Objectives

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

- Review class milestones
- JVM languages, Domain Specific Languages, Recursive Descent

Notes

- Quiz 2 on November 13 (announced October 7). Covers all course material since the previous quiz including (but not limited to)
  - Front-end, Javascript (for example, promises)
  - Lean Software Development
  - 4+1 view model and MVC
  - Refactoring
  - Java Functional Programming, Streams, and Concurrency
  - Type Systems and Java
  - ...
- One sprint remaining in the semester
  - Due November 22
- Lab exercise November 8 related to the concurrency utilities

Last Sprint

- For the last sprint, we would like you to submit your sprint goal and backlog by this Friday
- We will also send you data that should be entered into your application database
  - Specific set of users and employees
  - Stations (cities in KZ)
  - Routes (available routes between the stations)

Feature Request/Logging

- *Scrum* is an agile process
  - One of the main goals of agile software development (perhaps *the* main goal) is to respond to change
- Accordingly, we ask all teams to include a *logging* feature, that will be implemented in the final two sprints
  - Your application should include an administration panel that allows managers to configure how data is logged in the application
  - The administrator should be able to enable or disable logging
  - The administration panel should also include an interface that allows an administrator to browse the logs

What to log?

- Your application should log the following
  - API requests: date, agent, user, content type
  - Login and logout as well as failed login attempts

Fluent Interfaces/Builder

```
public class App {
    public static void main(String[] args) {
        TicketBuilder tb = new TicketBuilder();

        Ticket t = tb.from("Astana")
                     .on("March 12")
                     .to("Frankfurt")
                     .withClass("Economy")
                     .build();

        System.out.println(t);
    }
}
```

- This example shows how we could construct a `Ticket` from a `TicketBuilder`
- Notice that the order of the method calls does not matter (as long as we finish with `build()`)

Syntax

```
c = a + b;
```

- What can we say about `a`, `b`, and `c` if we find the following statement in a Java program?

Syntax


```
c = a + b;
```

- What can we say about `a`, `b`, and `c` if we find the following statement in a Java program?
- Those variables are probably either numeric types or strings
- If we are writing a class that has some natural notion of "+" (plus) then we still need to do something like `c = a.add(b);`
- However, other JVM languages give us more options

JVM Languages

- JVM Languages are programming languages that compile to bytecode and run on the Java virtual machine
- Why would someone create a JVM Language?
  - Interoperability with existing Java code and libraries
  - The Java virtual machine is a mature technology (you won't come up with some better bespoke garbage collector in a weekend)
- Examples
  - *Clojure*: Lisp variant
  - *Scala*: object-functional or multi-paradigm language
  - *Groovy*: aimed at cleaner syntax and DSLs
  - *Kotlin*: Android development

Some Scala

```
class TicketBuilder(var origin: String,
                    var dest: String) {
  def from(o: String): TicketBuilder = {
    this.origin = o;
    this}

  def to(t: String): TicketBuilder = {
    this.dest = t;
    this}
}
```

- In Scala, the `TicketBuilder` class might look something like the class above
- Note that the syntax is a little different: `val` and `var`, type annotations use ":", the last statement is the return value
- If you have a method with one argument then we can forego the dot and the parens
  - could just write: `builder from "Astana" to "Almaty"`

Other Features of Scala

- Flexible naming rules
  - Unlike Java, in Scala your methods can be called almost anything and can contain special characters: `*` etc.
- Infix and postfix notation
  - Infix notation looks like a binary operator (method written between the arguments), so we could define the semantics of `a * b`
  - Postfix means that the arguments come before the method
- Type inference and implicits
  - Scala attempts to determine the types identifiers in statements (less verbosity)
  - Can also define *implicit* parameters that automatically search the current context for a value of an appropriate type

Domain Specific Languages

- The attraction of languages like Scala and Groovy is that they enable easier creation of *Domain Specific Languages* (DSLs)
- A DSL is a programming language...
  - aimed at a specific problem
  - uses the same syntax and semantics of experts in that area
- Generally, the goal of DSLs is to allow domain experts to program without having to learn about the implementation details operating under the hood
- Examples
  - Matlab, Mathematica, R
  - Gradle, Make, Rake
  - SQL
  - HTML and CSS

Types of DSLs

- There are two major types of DSLs: *internal* and *external*
- **Internal DSL**
  - Embedded in an existing host language
  - Use idiomatic conventions that express the DSL
  - Can always fall back on the language features for corner cases
- **External DSL**
  - Created as an independent language
  - Requires a custom parser

XML DSL in Scala

```
val xml = <coffee>
  <condiments>
    <condiment>milk</condiment>
    <condiment >sugar</condiment>
  </condiments>
</coffee>
```

- Scala has a built-in DSL to support XML literals
- Note that there are no quotations around this expression, the tags and so forth are actually part of the scala code and are *syntax sugar* for constructors and method calls etc.

DDD

- In 2004, Eric Evans published a popular book called *Domain Driven Design*[1]
- Stressed the importance of the *domain*
  - Every piece of software relates to some activity or interest of its users, this is the *domain* of the software
- Stresses the importance of the *domain model*
  - A model is a "selectively simplified and consciously structured form of knowledge"
  - Developers and domain experts collaborate to fashion a model that becomes the basis for communication between all of the team members
- The domain model can be realized by the creation of a DSL

[1] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

Parsing Concepts

- *Parser*: software that analyzes a sequence of *tokens* and determines if the syntax is correct according to a formal grammar
  - Regular Expressions also do this but only can recognize regular languages
- Grammars consist of the following: Non-terminal Symbols, Terminal Symbols, and Productions (example below)
- Convention for writing a grammar is called Backus-Naur Form

$$\mathrm{E} \rightarrow \mathrm{E} + \mathrm{E} \mid (\mathrm{E}) \mid \text{-}\mathrm{E} \mid \mathrm{id}$$

$$\mathrm{id} \rightarrow \text{[a-z][a-zA-Z0-9]*}$$

Recursive Descent Parsing

- Notice that the rules from the previous example are recursive
  - To define what expressions are we refer to expressions
- *Recursive Descent Parser*: Write a series of mutually recursive functions to implement a grammar
  - One function for each non-terminal symbol
  - Productions correspond to different program control flow constructs
- Recursive Descent parsers are *top-down*
  - Example: if we want to see if a sequence of tokens is a program (according to some grammar) we will call a routine or function named `Program()`

Parser/Scanner Contract

- Parsing involves looking at a stream of tokens or a stream of characters
- It is helpful to have an interface for common operations with a stream of data
- Store tokens in an array, keep track of position with an index
  - `current()`: look at the current token
  - `next()`: look ahead to next token
  - `consume()`: advance the index by 1
  - `terminal(x)`: is the current token `x`, if yes consume and return true
  - `store()` and `undo()`: depending on the grammar we might want to save our current state before checking alternatives and be able to restore

Parser Rules [2], *following*

- We need to know how to convert productions to programs: one thing can follow another, one thing or something else, sequence of things
- Code listings below shows different possibilities
- To implement symbols in sequence we can use nested if blocks

```
// A is a B followed by a C
// A --> B C
function A() {
    if (B()) {
        if (C()) {
            return true;
        }
    }

    return false;
}
```

Rules Parsers, *alternatives*

- Call the alternatives in a production independently
- Non-trivial decision to make in deciding which order to test things in

```
// A is a B or a C
// A --> B | C
function A() {
   if (B()) {
      return true;
   }

   if (C()) {
      return true;
   }

   return false;
}
```

Rules for Parsers, *sequences*

- We can implement a sequence of symbols with a while loop
- Variations depending on if we want to allow empty sequences

```
// A is a sequence of one or more Bs
// A --> B+
function A() {
   if (B()) {
      while(B()) {
         //
      }

      return true;
   }

   return false;
}
```

Example: Arrays and Values

- As a non-trivial example of recursive descent parsing consider common programming language feature of array literals
- An array is a list of values, a value can be a number, identifier, or an array
- This definition clearly involves mutual recursion
- For a Recursive Descent Parser we need two functions: `ARR()` and `VAL()`

Values

- To avoid complexity with tokens assume that a value can be either the symbol a or it can be an array
- Production: VAL $\rightarrow$ TERMINAL(a) | ARR

```
function VAL() {
   if (TERMINAL('a')) {
      return true;
   }

   if (ARR()) {
      return true;
   }

   return false;
}
```

- An array is a sequence of values surrounded by balanced brackets
- Example String: [[a] [aa] [aa [a] []]]
- Production: ARR $\rightarrow$ TERMINAL([) VAL TERMINAL(])

```
function ARR() {
    if (TERMINAL('[')) {
        while(VAL()) {
            //
        }
        if (TERMINAL(']')) {
            return true;
        };
    }

    return false;
}
```

Recursive Descent Parsing Notes

- Cannot deal with all grammars directly.
- *left-recursion*: When a symbol can derive to a form with itself as the left-most symbol
  - If we try to apply the rules above directly to such a grammar we will have programs that recurse infinitely
  - In general the grammar for a language may have more than one form
- *Ambiguous Grammar*: Grammar where sentences or words in the language have more than one derivation
- Practically we also want to check that when our program finishes all of the tokens have been consumed

Parser Outline

- This code listing shows one way to provide the rest of our application with a stream of tokens
  - Here we are treating characters as tokens
- Each of our grammar rules should have shared access to one of these parser or scanner objects

```
function Parser(s) {
   // Setup token stream as array and place scanner at beginning
   this.index = 0;this.tokens = s.split("");

   this.current = function() {
      return this.tokens[this.index];
   };

   this.consume = function() {
      this.index += 1;
   };

   // additional functions
}

exports.Parser = Parser;
```

Going Beyond Recognizing Sentences

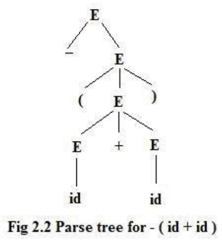

Figure: Parse tree from [3]

- The recursive descent parsers as described here can determine if a sentence (sequence of tokens) is in the language generated by a grammar
- To do some computing we also need to know the parse tree
- *Parse Tree*: Tree associated to a derivation, each interior node is a nonterminal and each leaf is a terminal

References I

[1] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[2] M. Fowler, *Domain-Specific Languages*, ser. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010.

[3] A. Aho and A. Ullman, *Principles of Compiler Design*. Addison Wesley Publishing Company, 1977.