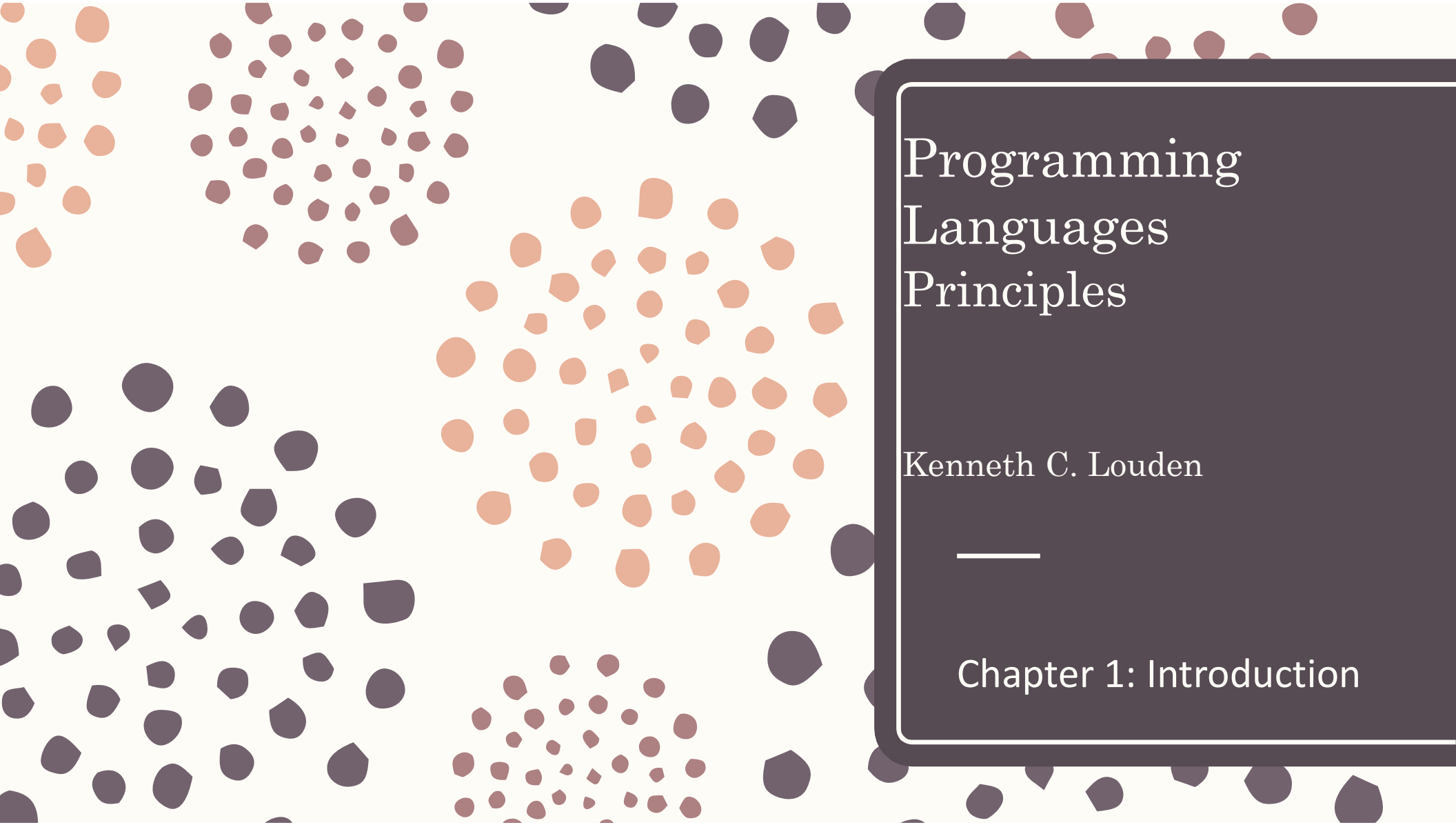

Programming
Languages
Principles
Kenneth C. Louden
Chapter 1: Introduction

1.1 What is a programming language?

Machine-readability

1. Language must have simple enough structure for efficient translation (into what?)
2. There must be an algorithm (what is that?) to translate the language. The algorithm must be unambiguous and finite (what does this mean?)
   The algorithm cannot be too complex. Otherwise it would take more time to translate the program than to execute it.
3. The programming language is restricted to something called a Context-Free language (i.e. no interdependencies in it)

1.1 What is a programming language?

Human-readability

1. The language must provide abstractions of the actions the computer must take.
2. These abstractions must be fairly easy to understand by humans.
3. The language must keep humans as far as possible from the internal workings of the computer.
4. Readability also involves the ease with which large programs can be written. If we need to fix a bug, how easy is it to find, and how easy is it to fix? (localization of data and functions)
5. Also a language must make it easy for teams of programmers to program and communicate with each other (Software Development and Software Engineering)

1.2 Abstractions in programming languages

Basically two types

- Data abstractions
  - Properties of data
- Control abstractions
  - Modification of execution path
- Abstractions also have levels
  - Basic –measures of the amount of info in the abstraction
  - Structured – more global info about program structure
  - Unit – collect info about entire pieces of a program

1.2.1 Data Abstractions

Basic Abstractions

- Abstraction of the internal data representation of common data
- Simple example is the data variable
  - int my_age;
  - float cost;
  - var x: integer; (this is in Pascal)
- The programmer does not have to know the internal bit-structure for these basic data values, only the abstract method of defining them.
- Variables are given names and a data “type”

1.2.1 Data Abstractions

Structured Abstractions

- Abstraction of a collection of data (a “data structure”)
- For example, a simple array of integers, or a record of data

```
int a[10]; in Java
INTEGER A(10) (in FORTRAN)
typedef int IntArray[10] (in C)
struct {
        char name[20];
        int age;
        float salary;
} employee;          (in C)
```

1.2.1 Data Abstractions

Unit Abstractions

- Abstraction of a collection of data PLUS related code
- A good example of this would be a “class” in Java
- Uses “data encapsulation” and “information hiding”
- Unit abstractions should be developed so that they are “reusable” in other programs.
- Unit abstractions can be used to form a “library” that others can use when programming.
- Must also have standard interface definitions to improve the “interoperability” between programming modules.

1.2.2 Control Abstractions

Basic Abstractions

1. Combining machine instructions into a more understandable form.
2. The “assignment statement” e.g. x = x + 3;
3. What is involved?
4. The “GO TO” statement. Not found in several higher level languages.
5. GO TO is the same as an internal JMP instruction.

1.2.2 Control Abstractions

Structured Abstractions

1. Divide the program into sets of instructions:

```
 if (x>0)
{       r1=sqrt(x);
        r2= - r1;
}else
{       numSolns = 0;
        r1 = r2 = 0;
}
```

2. Other examples include switch-case statements
3. Review examples in text
4. Also includes loops (for, while, repeat, function, method, subroutine)

1.2.2 Control Abstractions

Unit Abstractions

1. A collection of functions, methods or subroutines that are related in some way.
2. Can be translated separately, so that programmers do not need to know the details of how things are done.
3. Examples: java.applet.Applet, math.h

Other abstractions

1. For parallel processing

1.3 Computational Paradigms

1. A paradigm – what is it?
2. Languages were developed for specific computer architectures (e.g. sequential processing)
3. An “imperative language”
    1. *Sequential execution of instructions*
    2. *Variables representing memory locations*
    3. *Assignment to change values of variables*
4. Most languages are like this, but there are other paradigms – Object-oriented, Functional, Logic

1.3 Computational Paradigms

Object-oriented Programming

1. Based on “object” = memory locations + operations on them.
2. Each object is sort of like its own computer.
3. Objects are grouped into classes – having same properties
4. An object is an “instance” of a class.
5. Next page gives example of a “GCD” operation, but defined in a class.

1.3 Computational Paradigms

```
public class IntWithGcd
{ public IntWithGcd( int val) { value = val; }
  public int intValue( ) { return value; }
  public int gcd( int v )
  {  int z = value;
     int y  = v;
     while( y != 0)
     { int t = y;
       y = z % y;
       z = t;
      }
     return z;
  }
  private int value;
}
```

1.3 Computational Paradigms

1. What things are defined in the previous class?
2. Why does the gcd function only take one parameter?
3. Why is the “value” variable “private”?
4. How would you use this class in an actual Java program?
5. How else might the same things be accomplished in an imperative language?

1.3 Computational Paradigms

Functional Programming

1. The functional paradigm has evaluation of functions as its description of computation.
2. Also called “applicative” language.
3. Does not use much variable definition or assignment, except what a function may need to do its job.
4. Based on “passed” parameters and “returned” values”
5. Repetitive operations are based on function recursion, rather than on looping mechanisms.
6. Doing away with variables and loops makes the language
    1. More independent of the machine model
    2. More like mathematics and can make decisions on behavior better

1.3 Computational Paradigms

An example in ADA (by the way, what is ADA?)

```
function gcd (u, v: in integer) return integer is
begin
  if v = 0 then
     return u;
  else
     return gcd (v, u mod v);
  end if;
end gcd;
```

Where are the local variables? Where are the loops?

1.3 Computational Paradigms

Another even weirder example from LISP

```
(define (gcd u v) (if (= v 0) u (gcd v (modulo u v))))
```

1. What are the lists?
2. Is anything returned?
3. How does this thing work at all?
4. How would you call this function in LISP?

1.3 Computational Paradigms

Logic programming

1. Based on symbolic logic.
2. Statements are given about what is true of an object
3. No need for control abstraction such as loops
4. Control is supplied by the underlying system
5. Often called “declarative programming”
6. “Very High Level” languages – e.g. Prolog
    1. gcd of u and v is u if v = 0;
    2. gcd of u and v is same as gcd of v and u mod v, if v is not 0

Computational Paradigms

In Prolog

```
gcd(U, V, U) :- V= 0.
gcd(U, V, X) :- not (V = 0),
                Y = U mod V,
                gcd(U. Y, X).
```

a :- b, c, d means a is true if b, c, and d are true.

How would you “read” the above code?

A final word – languages generally do not follow only one paradigm exclusively. There is a mix-match to make things as easy as possible for the programmers.

Language Definition

1. A programming language needs a “precise” definition
2. We must know what each construction in a language will do so that we know what the computer will do.
3. A precise definition also helps make the language machine independent. (Follow standards set my other organizations such as ANSI or ISO.)
4. Programmers will need to know how programs interact; this also implies that a good definition is required.
5. With good definition, the program design phase of software development goes much easier.

Language Definition

Language Syntax

1. Syntax of a language is like the grammar of a regular language.
2. Describes how parts of the language can be put together to form other parts.
3. Usually uses some sort of rules.
4. <if-statement> ::= if(<expression>) <statement>
   [else <statement>]
5. Also important is the “lexical structure”. This is the structure of the words of the language. Each word is a “token”. For example “if” and “else” are tokens.
6. Syntax also defines punctuation, when to use “;”, etc.

1.4 Language Definition

Language Semantics

1. This is much more complex and difficult to define.
2. Basically semantics deals with the “meaning” of statements in the language.
3. “IF YOU DO THIS” then “THE COMPUTER WILL DO THAT”
4. Example: An if-statement is executed by first evaluating its expression, which must have arithmetic or pointer type, including all side effects, and if it compares unequal to 0, the statement following the expression is executed. If there is an else part, and the expression evaluates to 0, that statement following the “else” is executed.
5. But even the above semantic definition has problems. (What?)

Operational Semantics

- Definitional interpreters or compilers
- Specifies how an arbitrary program would be executed on a machine whose operation is completely unknown
- It defines the behavior of programs in terms of an abstract machine that is simple enough to be completely understood and simulated by any user
  - Answers questions about program behavior
- Reduction machine is a collection of steps in reducing programs by applying their operations to values

Denotational Semantics

- Uses functions to describe the semantics of a programming language
- Programs are converted to mathematical functions
- A function describes semantics by associating values to syntactically correct constructs
- Syntactic Domain
  - E: Expression
  - N: Number
  - D: Digit
  - E→ E_1 '+' E_2
- Semantic domains
  - Domain v: Integer={….,-1,0,1,2,…}
  - +: Integer x Integer → Integer
- Semantic functions
  - E:Expression→Integer

Axiomatic Semantics

- A program or statement or language construct
  - Describes the effect its execution has on assertions about the data manipulated by the program
  - Elements of mathematical logic are used here to specify the semantics

Assertions

```
class TestingAxioms {

        public static void main(String args[]) {
                Scanner sc=new Scanner(System.in);
                System.out.println("Enter age:");
                int value=sc.nextInt();
                assertValue>=18:"valid";
                System.out.println("Value is " + value);
        }
}

Java –ea <>
```

Properties of Formal Semantics Specification

- Complete
  - Every correct terminating program must have an associated semantics given by the rules
- Consistent
  - Same program cannot be given two different conflicting semantics
- Independent
  - No rule should be derivable from other rules

1.4 Language Translation

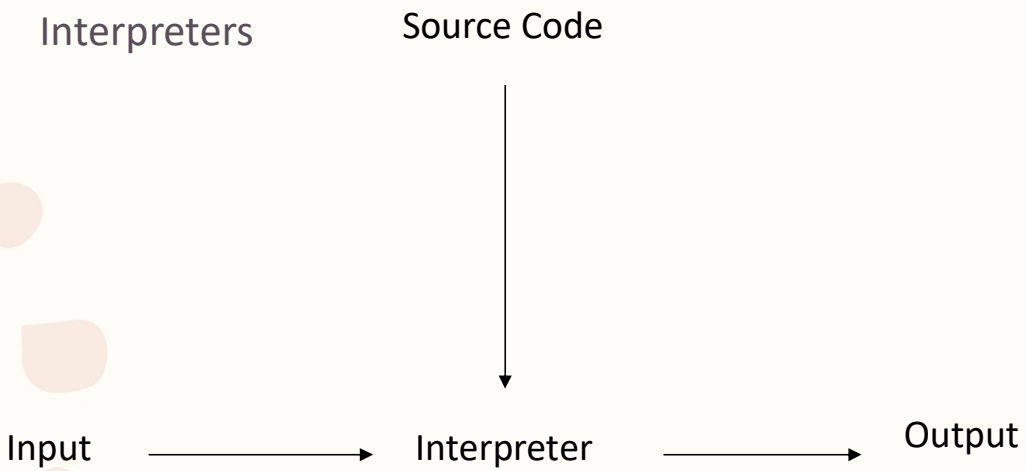
1. All programming languages must have a “translator”
2. A translator is another program that accepts programs written in the language and that

   a) executes the program directly or

   b) transforms them into a form suitable for execution at a later time.
3. Interpreter = direct execution
4. Compiler = creates file for later execution

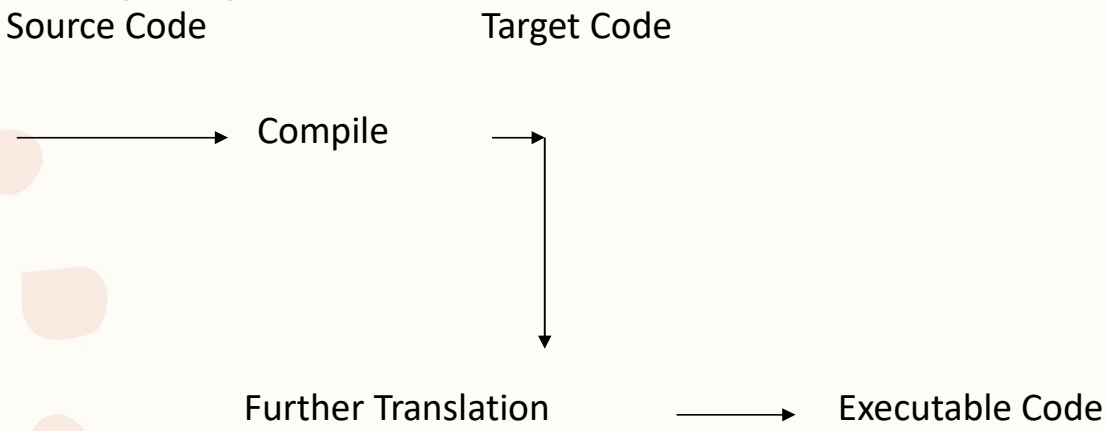
Language Translation

Interpreters



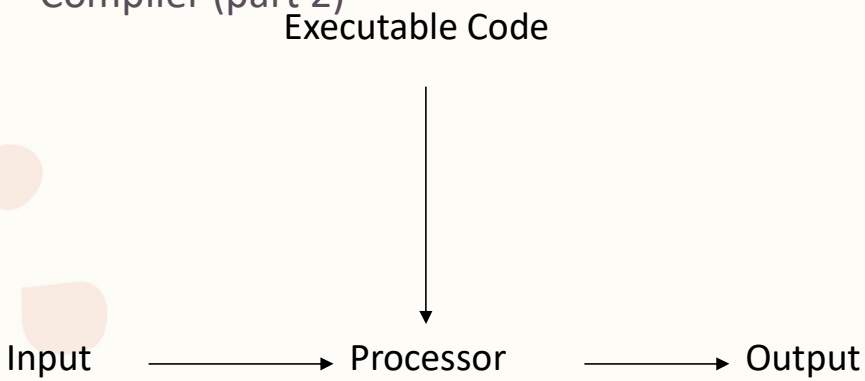
Language Translation

Compiler (part 1)



Language Translation

Compiler (part 2)



Language Translation

1. Be careful! Some compilers do not translate source code directly into machine language.
2. Some compilers produce “intermediate code” that is then interpreted by another program
3. Java is like this. JAVAC creates a .class file, which is then interpreted by the JVM (Java Virtual Machine)

Language Translation

All language translators must do similar operations:

1. Lexical Analyzer (scanner) – reads text and separates text into sequences of characters, based on punctuation – creates tokens
2. Syntax Analyzer (parser) – determines structure of the tokens and syntactic correctness of the source code
3. Semantic Analyzer – determines the meaning of the program and how to create the target code
4. These phases are not separate; they tend to go be done back-and-forth until the program is translated.
5. Also must maintain a “runtime environment” that contains space for variables, data structures, code, etc.

Discussion Topics

- Difference between data structure and abstract data types
- An abstraction allows programmers to say more with less in their code. Discuss about its pros and cons with examples