Domain-specific languages

Ralf Lämmel
Software Languages Team
University of Koblenz-Landau
http://www.softlang.org/
A DSL for finite state machines (FSMs)

A FSM for a ‘turnstile’ in a metro system

Imagine the FSM language (FSML) to have started on the black/whiteboard a long time ago.
Concepts of FSML illustrated for the turnstile FSM

States
locked  The turnstile is locked. No passenger is allowed to pass.
unlocked  The turnstile is unlocked. A passenger may pass.
exception  A problem has occurred and metro personnel needs to intervene.

Events
ticket  A passenger inserts a ticket into the card reader.
pass  A passenger passes the turnstile as noticed by a sensor.
mute  Metro personnel turns off alarm after exception.
release  Metro personnel turns on normal operation again.

e/a

Actions
collect  The ticket is collected by the card reader.
eject  The ticket is ejected by the card reader.
alarm  An alarm is turned on and metro personnel is requested.

e/a

Transitions
Semantics (I/O behavior) of FSML illustrated for the turnstile FSM

Input (= sequence of events)

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticket</td>
<td>A ticket is inserted. (The turnstile is unlocked, thus.)</td>
</tr>
<tr>
<td>ticket</td>
<td>Another ticket is inserted. (The superfluous ticket is ejected.)</td>
</tr>
<tr>
<td>pass</td>
<td>Someone passes the turnstile. (This is Ok.)</td>
</tr>
<tr>
<td>pass</td>
<td>Someone passes the turnstile. (This triggers alarm.)</td>
</tr>
</tbody>
</table>

Output (= sequence of actions)

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect</td>
<td>The inserted ticket is collected.</td>
</tr>
<tr>
<td>eject</td>
<td>A ticket inserted in unlocked state is ejected.</td>
</tr>
<tr>
<td>alarm</td>
<td>An attempt to pass in locked state triggers alarm.</td>
</tr>
</tbody>
</table>
DSL implementation in different ‘styles’

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

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

Our initial focus

N.B.: This is a gross oversimplification. There are options or hybrids using extensible languages, extensible compilers, metaprogramming systems, and language workbenches.
Internal DSL style

Ralf Lämmel
Software Languages Team
University of Koblenz-Landau
http://www.softlang.org/
We are going to do here ...  

Internal DSL style  

with Java and python libraries  

N.B.: If we were using C++, Scheme, Haskell, or others for internal DSL implementation, additional or different techniques could or should be leveraged, e.g., operator overloading, macros, or templates.
turnstile = new Fsm();
State s = new State();
s.setStateid("locked");
s.setInitial(true);
turnstile.getStates().add(s);
s = new State();
s.setStateid("unlocked");
turnstile.getStates().add(s);
s = new State();
s.setStateid("exception");
turnstile.getStates().add(s);
Transition t = new Transition();
t.setSource("locked");
t.setEvent("ticket");
t.setAction("collect");
t.setTarget("unlocked");
turnstile.getTransitions().add(t);
t = new Transition();
... add more transitions ...

‘Imperative’
Java API

N.B.: Arguably this was common style until perhaps 200?. Even today, this sort of code may still be written.
Java API with functional constructors

turnstile = new Fsm();
turnstile.getStates().add(new State("locked", true));
turnstile.getStates().add(new State("unlocked"));
turnstile.getStates().add(new State("exception"));
turnstile.getTransitions().add(new Transition("locked", "ticket", "collect", "unlocked"));
turnstile.getTransitions().add(new Transition("locked", "pass", "alarm", "exception"));
... add more transitions ...

N.B.: Functional constructors have been used by C++ and Java et al. programmers for a long time, but they are insufficient to avoid repetitive code and to hide the internal representation.
public class Fsm {
    private List<State> states = new LinkedList<>();
    private List<Transition> transitions = new LinkedList<>();
    public List<State> getStates() { return states; }
    public List<Transition> getTransitions() { return transitions; }
}

public class State {
    private String id;
    private boolean initial;
    public String getStateid() { return id; }
    public void setStateid(String state) { this.id = state; }
    public boolean isInitial() { return initial; }
    public void setInitial(boolean initial) { this.initial = initial; }
    public State() {
    }
    public State(String id) { this.id = id; }
    public State(String id, boolean initial) { this.id = id; this.initial = initial; }
}

public class Transition {
    private String source;
    private String event;
    private String action;
    public String getSource() { return source; }
    public void setSource(String source) { this.source = source; }
    public String getEvent() { return event; }
    public void setEvent(String event) { this.event = event; }
    public String getAction() { return action; }
    public void setAction(String action) { this.action = action; }
}
private String id;
private boolean initial;
public String getStateid() { return id; }
public void setStateid(String state) { this.id = state; }
public boolean isInitial() { return initial; }
public void setInitial(boolean initial) { this.initial = initial; }
public State() {}
public State(String id) { this.id = id; }
public State(String id, boolean initial) { this.id = id; this.initial = initial; }
}

public class Transition {
    private String source;
    private String event;
    private String action;
    private String target;
    ...
    getters and setters ...
    public Transition() {}
    public Transition(String source, String event, String action, String target) {
        this.source = source;
        this.event = event;
        this.action = action;
        this.target = target;
    }
}
Use of a fluent API in Java

Fsm turnstile =
   fsm()
   .addState("locked")
     .addTransition("ticket", "collect", "unlocked")
     .addTransition("pass", "alarm", "exception")
   .addState("unlocked")
     .addTransition("ticket", "eject", "unlocked")
     .addTransition("pass", null, "locked")
   .addState("exception")
     .addTransition("ticket", "eject", "exception")
     .addTransition("pass", null, "exception")
     .addTransition("mute", null, "exception")
     .addTransition("release", null, "locked");

Leveraged techniques:
- Factory methods
- Method chaining
- Implicit parameters
- Conventions (defaults)

N.B.: The current state is maintained along the way. The state declared first is assumed to be the initial one. The representation is not revealed—no constructors are used.
Use of a **fluent API** in Python

```python
turnstile = Fsm()
  .addState("locked")
    .addTransition("ticket", "collect", "unlocked")
    .addTransition("pass", "alarm", "exception")
  .addState("unlocked")
    .addTransition("ticket", "eject", "unlocked")
    .addTransition("pass", None, "locked")
  .addState("exception")
    .addTransition("ticket", "eject", "exception")
    .addTransition("pass", None, "exception")
    .addTransition("mute", None, "exception")
    .addTransition("release", None, "locked")
```

Leveraged techniques:
- Factory methods
- Method chaining
- Implicit parameters
- Conventions (defaults)

N.B.: If we were using C++, Scheme, Haskell, or others for internal DSL implementation, additional or different techniques could or should be leveraged, e.g., operator overloading, macros, or templates.
Definition of **fluent API** in Java

```java
public interface Fsm {
    public Fsm addState(String state);
    public Fsm addTransition(String event, String action, String target);
    public String getInitial();
    public ActionStatePair makeTransition(String state, String event);
}
```

```java
public class ActionStatePair {
    public String action;
    public String state;
}
```

N.B.: This interface does not expose the internal representation.
The interface does not just cover fluent construction; it also covers ‘observation’ of the opaque representation.
public class FsmlImpl implements Fsm {
    private String initial; // the initial state
    private String current; // the "current" state
    // A cascaded map for maintaining states and transitions
    private HashMap<String, HashMap<String, ActionStatePair>> fsm =
        new HashMap<>();
    private FsmlImpl() {
    }
    // Construct FSM object
    public static Fsm fsm() { return new FsmlImpl(); }
    // Add state and set it as current state
    public Fsm addState(String id) {
        // First state is initial state
        if (initial == null) initial = id;
        // Remember state for subsequent transitions
        this.current = id;
        if (fsm.containsKey(id)) throw new FsmlDistinctIdsException();
        fsm.put(id, new HashMap<String, ActionStatePair>());
        return this;
    }
    // Add transition for current state
    public Fsm addTransition(String event, String action, String target) {
        ...
if (fsm.containsKey(id)) throw new FsmlDistinctIdsException();
fsm.put(id, new HashMap<String, ActionStatePair>();
return this;
}

// Add transition for current state
public Fsm addTransition(String event, String action, String target) {
    if (fsm.get(current).containsKey(event)) throw new FsmlDeterministismException();
    ActionStatePair pair = new ActionStatePair();
    pair.action = action;
    pair.state = target;
    fsm.get(current).put(event, pair);
    return this;
}

// Getter for initial state
public String getInitial() {
    return initial;
}

// Make transition
public ActionStatePair makeTransition(String state, String event) {
    if (!fsm.containsKey(state)) throw new FsmlResolutionException();
    if (!fsm.get(state).containsKey(event)) throw new FsmlInfeasibleEventException();
    return fsm.get(state).get(event);
}
public class FluentTest {

    private static final String[] input = {
        "ticket", "ticket", "pass", "pass", "ticket", "mute", "release"};
    private static final String[] output = {
        "collect", "eject", "alarm", "eject"};

    @Test
    public void runSample() {
        assertEquals(output, run(Sample.turnstile, input));
    }
}

N.B.: This is how a Java programmer (a DSL user) would document a use case of a specific FSM (and validate intuitions).
public class FsmlInterpreter {
    public static String[] run(Fsm fsm, String[] input) {
        ArrayList<String> output = new ArrayList<>();
        String state = fsm.getInitial();
        for (String event : input) {
            ActionStatePair pair = fsm.makeTransition(state, event);
            if (pair.action != null) output.add(pair.action);
            state = pair.state;
        }
        return output.toArray(new String[output.size()]);
    }
}

N.B.: The interpreter essentially models the dynamic semantics of FSML. This is a non-interactive interpreter. In practice, an interactive DSL implementation may be required.
class Fsm():
    def __init__(self):
        self.fsm = defaultdict(list)
        self.current = None
    def addState(self, id):
        return self.addStateNoDefault(self.current is None, id)
    def addStateNoDefault(self, initial, id):
        if id in self.fsm[id]: raise FsmlDistinctIdsException;
        self.stateObject = dict()
        self.stateObject['transitions'] = defaultdict(list)
        self.stateObject['initial'] = initial
        self.fsm[id] += [self.stateObject]
        self.current = id
        return self
    def addTransition(self, event, action, target):
        if event in self.stateObject['transitions']:
            raise FsmlDeterminismException;
        self.stateObject['transitions'][event] += \
        [(action, self.current if target is None else target)]
        return self

N.B.: no high-level API is provided for ‘observation’; one would access the dictionary directly.
An interpreter (a ‘simulator’) in Python

def run(fsm, input):
    # Determine initial state
    for id, [decl] in fsm.iteritems():
        if decl['initial']:
            current = decl
            break
    # Consume input; produce output
    output = []
    while input:
        event = input.pop(0)
        if event not in current['transitions']: raise FsmlInfeasibleEventException
        else:
            [(action, target)] = current['transitions'][event]
            if action is not None: output.append(action)
            if target not in fsm: raise FsmlResolutionException
            [current] = fsm[target]
    return output

N.B.: When compared to the Java-based interpreter, we access directly the presentation.
‘Minimum’ DSL implementation

✓ Syntax (fluent API for internal DSL)

✓ (Dynamic) semantics (e.g., by means on an interpreter)

- Well-formedness / -typedness (aka static semantics)

N.B.: Just like the interpreter, we implement a ‘well-formedness checker’ as functionality on top of (the API for) the internal DSL representation. (We could use a constraint language such as OCL.)
Well-formedness of FSMs

distinctStateIds  The state ids of the state declarations must be distinct.
singleInitialState  An FSM must have exactly one initial state.
deterministicTransitions  The events must be distinct per state.
resolvableTargetStates  The target state of each transition must be declared.
reachableStates  All states must be reachable from the initial state.

```java
resolutionNotOk = \ 
  Fsm() \ 
  .addState("stateA") \ 
  .addTransition("event1", "action1", "stateB") \ 
  .addTransition("event2", "action2", "stateC") \ 
  .addState("stateB")
```

N.B.: This sample violates `resolvableTargetStates`.

N.B.: a violated `resolvableTargetStates` can (should) be detected even before running an FSM on a specific input.
def ok(fsm):
    for fun in [distinctStateIds,
                singleInitialState,
                deterministicTransitions,
                resolvableTargetStates,
                reachableStates ]:
        fun(fsm)

N.B.: Violations of distinctStateIds and deterministicTransitions can be detected during construction, but we may need explicit checks if we also accommodate ‘serialization’.

def distinctStateIds(fsm):
    for state, decls in fsm.items():
        if not len(decls) == 1: raise FsmlDistinctIdsException()

def singleInitialState(fsm):
    initials = [initial for initial, [decl] in fsm.items() if decl["initial"]]
    if not len(initials) == 1: raise FsmlSingleInitialStateException()

def deterministicTransitions(fsm):
    for state, [decl] in fsm.items():
        for event, transitions in decl["transitions"].items():
            if not len(transitions) == 1: raise FsmlDeterminismException()

def resolvableTargetStates(fsm):  ...
def singleInitialState(fsm):
    initials = [initial for initial, [decl] in fsm.iteritems() if decl["initial"]]
    if not len(initials) == 1: raise FsmlSingleInitialStateException()

def deterministicTransitions(fsm):
    for state, [decl] in fsm.iteritems():
        for event, transitions in decl["transitions"].iteritems():
            if not len(transitions) == 1: raise FsmlDeterminismException()

def resolvableTargetStates(fsm):
    for _, [decl] in fsm.iteritems():
        for _, transitions in decl["transitions"].iteritems():
            for (_, target) in transitions:
                if not target in fsm: raise FsmlResolutionException()

def reachableStates(fsm):
    for initial, [decl] in fsm.iteritems():
        if decl["initial"]:  
            reachables = set([initial])
            chaseStates(initial, fsm, reachables)
        if not reachables == set(fsm.keys()): raise FsmlReachabilityException()

# Helper for recursive closure of reachable states
def chaseStates(source, fsm, states): ...
External DSL style

Ralf Lämmel
Software Languages 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

```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.
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.: 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 ;

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

```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]));
    }
}
```

- 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.
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 FsmIToObjects extends FsmIBaseListener {
    private Fsm fsm;
    private State current;
    public Fsm getFsm() { return fsm; }
    @Override public void enterFsm(FsmIParser.FsmContext ctx) {
        fsm = new Fsm();
    }
    @Override public void enterState(FsmIParser.StateContext ctx) {
        current = new State();
        current.setStateid(ctx.stateid().getText());
        fsm.getStates().add(current);
    }
    @Override public void enterTransition(FsmIParser.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.)
Program generation with template processing

Ralf Lämmel
Software Languages Team
University of Koblenz-Landau
http://www.softlang.org/
Problem statement

N.B.:  
- The C code should run on some ‘target hardware’.  
- The ‘program generator’ uses ‘template processing’.
An FSM for a turnstile in a metro

- **States** (nodes): locked, unlocked, exception
- **Events**: ticket, pass, release, mute
- **Actions**: collect, eject, alarm
- **Transitions** (edges)
enum State { LOCKED, UNLOCKED, EXCEPTION, UNDEFINED };
enum State initial = LOCKED;
enum Event { TICKET, RELEASE, MUTE, PASS };
void alarm() {}
void eject() {}
void collect() {}
enum State next(enum State s, enum Event e) {
    switch(s) {
        case LOCKED:
            switch(e) {
            case TICKET: collect(); return UNLOCKED;
            case PASS: alarm(); return EXCEPTION;
            default: return UNDEFINED;
            }
            case UNLOCKED:
                switch(e) {
                case TICKET: eject(); return UNLOCKED;
                case PASS: return LOCKED;
                default: return UNDEFINED;
                }
        case EXCEPTION:
            switch(e) {
                case TICKET: eject(); return EXCEPTION;
                case PASS: return EXCEPTION;
                case MUTE: return EXCEPTION;
                case RELEASE: return LOCKED;
                default: return UNDEFINED;
            }
    default: return UNDEFINED;
    }
}
‘Hello, World’ of template processing

Source: https://github.com/antlr/stringtemplate4/blob/master/doc/introduction.md

**Templates** contain text and references to names:

Hello, <name>

**Actual parameters can be passed to a rendering process**

```java
import org.stringtemplate.v4.*;
...
ST hello = new ST("Hello, <name>"');
hello.add("name", "World");
System.out.println(hello.render());
```

**The rendering result is text.**

Hello, World

N.B.: There exist many template processors. We use StringTemplate http://www.stringtemplate.org/ which is a mainstream option for the Java platform. StringTemplate can also be used with other languages.
N.B.:
- IR = Intermediate Representation (an AST closer to output).
- We massage FSML ‘models’ to appeal to code generation patterns.
- We could also translate FSML ‘models’ directly into C ‘models’.
public class Fsm {
    public List<State> states = new LinkedList<>();
    public List<Transition> transitions = new LinkedList<>();
}

class State {
    public String stateid;
    public boolean initial;
}

class Transition {
    public String source;
    public String event;
    public String action;
    public String target;
}

A very simple Java-based model of FSMs

N.B.: In reality, we use private fields, public getters, and possibly functional constructors instead of public fields. In fact, we may also assume a fluent API.
// Main template for complete source code

main(states, initial, events, actions, tgroups) ::= << ... >>

// Template for functions that implement actions

action(a) ::= "..."

// switch-case for transitions grouped by source state

tgroup(g) ::= <<...>>

// case for a specific transition

transition(t) ::= <%...%>

N.B.: Because the templates have names, they can refer to each other. This is called a template group in StringTemplate.
main(states, initial, events, actions, tgroups) ::= <<
enum State { <states; format="upper", separator="", "> };
enum State initial = <initial; format="upper">;
enum Event { <events; format="upper", separator="", ">
<actions:action(); format="lower", separator="\n">;
enum State next(enum State s, enum Event e) {
    switch(s) {
    <tgroups:tgroup(); separator="\n">;
        default: return UNDEFINED;
    }
}}

action(a) ::= "void <a>() { "}

tgroup(g) ::= <<
    case <g.stateid; format="upper">:
        switch(e) {
        <g.ts:transition(); separator="\n">;
            default: return UNDEFINED;
        }
>>

transition(t) ::= <%
    case <t.event; format="upper">:
    <if(t.action)<t.action; format="lower">(); <endif>
    return <t.target; format="upper">;%>
# StringTemplate’s metanonotation

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tmpl(p) ::= &quot;...&quot;</code></td>
<td>Define a template <code>tmpl</code> with parameter <code>p</code></td>
</tr>
<tr>
<td><code>tmpl(p) ::= &lt;&lt;...&gt;&gt;</code></td>
<td>Multiline template with indentation and linebreaks</td>
</tr>
<tr>
<td><code>tmpl(p) ::= &lt;...%&gt;</code></td>
<td>Ignore indentation and linebreaks</td>
</tr>
<tr>
<td><code>&lt;p&gt;</code></td>
<td>Refer to parameter <code>p</code></td>
</tr>
<tr>
<td><code>&lt;p; format=&quot;upper&quot;&gt;</code></td>
<td>Format in uppercase</td>
</tr>
<tr>
<td><code>&lt;p; separator=&quot;&quot;, &quot;&quot;&gt;</code></td>
<td>Separate elements of collection by comma</td>
</tr>
<tr>
<td><code>&lt;p:tmpl()&gt;</code></td>
<td>Apply <code>tmpl</code> on (each element of) <code>p</code></td>
</tr>
<tr>
<td><code>&lt;p.x&gt;</code></td>
<td>Refer to property <code>x</code> of <code>p</code></td>
</tr>
<tr>
<td><code>&lt;if(p.x)&gt;...&lt;endif&gt;</code></td>
<td>Include text only if <code>p.x</code> is true or not NULL</td>
</tr>
</tbody>
</table>
import org.stringtemplate.v4.ST;
import org.stringtemplate.v4.STGroup;
import org.stringtemplate.v4.STGroupFile;
import org.stringtemplate.v4.StringRenderer;
import java.io.File;
import java.util.*;

public class Fsm1CGGenerator {
    private static class TGroup {
        public String stateid;
        public List<Transition> ts;
    }

    public static String generate(Fsm fsm) {
        // Build list of states with extra "UNDEFINED"
        List<String> states = ...;
        // Build set of events
        Set<String> events = ...;
        // Build set of actions
        Set<String> actions = ...;
        // Group transitions by state
        List<TGroup> tgroups = ...;

        // Load template group and retrieve top-level template
        STGroup group = new STGroupFile(... + "Fsm.stg");
        group.registerRenderer(String.class, new StringRenderer());
        ST main = group.getInstanceOf("main");

        // Set template parameters and render
        main.add("states", states);
        main.add("initial", fsm.getInitial());
        main.add("events", events);
        main.add("actions", actions);
        main.add("tgroups", tgroups);
        return main.render();
    }
}
// Build list of states with extra "UNDEFINED"
List<String> states = new LinkedList<>();
for (State s : fsm.getStates()) states.add(s.getStateid());
states.add("UNDEFINED");

// Build set of events
Set<String> events = new HashSet<>();
for (Transition t : fsm.getTransitions()) events.add(t.getEvent());

// Build set of actions
Set<String> actions = new HashSet<>();
for (Transition t : fsm.getTransitions())
    if (t.getAction() != null) actions.add(t.getAction());

// Group transitions by state
List<TGroup> tgroups = new LinkedList<>();
for (State s : fsm.getStates()) {
    TGroup tg = new TGroup();
    tg.stateid = s.getStateid();
    tg.ts = new LinkedList<>();
    for (Transition t : fsm.getTransitions())
        if (tg.stateid.equals(t.getSource())) tg.ts.add(t);
    tgroups.add(tg);
}

N.B.: We aim at a strict separation of model and view and thus, we do not try to ‘compute’ anything in the template.
Template processing issues

- **Escaping** template metanotation
- Spacing, **indentation**, and linebreaks
- Underlying use of **reflection** (typically)
- **Formatting** (e.g., upper- versus lowercase)
- **Mapping** the model to the template parameters
- **Idiosyncrasies** of particular template metanotation
A Jinja2 template for use with Python

```plaintext
enum State { {{states|join(',', ')|upper()}}} };
enum State initial = {{initial|upper}};
enum Event { {{events|join(',', ')|upper()}}} };

{% for a in actions %}void {{a}}(){ }{% endfor %}

enum State next(enum State s, enum Event e) {
    switch(s) { {% for (s, ts) in transitions %}
        case {{s|upper()}}:
            switch(e) { {% for (e, a, t) in ts %}
                case {{e|upper()}}: {% if a %}{{a}}(); {%endif%}return {{t|upper()}};{% endif %}
                default: return UNDEFINED;
            }{% endfor %}
            default: return UNDEFINED;
        }{% endfor %}
    }{% endif %}
}

N.B.: This approach slightly differs from the one shown before.
In particular, we use a for-loop.
Also, we use a single template as opposed to a group.
```
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
Subdirectories Java and Python

The Software Languages Book
http://www.softlang.org/book
See Chapter 2 in particular.