Assignment 1
Railway Modelling in metaDepth

Simon Van Mierlo
`simon.vanmierlo@uantwerpen.be`

1 Practical Information

The goal of this assignment is to design a domain-specific modelling language (formalism) and subsequently to model railways in that language in the textual modelling tool metaDepth. The different parts of this assignment are listed below:

1. Implement the abstract syntax of your language in metaDepth.
2. Enrich the abstract syntax with constraints (using EOL) so that you can check that every model is well-formed.
3. Create some railway models that are representative for all the features in your language. Some should be valid models (check by verifying them), and some should be invalid, to show that your constraints are correct.
4. Write operational semantics (using EOL) that simulate the railway system.

Write a report that includes a clear explanation of your complete solution, the modelling choices you made, as well as an explanation of your testing process. Also mention possible difficulties you encountered during the assignment, and how you solved them. You will have to complete this assignment in groups of 2. Submit your assignment (report in pdf, abstract syntax definition, example models, and simulator) on Blackboard before Friday, October 14, 13:00h.

Contact Simon Van Mierlo (`simon.vanmierlo@uantwerpen.be`) if you have a problem.

2 Requirements

This section lists the requirements of the railway domain-specific language. They are split into two sections: one on abstract syntax, and one on operational semantics. Make sure to test each requirement with test models!

2.1 Abstract Syntax

The abstract syntax of the DSL captures its *syntax* and *static semantics*. The requirements of the railway language are listed below:

- A railway system consists of the infrastructure with the trackwork, the signalling equipment, and the stations, as well as the trains driving on the trackwork and through the stations.
- A railway network consists of a number of interconnected railway *segments*. The types of segments your language needs to support are listed below:
  - **Straights**—the most trivial segment, allows a train to move straight. Has one incoming and one outgoing segment.
  - **Turnouts**—allows a train to go straight, or take a diverging route to another track connected to the segment. Has one incoming segment and two outgoing segments. The direction of a turnout can be changed—it's either in *straight* mode, meaning the train will take the straight route, or in *diverging* mode, meaning the train will take the diverging route.
  - **Junctions**—joins two segments. Has two incoming segments and one outgoing segment. The direction of a junction can be changed—it's either in *straight* or *diverging* mode, selecting the train which can enter the segment.
  - **Stations**—like straights, but can also be at the end or beginning of a track. Has zero or one incoming and zero or one outgoing segment, but is always connected to at least one track. Each station has a unique name, that starts with a single upper case letter followed by zero or more lower case letters, and ends with zero or more numbers.
- Although an undesirable property at runtime, your language should allow for more than one train to be present on a railway segment.
- A railway network does not allow loops (*i.e.*, it should never be possible to start at a station, move in one direction, and end up at that same station). Tip: to model this constraint, use EOL's **closure** function, as you have to write it as one line of code.
- The signalling equipment of the railway system consists of *lights*. These lights control the traffic on a segment and can be in two states—*red* means traffic on that segment needs to wait, *green* means the traffic can pass. Segments only allow one-way traffic.
- Trains have a unique identifier, that start with an upper case letter, followed by four numbers. A train can be on at most one segment.

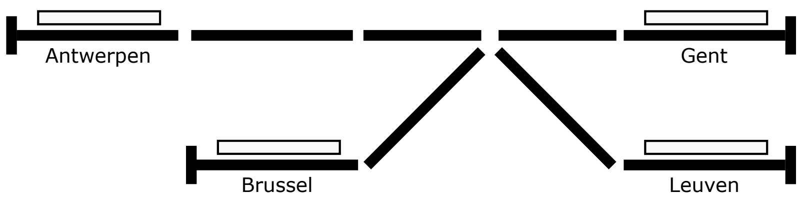
Each train has an associated *schedule*. This schedule is modelled in a second domain-specific language. We do this because we want to load multiple schedules for the same railway network. The requirements for the scheduling language are listed below:

- A schedule is associated to a train by referring to the name of the train.
- The schedule of a train tells it where to go—it contains a sequence of consecutive steps. The 'start' step contains the name of the station where the train will start. It has no incoming steps and one outgoing step. The 'end' step contains the name of the station where the train will stop. It has one incoming step and no outgoing steps. The start and end steps are mandatory. In between, the train has to be instructed to take the diverging or straight route when it encounters a turnout. These 'in-between' steps have exactly one incoming and one outgoing step.
- Each train needs exactly one schedule.
- Each schedule needs exactly one train.

2.2 Operational Semantics

In this part of the assignment, you will model the semantics of a "control room" that regulates the flow of trains on the railway network. The goal is to get each train to its destination safely. This means that the control room needs to make sure two trains can never be on the same track, as well as switching turnouts and junctions to the correct position. More detailed rules are listed below:

- The simulation is broken up into a number of "steps". In each step, the control room first sets all lights to the correct "mode" and switches the direction of turnouts and junctions. After that, all trains move (concurrently) to the next segment, if allowed.
- In the initial step, all trains are placed in their start station.
- A train is allowed to move to the next segment if the light on its current segment is set to green. If the light is red, the train has to wait.
- If a train is on a turnout, it moves in the direction the turnout is set to. In other words, a train has no access to its schedule and follows the directions set by the control room.
- The control room iterates over all segments that contain a train. If no train is present on the segment the train wants to move to, it sets the light of the segment the train is currently on to green.



E2718: Antwerpen -> Straight -> Gent
P3142: Brussel -> Diverging -> Leuven

Figure 1: An example Railway model.

- If a train is on a turnout, the control room switches the direction to where the train wants to go, by looking at the next step in the schedule of the train. It is an error if there is no step (the schedule has reached the end) when a train wants to leave a turnout.
- In case of two trains wanting to enter a junction at the same time, the control room chooses one randomly.
- A train can only enter a junction if the direction of the junction is set correctly.
- When a train reaches its end station, it is removed from the model.

Your simulation should produce a textual trace. The trace associated with the model shown in Figure 1.

```
E2718 | starts at station:Antwerpen
P3142 | starts at station:Brussel
---===---=== Step 1 ===---===---
E2718 | moves forward past the green light
P3142 | moves forward past the green light
---===---=== Step 2 ===---===---
E2718 | waits for a red light
P3142 | moves forward past the green light
---===---=== Step 3 ===---===---
E2718 | moves forward past the green light
P3142 | moves forward past the green light and
    leaves the junction diverging
P3142 | arrives at station:Leuven
---===---=== Step 4 ===---===---
E2718 | moves forward past the green light and
    leaves the junction straight
E2718 | enters station:Gent
---===---=== Step 5 ===---===---
```

```
E2718 | moves forward past the green light
E2718 | arrives at station:Brugge
```

3 Useful Links

- metaDepth download: `http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/assignments/metaDepth.jar`
- metaDepth main page: `http://metadepth.org/`
- metaDepth information:
  - `http://metadepth.org/papers/TOOLS.pdf`
  - `http://metadepth.org/Documentation.html`
  - `http://metadepth.org/Examples.html`
  - The examples we created during the lecture: `FSA.mdepth` and `FSAsim.mdepth`
  - `MetaDepthEOL.txt` (an ever updating list of remarks—contribute by emailing your issues!)
- Epsilon book: [PDF]. For this exercise, you need chapter 3 on the Epsilon Object Language (EOL).