Week 2

Patterns

- Patterns are crucial and provide significant flexibility in programming.
- Pattern matching is fundamental to *reactive programming*:
    - It involves *sensing events* or *recognizing expressions*
    - Reactions are triggered based on matched patterns
- The described concept focuses on matching expressions rather than external events.
- When a pattern is recognized, a new expression is returned as a response.
- The process can be described as a function:
    - Takes an input
    - Returns an output based on pattern matching
- Typically, there are multiple possible paths or patterns to match against.
- This function structure is like a "push" operation in certain programming paradigms.
- The overall concept seems to be describing a core mechanism of pattern matching and reactive systems in programming.

Example:

```
val recip: Partial function [Double, Double],
could be re-written as Function [Double, Double] is equivalent to Double => Double
```

syntactic sugar -> writing something in an easier and preferable way

- Syntactic sugar is a programming concept that allows for writing code in an easier and preferable way.
- The text discusses equivalent ways of writing certain programming constructs, with one being more concise (the "sugar").
- There's a trade-off between reusability and simplicity:
    - One method allows for installation and reuse in multiple places.
    - The simpler, "sugared" syntax is more likely to be used when the construct is only needed in one place.
- The text introduces a specific syntax using a "rocket symbol" (=>) to indicate pattern matching and expression evaluation.
- Pattern recognition is tied to the right-hand side of this rocket symbol.
- The concept is then extended to include conditional (if-else) statements:
    - This allows for Boolean conditions in the pattern matching.
    - It introduces a structure like: "if [condition] then [expression1] else [expression2]"
- The text hints at a flexible system where various types (Boolean, expressions) can be used within these patterns.
- There's an implication that this syntax allows for more expressive and possibly more readable code in certain situations.

Type, conditions, statements

- Type Inference in Conditional Statements:
  - When assigning a value based on a condition (if-else), the compiler needs to infer the type of the result.
  - The inferred type must be a supertype of both possible outcomes (X and Y).
- Type Consistency in Conditionals:
  - In a statement like "val z = if (b) then x else y", x and y must share the same supertype.
- Evolution of Switch Statements in Java:
  - Java 14 introduced a "smart switch" with enhanced pattern matching capabilities. -> This is an improvement over the older style switch statements.
- Limitations of Traditional Switch Statements:
  - In older versions of Java, switch statements could only operate on integers. -> The new style allows switching on Booleans and potentially other types.
- Desire for More Flexible Pattern Matching:
  - There's a need to match on more than just Booleans and integers. -> The goal is to match on more complex structures.
- Advanced Pattern Matching Concepts:
  - Introduce the idea of matching on "partial defined functions" or "compound objects".
  - This allows for matching on multiple characteristics of an object independently.
- Shift from Statements to Expressions:
  - The new style of switch is moving towards being an expression rather than just a statement, allowing for more flexible use in assignments and returns.
- Programming Language Evolution:
  - The discussion highlights how programming languages evolve to provide more powerful and flexible constructs to developers.

```
Case Class complex (r: double, i:double)
Val c =  ???
c match {
case Complex(a, 0) -> real (a)
case Complex(0, b) -> imagin (b)
case _ => doComplex(a,b)
}
If putting an if condition in case Complex(a,0), like a>0 => pos_real(a) else neg_real(a)
```

Effects

- Purity in Functional Programming:
  - There's a strong emphasis on pure functions and expressions in certain programming paradigms.
  - Pure functions only depend on their inputs and don't interact with the outside world.
- Limitations of Pure Computations:
  - While pure computations can be powerful (e.g., calculating π to high precision), they become limited if their results can't be shared or used externally.
- Definition of "Outside World":
  - The "outside world" includes anything not directly part of the current thread of execution.
  - This can include I/O operations, system clock, console output, network communications, etc.
- Introduction of Effects:
  - An "effect" occurs when a program interacts with the outside world in any way.
  - Effects are essentially anything that's not a pure expression.
- Examples of Effects:
  - Printing a value to the console
  - Sending data to a server
  - Reading the system time
- IO Monad in Haskell:
  - Haskell uses the IO monad to handle interactions with the outside world.
  - This allows Haskell to maintain code purity while still enabling necessary side effects.
- Predictability and Correctness:
  - Using structured approaches to handle effects (like IO in Haskell) can make program output more predictable.
  - This predictability can help in proving the correctness of programs.
- Balancing Purity and Practicality:
  - While pure computations are desirable for their predictability and ease of reasoning, practical programs often need to interact with the outside world.
  - Languages and paradigms that emphasize purity (like Haskell) have developed ways to manage this balance.
- Importance of Controlled Side Effects:
  - Even in functional programming, controlled mechanisms for side effects are necessary for creating useful programs.

Lazy Evaluation

- Definition:
  - Lazy evaluation is a strategy where expressions are not evaluated until their results are actually needed.
- In Scala, it's implemented using the 'lazy' keyword.
  - Syntax: to make a val

```
lazy: lazy val x = expensiveComputation()
```

  - Key characteristics:
    - The expression is only evaluated once, when it's first accessed.
    - Subsequent accesses return the cached result.
    - Lazy vals must be fields of objects; they can't be local variables.
  - Benefits:
    - Improved performance by avoiding unnecessary computations.
    - Enables working with potentially infinite data structures.

        - Helps in managing side effects.
        - Useful for implementing complex algorithms efficiently.
    - Common use cases:
        - Deferring expensive computations until needed.
        - Implementing circular dependencies.
        - Creating infinite sequences or streams.
        - Optimizing I/O operations.
    - Comparison with eager evaluation:
        - Eager evaluation (the default in Scala) computes values immediately.
        - Lazy evaluation delays computation until the value is needed.
    - Potential drawbacks:
        - Can make code harder to reason about due to delayed execution.
        - May introduce unexpected performance characteristics.
        - Not thread-safe by default; requires additional synchronization for concurrent access.
    - Interaction with other Scala features:
        - Works well with Scala's immutable data structures.
        - Can be combined with traits for modular, efficient code.
        - Often used in conjunction with Scala's Stream class for lazy sequences.
    - Best practices:
        - Use lazy evaluation judiciously, where it provides clear benefits.
        - Document lazy vals clearly to avoid confusion.
        - Be aware of potential memory leaks with long-lived lazy vals.

Design a language for Big-data processing

- Importance of Lazy Evaluation:
    - The speaker is particularly excited about lazy evaluation in programming languages.
    - This suggests lazy evaluation is a powerful and interesting concept in language design.
- Cross-Platform Compatibility:
    - There's a focus on how to make a language run on various machines.
    - This was a challenge before Java, with different runtime implementations for different microprocessors.
- Java Virtual Machine (JVM) as a Target:
    - The JVM became a popular target for language implementation, especially for big data applications.
    - It provides a level of abstraction from the underlying hardware.
    - Components of Java:
        - Java Language: Used to write programs, compiled into bytecode.
        - Java Virtual Machine: Runs the bytecode, interpreting it into machine instructions.
        - Java Runtime Library: Provides standard classes and functions.
    - JVM Characteristics:
        - Described as primitive but effective.
        - Handles basic types (int, double, char, byte) and object-oriented concepts.
        - Can load new classes dynamically.
        - Language Agnosticism of JVM:
            - JVM doesn't care which language produced the bytecode.
            - This allows multiple languages (Scala, Groovy, etc.) to target the JVM.
        - Java Runtime Library:
            - Contains standard classes like String.
            - Separate from the JVM itself but crucial for Java programs.
        - Ubiquity of JVM:

        - JVM is described as being "pretty much everywhere," indicating its wide adoption.
- Evolution of Platform Independence:
    - The text suggests a shift from platform-specific implementations to more portable solutions like the JVM.

Object-oriented

- Types as Behavior Definers:
    - Types are introduced to define the behavior of objects in programming.
    - This concept is linked to mathematical sets (Z, Q, etc.) in number theory.
- Limited Operations in Basic Number Systems:
    - Basic counting numbers have limited operations (successor and predecessor).
    - More complex number systems allow for more operations.
- Practical Implementation of Number Types:
    - Programming languages use specific types like 'int' (64-bit integer) for practical applications.
    - These types have defined ranges and operations.
- Cryptography and Big Numbers:
    - Standard integer types are often insufficient for cryptography.
    - Languages provide "BigInt" or "Big Integer" for handling very large numbers.
    - Modular arithmetic is crucial in cryptography, altering typical integer operations.
- Types Define Behavior, Range, and Legal Values:
    - Types specify what operations can be performed on objects.
    - They also define the range and legal values for data.
- Simple Types Examples:
    - Boolean: Has two legal values (true, false).
    - Unit: Has only one legal value, often used in functional programming.
- Parametric Classes in Object-Oriented Programming:
    - Allow definition of generic behaviors (e.g., List<T>).
    - Enable creation of flexible, reusable code structures.
- Type Constraints in Parametric Classes:
    - Restrictions can be placed on generic types (e.g., requiring elements to be sortable).
- Instances vs. Classes:
    - The discussion shifts from talking about parametric classes to instances of classes.
    - This highlights the difference between class definitions and object instantiations.
- Evolution of Type Systems:
    - The text suggests an ongoing development in how types are conceptualized and implemented in programming languages.

Example:

```
New Boolean(true)
New Boolean(false)
```

- Class Definition and Limitations:
    - A class is typically defined as having the possibility of different members or instances.
    - In the case of Boolean, there are only two possible values (true and false), which doesn't fit the typical class definition.
- Introduction of Singleton Concept:
    - When there's only one possible member (or a very limited set like true/false), it's treated as a Singleton class.
- Object Terminology:

    - In the language being designed, an instance of a Singleton class is referred to as an "object".
    - This differs from Java's definition of an object and is also distinct from Scala's approach.
- Non-Parametric Nature of Singletons:
    - These objects (Singletons) are described as not having any parametric value.
    - This suggests a fundamental difference in how they're treated compared to regular classes.
- Balancing Theory and Practice:
    - There's an implicit balance being struck between theoretical purity (treating Booleans as special cases) and practical implementation.
- Importance of Precise Terminology:
    - The careful distinction between terms like "class," "object," and "singleton" highlights the importance of precise language in programming language design.

Subtype or Supertypes

- Subtyping and Inheritance:
    - The concept of subtypes and supertypes is introduced, showing a hierarchical relationship between types.
    - Example: Counting numbers as a supertype, with prime numbers and composite numbers as subtypes.
- Operation Specificity:
    - Different types can have specific operations that may or may not make sense for their subtypes.
    - Example: Factorization makes sense for composite numbers but not for prime numbers.
- Liskov Substitution Principle:
    - This principle is implicitly described: a subtype can be used wherever its supertype is expected.
    - Example: Using a prime number in an expression that expects a counting number.
- Type Compatibility:
    - Subtypes inherit properties and behaviors from their supertypes, ensuring compatibility in operations.
    - This allows for flexible use of subtypes in expressions designed for supertypes.
- Object-Oriented Concepts in Functional Programming:
    - The speaker suggests that object-oriented concepts can be beneficial even in functional programming languages.
- Practical Considerations in Language Design:
    - The use of object-oriented concepts is justified by their practical value, especially considering the Java Virtual Machine's architecture.
- Behavioral Consistency:
    - The importance of maintaining consistent behavior when substituting subtypes for supertypes is emphasized.
- Type System Flexibility:
    - The discussion implies that a well-designed type system can provide both rigorous type checking and flexibility in code usage.
- Integration of Mathematical Concepts:
    - Mathematical concepts (like prime and composite numbers) are used to illustrate programming language design principles.
- Balance Between Theory and Implementation:
    - The insights show a balance between theoretical principles of type systems and practical implementation considerations.

Expressions

- Precedence Rules in Operations:
  - Importance of defining clear precedence rules for operations, especially when dealing with different types or complex structures.
- Commutativity in Operations:
  - Some operations like addition (+) are commutative (x + y = y + x) for certain types, but this doesn't apply universally.
- Method Invocation Syntax:
  - Traditional object-oriented syntax (e.g., x.plus(y)) can be counterintuitive in some cases, especially with lists.
- List Operations and Left-Handedness:
  - Lists are typically "left-handed" in many programming languages, meaning operations like adding or removing elements are often performed at the head (left side) of the list.
- Scala's Syntactic Innovation:
  - Scala introduces a special syntax using colons (:) to change the precedence and visual representation of method calls. -> This allows for more intuitive representation of operations, especially for list manipulations.
- Colon Binding Rule:
  - In Scala, methods with a colon at the end bind more closely to the object on their right, inverting the usual left-to-right method call syntax.
- Visual vs. Logical Representation:
  - There's a distinction between how code looks visually to programmers and how it's interpreted by the compiler.
- Prepending to Lists:
  - The common operation of adding elements to the front of a list and how language syntax can make this more or less intuitive.
- Language Design Considerations:
  - The text illustrates how programming language designers must balance intuitive syntax for developers with logical consistency for compilers.
- Custom Operator Definitions:
  - The ability to define custom operators (like +:) allows for more expressive and domain-specific code.
- Importance of Syntax in Readability:
  - The discussion underscores how syntax choices can significantly impact code readability and intuitiveness.
- Flexibility in Method Invocation:
  - Scala's approach demonstrates how a language can provide flexibility in method invocation syntax to suit different programming paradigms or preferences.

```
Def sqr(x:Int) = x * x
Val y = sqr(x)
Val y = x*x
```

- Function Definition and Substitution:
  - The concept of function definition is explained as a substitution rule.
    → In practice, it may not be implemented exactly this way, but it's a useful mental model.
- Program Reduction through Substitution:
  - Complex programs can theoretically be reduced to basic expressions through repeated substitution.
- Lambda Calculus and Function Syntax:
  - Example: x => x * 2 as a function to multiply a value by 2.
- Expression Extraction for Readability:
  - Complex expressions can be made more readable by extracting parts and giving them meaningful names.
    → This process is the inverse of substitution and helps in understanding complex logic.
- Code Readability vs. Compiler Understanding:

    - While compilers can easily understand complex expressions, humans may struggle
      → writing code that is not just functional but also readable.
- Stepwise Transformation of Data:
    - The meeting room example shows how data can be transformed step-by-step (transitions, sorting, etc.) to reach the result.
- Balancing Conciseness and Clarity:
    - Trade-off between writing concise, expressive code and maintaining readability.
- Reversibility of Extraction and Substitution:
    - The process of extraction is the reverse of substitution, illustrating the dual nature of these code transformation techniques.