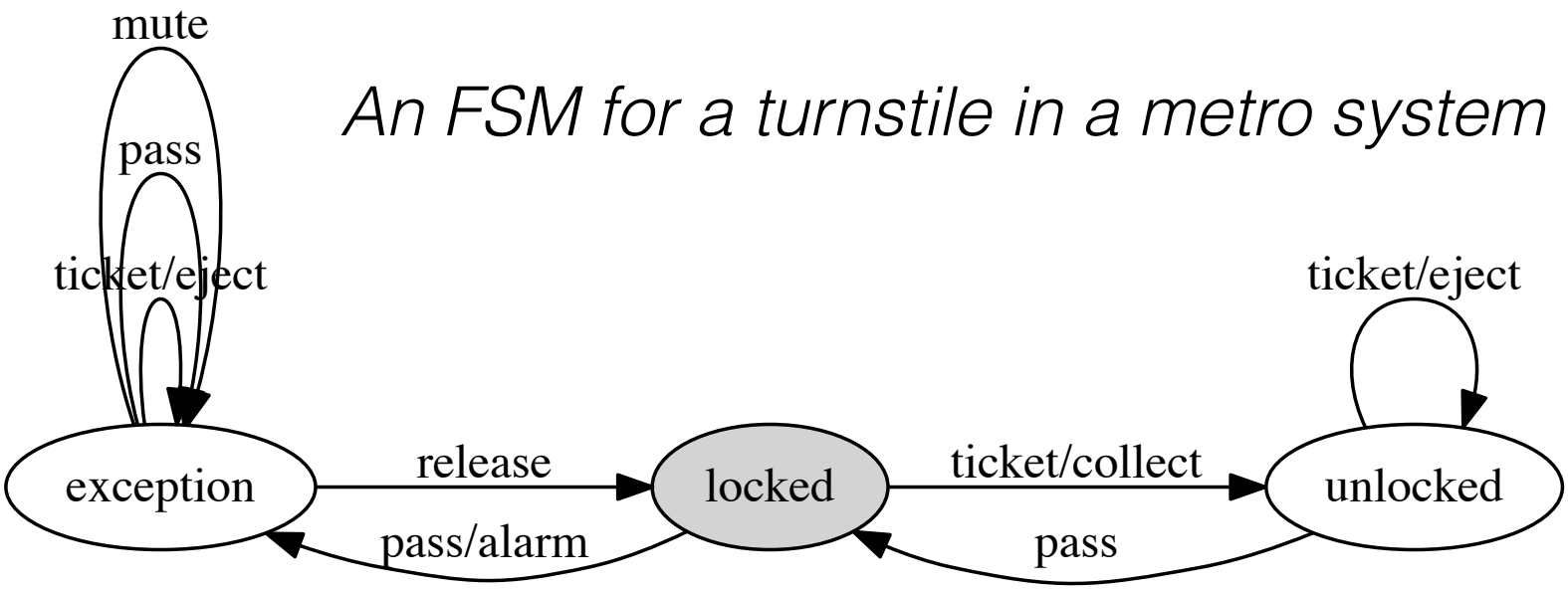
External DSL style

Ralf Lämmel
Software Language Engineering Team
University of Koblenz-Landau
http://www.softlang.org/

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

```
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’

Our focus for now*!*

- **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

https://upload.wikimedia.org/wikipedia/commons/f/f9/Stop.png

We are going to do here …

External DSL style

with 

and 

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.: Action and target state are optional.

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

ANTLR+Java-based syntax checker

```
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!

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

Java code driving the syntax checker

based on generated classes *Parser and *Lexer

```
public class FsmISyntaxChecker {
  public static void main(String[] args) throws IOException {
    FsmIParser parser =
      new FsmIParser(
        new CommonTokenStream(
          new FsmILexer(
            new ANTLRFileStream(args[0]))));
    parser.fsm();
    System.exit(parser.getNumberOfSyntaxErrors()−Integer.parseInt(args[1]));
  }
}
```

- 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.

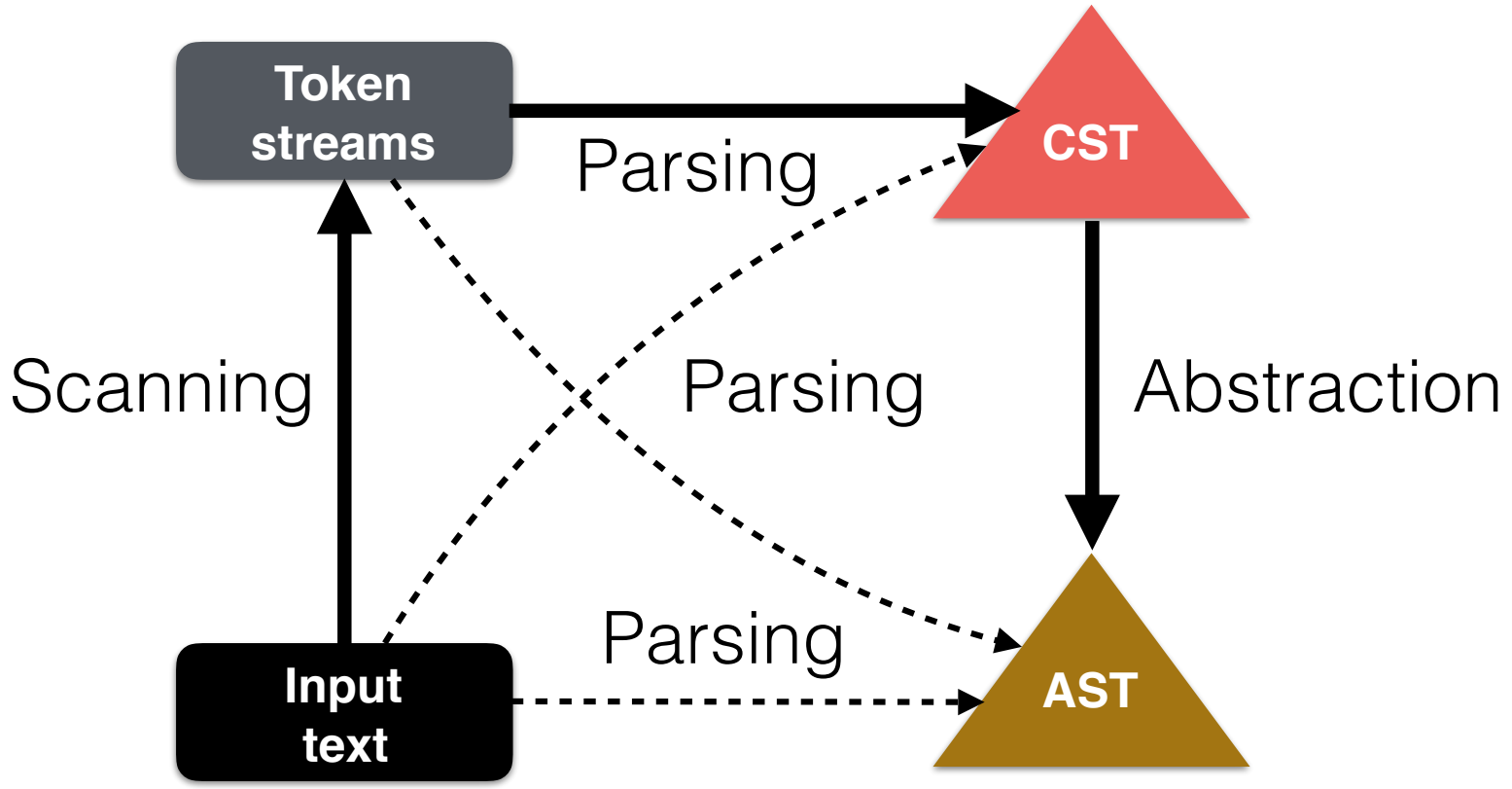
N.B.: This is boilerplate code.

Makefile
for building and testing the syntax checker

```
cp = -cp .:../../../lib/Java/antlr-4.5.3-complete.jar
antlr = java ${cp} org.antlr.v4.Tool -o org/softlang/fsml
fsmlSyntaxChecker = java ${cp} org.softlang.fsml.FsmlSyntaxChecker

all:
  make generate
  make compile
  make test

generate:
  ${antlr} Fsml.g4

compile:
  javac ${cp} org/softlang/fsml/*.java

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

```
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();
    ...
```

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

```
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.