntactic analysis: check that token sequence is syntactically correct
- Generate abstract syntax trees (AST), check types
- Intermediate code generation: “walk” the ASTs and generate intermediate code
  - Apply optimizations to produce efficient code
- Final code generation: produce machine code

September 2019

CSE341 Lecture 1.2

50

## Language Interpretation

- Read-eval-print loop
  - Read in an expression, translate into internal form
  - Evaluate internal form
    - This requires an abstract machine and a “run-time” component (usually a compiled program that runs on the native machine)
  - Print the result of evaluation
  - Loop back to read the next expression

September 2019

CSE341 Lecture 1.2

51

## Bytecode Compilation

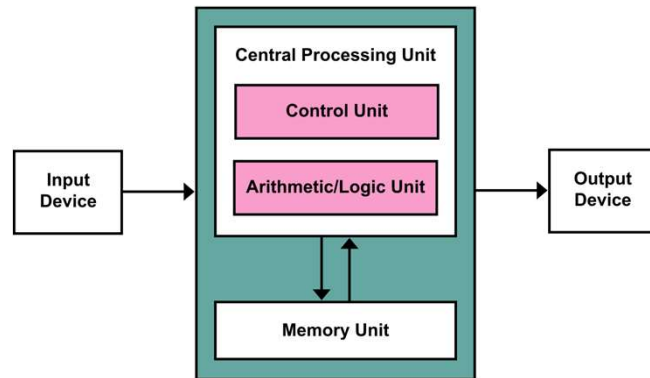
- Combine compilation with interpretation
  - Idea: remove inefficiencies of read-eval-print loop
- Bytecodes are conceptually similar to real machine opcodes, but they represent compiled instructions to a virtual machine instead of a real machine
  - Source code statically compiled into a set of bytecodes
  - Bytecode interpreter implements the virtual machine
  - In what way are bytecodes “better” than real opcodes?

September 2019

CSE341 Lecture 1.2

52

## Von Neumann Model

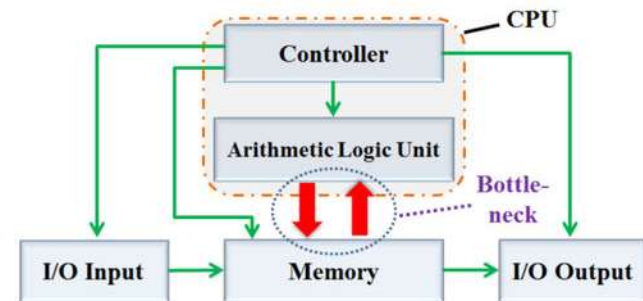


September 2019

CSE341 Lecture 1.2

53

## Von Neumann Bottleneck



September 2019

CSE341 Lecture 1.2

54

## Von Neumann Programming Languages

- PLs that are high-level abstract isomorphic copies of von Neumann architectures
- The isomorphism between von Neumann programming languages and architectures is in the following manner:
  - program variables  $\leftrightarrow$  computer storage cells
  - control statements  $\leftrightarrow$  computer test-and-jump instructions
  - assignment statements  $\leftrightarrow$  fetching, storing instructions
  - expressions  $\leftrightarrow$  memory reference and arithmetic instructions

September 2019

CSE341 Lecture 1.2

55