Let’s have a **textual syntax** for FSML: the finite state machine (FSM) language

**An FSM for a turnstile in a metro system**

- **States** (nodes): locked, unlocked, exception
- **Events**: ticket, pass, release, mute
- **Actions**: collect, eject, alarm
- **Transitions** (edges)
Textual syntax for FSML

```plaintext
initial state locked {
  ticket/collect -> unlocked;
  pass/alarm -> exception;
}
state unlocked {
  ticket/eject;
  pass -> locked;
}
state exception {
  ticket/eject;
  pass;
  mute;
  release -> locked;
}
```

N.B.: DSL design (concrete or abstract syntax design) commences in a sample-driven manner.
DSL implementation in different ‘styles’

- **External DSL:**
  - Designated parser, checker, interpreter, compiler

- **Internal DSL:**
  - Implementation as library using host language features

N.B.: This is a gross oversimplification. There are options or hybrids using extensible languages, extensible compilers, metaprogramming systems, and language workbenches.
Dependencies:
• Internal DSL style
We are going to do here …

External DSL style

with ANTLR and Java

N.B.: We are committing to a particular parser generator (ANTLR). We could also be using hand-written parsers, parser combinators, and model-to-text technologies. ANTLR, by itself, also serves other ‘target’ languages, e.g., Python.
Grammar-based concrete syntax definition

fsm : state+ EOF;
state : 'initial'? 'state' stateid '{' transition* '}' ;
transition : event ('/' action)? ('->' target=stateid)? ';' ;
stateid : NAME ;
event : NAME ;
action : NAME ;
NAME : ('a'. 'z'|'A'. 'Z')+ ;

N.B.: This is essentially Extended Backus Naur Form. ‘?’ for options, ‘*’/‘+’ for lists, etc.
Well, we use the specific grammar notation of ANTLR.

N.B.: Action and target state are optional.
ANTLR+Java-based syntax checker

A grammar is almost an effective definition of a syntax checker. We need ‘pragmas’ and driver code (see next slide).

```
grammar Fsml;
@header {package org.softlang.fsml;}
fsm : state+ EOF ;
state : 'initial'? 'state' stateid '{' transition* '}' ;
transition : event ('/' action)? ('->' target=stateid)? ';' ;
stateid : NAME ;
event : NAME ;
action : NAME ;
NAME : ( 'a'..'z'|'A'..'Z' )+ ;
WS : [ \t\n\r]+ -> skip ;
```

Package for generated code

Skip whitespace!
2.3 External DSL

• The grammar is given a name: `Fsml` (line 1). This name is used in names of generated Java classes such as `FsmlParser` and `FsmlLexer`.

• By means of a header pragma, a Java package name is specified: `org.softlang.fsml` (line 2). The generated Java classes are put into this package.

• A special grammar symbol for whitespace is declared: `WS` (line 10); such whitespace is to be skipped in the input, as controlled by the `skip` action.

• Two of the nonterminals use uppercase identifiers: `NAME` and `WS` (lines 9–10). This is a hint for ANTLR that these nonterminals model lexical syntax. That is, the input text is first converted into a sequence `NAME` and `WS` tokens as well as keywords or special tokens from the other rules, before parsing commences.

The development steps for obtaining an (ANTLR-based) syntax checker are briefly summarized by means of a recipe.

**Recipe 2.5 (Development of a syntax checker)**

1. Design the grammar or, in fact, the input for the parser generator.
2. Use a parser generator to generate code for the syntax checker.
3. Develop driver code for applying the generated parser to input.
4. Compile all the code (assuming a compiled language).
5. Set up a test suite with positive and negative language examples.

Here is the driver code, as mentioned in the recipe above, so that syntax checking can be applied to actual files. This boilerplate code is idiosyncratic to ANTLR.

**Illustration 2.19 (Driver code for the generated syntax checker (parser))**

```java
public class FsmlSyntaxChecker {
    public static void main(String[] args) throws IOException {
        FsmlParser parser =
            new FsmlParser(
                new CommonTokenStream(
                    new FsmlLexer(
                        new ANTLRFileStream(args[0]))));
        parser.fsm();
        System.exit(parser.getNumberOfSyntaxErrors() – Integer.parseInt(args[1]));
    }
}
```

This code entails the following steps:

• Token stream to parser
• Lexer to token stream
• Stream to lexer
• Filename to stream

We assume a command-line interface for running positive and negative test cases.

N.B.: This is boilerplate code.
To make things more concrete, a Makefile is included here for building and running the syntax checker.

Illustration 2.15 (Makefile for the FSML syntax checker).

```makefile
# cp = -cp :./lib/*-complete.jar
# antlr = java -cp ../lib/antlr-4.5.3-complete.jar org.antlr.v4.Tool -o org/softlang/fsml
# fsmlSyntaxChecker = java -cp ../lib/antlr-4.5.3-complete.jar org/softlang/fsml

all: 
  make generate 
  make compile 
  make test

generate: 
  $(antlr) Fsml.g4

compile: 
  javac -cp org/softlang/fsml/*

test: 
  $(fsmlSyntaxChecker) ../sample.fsml 0 
  $(fsmlSyntaxChecker) ../tests/syntaxError.fsml 1
```

N.B.: Automation (build management and testing) is crucial in DSL implementation to deal with experimentation and evolution.
From checking to parsing

N.B.:
- AST/CST = abstract/concrete syntax tree
- CSTs are served by a parsing technology like ANTLR.
- ASTs are modeled by language-specific object models.
An ANTLR listener for **abstraction**

```java
public class FsmlToObjects extends FsmlBaseListener {
    private Fsm fsm;
    private State current;
    public Fsm getFsm() { return fsm; }
    @Override public void enterFsm(FsmlParser.FsmContext ctx) {
        fsm = new Fsm();
    }
    @Override public void enterState(FsmlParser.StateContext ctx) {
        current = new State();
        current.setStateid(ctx.stateid().getText());
        fsm.getStates().add(current);
    }
    @Override public void enterTransition(FsmlParser.TransitionContext ctx) {
        Transition t = new Transition();
        t.setSource(current.getStateid());
        t.setEvent(ctx.event().getText());
        if (ctx.action() != null) t.setAction(ctx.action().getText());
        t.setTarget(ctx.target != null ? ctx.target.getText() : current.getStateid());
    }
}
```

N.B.:

- There are ‘events’ for entering (and leaving) nonterminals.
- The listener applies to the grammar for syntax checking.
public class FsmlToObjects extends FsmlBaseListener {
  private Fsm fsm;
  private State current;
  public Fsm getFsm() { return fsm; }
  @Override public void enterFsm(FsmlParser.FsmContext ctx) {
    fsm = new Fsm();
  }
  @Override public void enterState(FsmlParser.StateContext ctx) {
    current = new State();
    current.setStateid(ctx.stateid().getText());
    fsm.getStates().add(current);
  }
  @Override public void enterTransition(FsmlParser.TransitionContext ctx) {
    Transition t = new Transition();
    fsm.getTransitions().add(t);
    t.setSource(current.getStateid());
    t.setEvent(ctx.event().getText());
    if (ctx.action() != null) t.setAction(ctx.action().getText());
    t.setTarget(ctx.target != null ? ctx.target.getText() : current.getStateid());
  }
}
Java code driving the parser

```java
public Fsm textToObjects(String filename) throws IOException {
    FsmlParser parser = new FsmlParser(
        new CommonTokenStream(
            new FsmlLexer(
                new ANTLRFileStream(filename))));
    ParseTree tree = parser.fsm();
    assertEquals(0, parser.getNumberOfSyntaxErrors());
    FsmlToObjects listener = new FsmlToObjects();
    ParseTreeWalker walker = new ParseTreeWalker();
    walker.walk(listener, tree);
    return listener.getFsm();
}
```

N.B.: This is boilerplate code.
‘Minimum’ DSL implementation

- Syntax:
  - Object model for abstract syntax
  - **Parser based on grammar for concrete (textual) syntax**

- (Dynamic) semantics:
  - Interpreter operating on abstract syntax (object model)

- Well-formedness/typedness (aka static semantics):
  - Checker operating on abstract syntax (object model)

N.B.: Everything **not in bold face** can be implemented in the same way as in a DSL implementation in internal style. (Clearly, we only consider here a particular approach to DSL implementation.)
Online resources

YAS’ GitHub repository contains all code. YAS (Yet Another SLR (Software Language Repository))
http://www.softlang.org/yas
See here specifically:
https://github.com/softlang/yas/tree/master/languages/FSML/Java

The Software Languages Book
http://www.softlang.org/book
The topic is covered in Chapter 2.