Domain-Specific Languages with Xtext or Scala: Final report

Dominik Habermann (108019250645)
Masuthan Mathiyalakan (108017235408)

March 7, 2024

Contents

1 Introduction

Technologies like DSLs (domain specific language) can be understood with examples, which serve as best way to explain different aspects, e.g. use cases, typical problems, limitations, etc. They can be even better understood, when instead of an artificial example, a real world example is created, which offers a more plain view to a technology.

Because of this reason the IUPAC naming system as an example was chosen. This is a naming convention from the chemistry, especially in the organic chemistry this convention is used frequently. For classification: *This report focuses on the naming system itself* and has only low contact points to the chemical background.

This document is the final report of the Project 18: *Domain-Specific Languages with Xtext or Scala* in WS 23/24. The chapter 2 (Introduction to the IUPAC nomenclature) contains historical background and an introduction to the domain on an abstract level. Basic rules are introduced and corresponding to the new rule a possible context-free grammar is proposed. In the chapter 3 the implementation via Xtext is shown; at the begin the abstract syntax via a meta model and suitable Xtext code as an implementation of the concrete syntax. At the end a discussion with conclusion as well as a reflection is given in the last two chapters 4 + 5.

2 Introduction to the IUPAC nomenclature

2.1 History of the IUPAC naming convention

IUPAC is an abbreviation for *International Union of Pure and Applied Chemistry*. This union was founded in 1919 with the goal to simplify the global communication between chemists. Up to this year no global rules were available for naming chemical structures. Every country used his own system and this leaded to many misunderstandings; sometimes even in a single country. The situation can be compared like before the metric system was introduced.

The IUPAC invented a systematic way to create a unique[1] name for – in theory – every chemical structure. In most cases the literature uses the term „IUPAC name“ or „IUPAC nomenclature“ for a name in the IUPAC convention. In this report both of this terms will be also used.

The rules of this naming convention can be formulated as context-free grammar. So it can be also understood as a kind of DSL. In the following sections basic rules of the naming convention will be introduced. And also the way to cast it to a context-free grammar.

Carbon (C) has the unique skill to create long structures with branches and nesting, that are stable – in a chemical point of view. The possibilities are tremendous. Almost all other elements are not capable of creating larger structures. So the naming convention focuses on the carbon and the describing of that structures.

2.2 Nothing than straight chains

In a simplified manner carbon atoms can be compared with building blocks: The simplest way is a straight chain of carbon atoms without any branches. The IUPAC nomenclature encodes the length of such a straight chain in specific terms like „Methan“, „Ethan“, etc. See the table 2.1 below for a length up to 10 carbon atoms.

[1] A unique name is only with the implementation of all semantic rules possible.

Number of carbon atoms	Encoded name
1	Methan
2	Ethan
3	Propan
4	Butan
5	Pentan
6	Hexan
7	Heptan
8	Octan
9	Nonan
10	Decan

Table 2.1: List of the first 10 names for a straight chain

For example:
Ethan C—C
or
Hexan C—C—C—C—C—C

In a sense of a grammar: This term for the straight chain is the main symbol. We call this symbol c (for chain). S is our mandatory start symbol and thus it is a non terminal symbol. For this tutorial T the set for terminal symbols; N for non terminal and P for all production rules are used. In summary our first CFG grammar has the following definition:

$$
\begin{aligned}
T &= \{c\} \\
N &= \{S\} \\
P &: \\
S &\rightarrow c
\end{aligned}
$$

Symbol	Description
c	Main chain

Figure 2.1: The CFG with only straight chains.

2.3 First order branches

A chain of carbon atoms can also contain branches. This branches could also be nested; for now we focus on branches without deeper nesting. To identify a branch we need the length of this branch and the position. The position will be formulated from the point of view of the main chain. The length of the branch will be encoded in terms. These terms

are comparable with the terms for the main chain but with a postfix „yl“ instead of „an“. This are the first 4 terms for the encoded branch length:

Number of carbon atoms in branch	Encoded name
1	Methyl
2	Ethyl
3	Propyl
4	Butyl

Table 2.2: List of the first 4 names for a branch

An example for this:

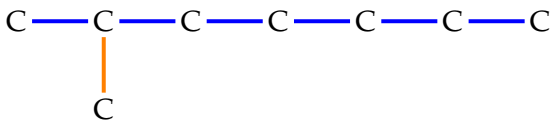

Figure 2.2: Example with one branch of the length 1

The orange one is the branch with the length of one C atom. So the name is *Methyl*. Any position information is written in front of the corresponding branch with a minus char between them. The full branch is therefore: *2-Methyl*. The main chain has the length 7 $\rightarrow$ *Heptan*.

So we can see, that a branch with the position information consists of three parts:

- Position $p \in \mathbb{N}$
- Minus sign m
- Term for the length of the branch b

Because every branch has this structure, we can introduce a non terminalsymbol to summarize all three terminals in this symbol B.

What about the order of branches and main chain? The IUPAC convention defines, that the term for the main chain will be always at the end of the name. This is good for the CFG definition, because we don't need to add production rules for both orders. With this and the other new information we have now following grammar (new elements are highlighted):

$$
\begin{aligned}
T &= \{c, p, m, b\} \\
N &= \{S, B, M\} \\
P &: \\
S &\rightarrow M \\
M &\rightarrow Bc \\
M &\rightarrow c \\
B &\rightarrow pmb
\end{aligned}
$$

Symbol	Description
c	Main chain
p	position
m	minus char
b	branch

Figure 2.3: CFG with exact one branch. $B \rightarrow pmb$ is an optional production rule.

It is important to notice, that an additional non terminal (M) was added. Because a grammar can only contains exact one start variable, we need M as „middle variable“ to make words with and without branches possible. In summary: the B is – like a branch – an optional non terminal. The whole name of the structure in figure 2.2 is: *2-MethylHeptan*[2].

2.4 Multiple first order branches

No one can forbid carbon atoms to create multiple branches of the same length. Let add an another branch of the length one at the third position:

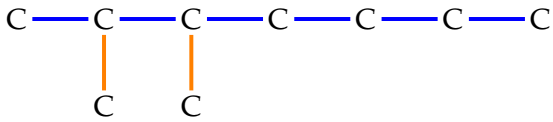

Figure 2.4: Example with two branches of the length 1

The intuitive way is to assume, that a second branch with the new position will be added to the begin of the name. It would be understandable; but it increases the name and in many molecules 5 branches and sometimes more are usual. For this reason there is a prefix to summarize branches of the same length. Table 2.3 shows the first 4 prefixes.

[2]It can produced with $S(M(B(pmb), c))$

Branches with the same length	Prefix
1	Mono
2	Di
3	Tri
4	Tetra

Table 2.3: List of the first 4 prefixes to summarize branches with the same length

In normal cases the prefix „Mono“ is not used. So in the name *1-MethylHeptan* is no „Mono“ missing! **The nomenclature defines no specific rules for the usage. For the introduction „Mono“ will be not used. In the implementation „Mono“ will for the sake of simplicity be always used.**

To identify also the second branch, we need the position of them. All position information will be written before the branch term and the prefix to summarize branches with the same length before the corresponding branch term: *2,3-DiMethylHeptan*. Usually the positions are sorted in ascending order – this is one possible static semantic rule. Between two numbers a comma is added to avoid ambiguous meanings if the positions larger than 9.

To formulate this in a grammar we have the situation, that position numbers can occur multiple times on the same branch term. So at least one production rule for branches needs to create a non terminal symbol. We can isolate the parts for the position information from the symbol $B \rightarrow pmb$ in a new symbol C. p and m will be moved to C, so that C is now $C \rightarrow pm$. A second C, that creates itself, is necessary for multiple position information. This second C could be: $C \rightarrow poC$ (o represents here the comma char). *Or in more simple words: we need to define production rules, they can create itself to make loops possible.*

With the prefix, that summarizes multiple branches (we call it d), we have a similar situation like with M. This prefix d is an optional symbol, so we need an additional production rule to achieve this. An additional B seems to be a good choice.

$$
\begin{aligned}
T &= \{c, p, m, b, o, d\} \\
N &= \{S, B, M, C\} \\
P &: \\
S &\rightarrow M \\
M &\rightarrow Bc \\
M &\rightarrow c \\
B &\rightarrow C \\
C &\rightarrow pmb \\
C &\rightarrow poC \\
C &\rightarrow mdb
\end{aligned}
$$

Symbol	Description
c	Main chain
p	position
m	minus char
b	branch
o	comma char
d	summary prefix

Figure 2.5: CFG with multiple branches and same length. Several usage of $C \rightarrow poC$ creates successive positions.

The word *2,3-DiMethylHeptan* can be produced with: $S(M(C(po, C(po, C(mdb))), c))$

In theory we have the grammar, which fulfills the requirements. But with the production $S(M(B(C(mdb)), c))$ the word **mdbc** is creatable. A word with a minus char, a prefix like „Tri“, a branch and the name for the main chain like „Hexan“. E.g.: *-TriMethylHexan*. Such a word makes no sense. So the grammar is too open. We describe more words than wanted. The reason for this is, that C contains multiple different productions; only with sense, if there are used in a specific order. This is a hint for missing non terminalsymbols. A generic C is also too simple.

One problem are the position information. We have here two different scenarios:

- One singe position with a minus sign; E.g. *2-*
- Multiple position information separated with a comma and with a minus sign at the end. E.g. *2,3,4-*

We cannot guarantee with our current approach, that a summary prefix will be followed, if we have multiple position information. For example: *2,3,4-Methyl...* is in our grammar although the summary prefix *Tri* is missing. The reason is, that we here focused only on the position information and we have no way to influence the using of the summary prefix. The is an recurring problem with loops in a grammar when the number of used loops influence part of the words outside of the loop content. The figure 2.6 shows the situation with the current P.

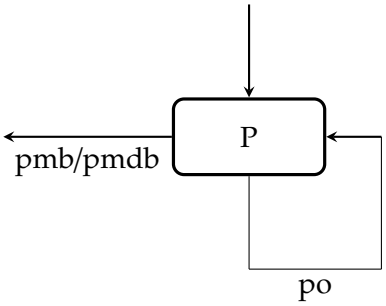

Figure 2.6: P can produce itself. *pmd* should only created, when at least one time P created itself.

P can create correct words. But it cannot determine, whether at least one time P created itself. So all words – with and without – summary prefix can be created. In a programming language we could use a flag or something similar to distinguish this two cases. In a CFG this approach is not usable, because we don't have variables or other kind of states, that can be altered. The only possibility to solve this issue is to introduce a new non terminal – here called P_2. For the sake of completeness we rename P to P_1.

P_2 can only be reached, if at least one times *po* was produced. So the usage of the non terminal P_2 itself holds the information, that a summary prefix must be used. Figure 2.7 shows the modified visualization of P.

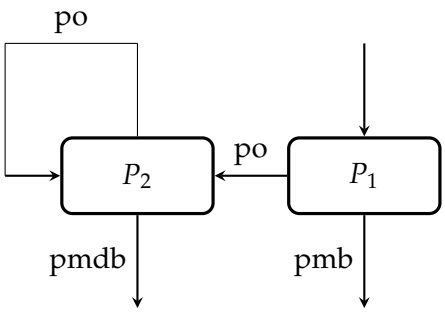

Figure 2.7: Modified P. The usage of P_2 gurantees, that at least one times *po* was created.

$$
\begin{aligned}
T &= \{c, p, m, b, o, d\} \\
N &= \{S, M, P_1, P_2\} \\
P&: \\
S &\rightarrow M \\
M &\rightarrow P_1 c \\
M &\rightarrow c \\
P_1 &\rightarrow pmb \\
P_1 &\rightarrow poP_2 \\
P_2 &\rightarrow poP_2 \\
P_2 &\rightarrow pmdb
\end{aligned}
$$

Symbol	Description
c	Main chain
p	position
m	minus char
b	branch
o	comma char
d	summary prefix

Figure 2.8: Adjusted CFG with P_1 and P_2.

We see, that our grammar uses now complete new production rules. One noticeable change are the production rules P_1 and P_2 with the same result poP_2. This is an usual way to define loops in a CFG. With the altered CFG, the production of the valid word *2,3-DiMethylHeptan* is also modified: $S(M(P_1(po, P_2(pmdb)), c))$.

2.5 Second order branches

In the chapter 2.3 and 2.4 first order branches are used. There are also so called „second order branches“. This is are nested branch, that is connected to a first order branch. Maybe the question can occur, how many nesting steps are possible. The nomenclature itself defines no limit[3]. Figure 2.9 is an easy example for a second order branch.

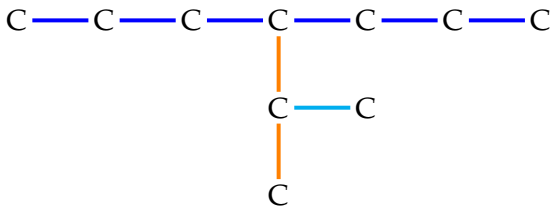

Figure 2.9: Example of a second order branch. The cyan connection is the second order branch.

[3]In normal cases chemicals with a deeper nesting than second order branches are not stable. So the real life gives a practical limit.

The last chapters explained how to handle a first order branch. Here: It has a length of two carbon atoms (→ *Ethyl*) on the fourth position → *4-Ethyl*. The IUPAC nomenclature handles a nesting with encapsulation of these in parentheses. The position of the nested branch is determined by the connection to the branch with the upper order (here *1*). And the length is described with the common rule for branches. The position of the upper order branch will be written before the opening parentheses to make clear on which upper branch the nesting is connected to. The solution for the branch in our example makes it more understandable: *4-(1-Methyl)EthylHeptan*.

In words: On a main chain with the length 7 – there is a first order branch at the fourth position with the length 2. And on this branch is on the first position a nested branch with the length 1.

The current CFG is not designed to describe nestings. An adjustment would be necessary. The example with nested branches shows, that already described elements are only a tiny part of the whole nomenclature. However for this project it sufficient.

2.6 Abstract syntax tree: An example

In the section 2 we used the tuple description of the CFG. Now we show here an example of an abstract syntax tree to simplify the visualization of a parsing process and focusing on structural details instead of describing the whole language with every intermediate step.

In figure 2.10 the main structure of the production is immediately noticeable. For example, that „Position 2 with length“ creates the position, the summary prefix *and* the branch. This is not an error – it is here a design choice.

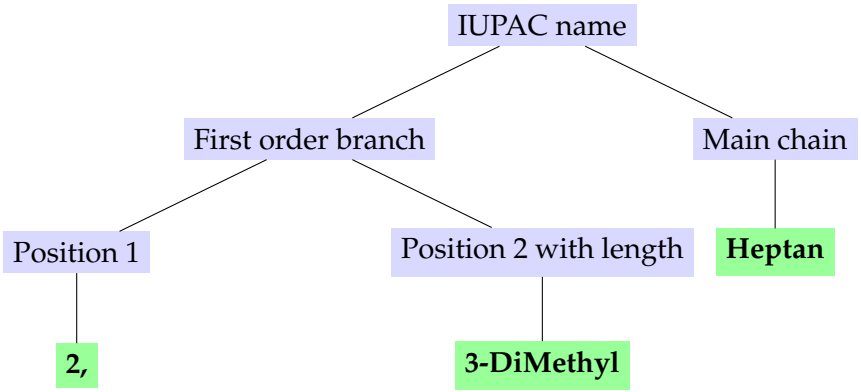

Figure 2.10: Abstract syntax tree of the word *2,3-DiMethylHeptan*.

3 Xtext

In this section, we show how the domain was implemented as a meta model and the resulting Xtext code. We also show examples from section 2 in the generated editor, where our domain-specific language is used.

3.1 Abstract Syntax: Meta Model

With meta modeling we model the abstract syntax of our DSL. We here designed our meta model (Figure 3.1) with the modeling language *Ecore* from the *Eclipse Modelling Framework (EMF)*. How to create such a meta model is described here[1]. Now, we want to look at the classes and how they're connected. We're also particularly interested in their attributes, especially these from type Enum.

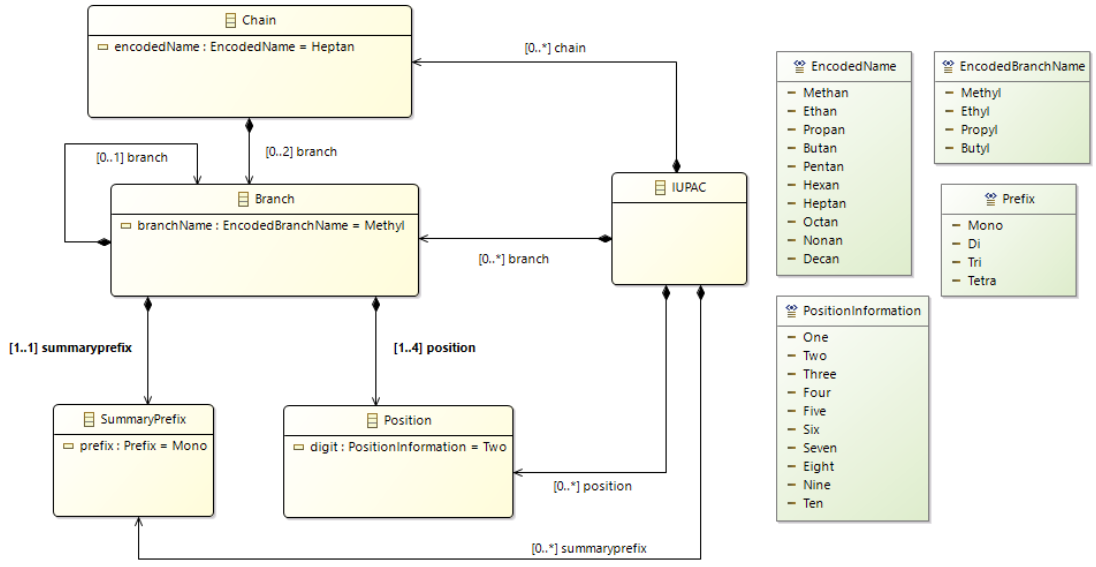

Figure 3.1: Meta model of the IUPAC nomenclature.

The *IUPAC* class serves as the meta model class, and each of the other classes (*Chain, Branch, SummaryPrefix, Position*) is a component of this class. This relationship is realized

[1] Appendix B: Eclipse Modeling Framework

through a *composition* with a cardinality of *0 to many*. Through these relationships, we aim to convey that numerous nomenclatures can be created.

We started describing in section 2.2, that the IUPAC nomenclature must have at least a main chain. In the class diagram, we assigned a different name to it. This decision was made, because we omit the semantic rules between the main chain and branch. We have chosen an Enum to store the *encoded names* of our *Chain class* – just like described in table 2.1. With this Enum, we have also established a framework for specifying the number of carbon atoms in a straight line that we intend to encode. Our goal is to encode a maximum of ten carbon atoms in a straight line without any branches. Now that we have established the foundation with the *Chain class*, the remaining classes are derived hierarchically from it.

Through the composition relationship from the *Chain class*, we constrained the *encoding of chains* to those having a maximum of two branches aligned. These branches are defined as *encoded branch names* and are also stored in an Enum.

Without any branches, we can not derive the prefix and the position information. That's why the two classes *SummaryPrefix* and *Position* are part of the *Branch class* via a composition. From Figure 2.8, we can derive, that there must be in total one prefix, when position information are introduced.

Alternatively, when defining a prefix based on this, you must specify the exact number of distinct position information. In the meta-model, the *1 to 4 relationship* indicates that up to four position information can be utilized in an IUPAC name, resulting in an Enum of up to four distinct prefixes. Initially, the number four may seem arbitrary, yet it aids in minimizing the verbosity of our domain-specific language.

Our position information is stored in an Enum, limited to ten different position names. This constraint aligns with our specification that the main chain can contain up to ten carbon atoms.

Since the syntax of Ecore requires to define default values, we have decided to take the IUPAC name from Figure 2.2. Last but not least, let's take a look at nesting, which was realised using the *Branch class*. This implements what appears in section 2.5 Here, however, we again restrict the fact that only one branch can be nested in a branch. For the time being, we assume that you can nest an infinite number of times.

3.2 Static Semantic: OCL Constraints

In section 2.4 static semantic was briefly touched upon.There are more static semantic rules describing how the IUPAC name should look like. To add those rules we used the textual editor of Ecore. There we implemented so-called OCL constraints to define the

static semantics of our DSL. However, we have implemented three OCL constraints in total. This was initially sufficient for our project.

We considered the following semantic rules:

1. The number of position information must be compatible with the prefix.
2. If there are more than one position information, then these must be in ascending order.
3. If there are several branches, they must be sorted alphabetically.

3.3 Concrete Syntax: Xtext Code

```
// Our adjusted Xtext code
grammar org.xtext.example.mydsl.MyDsl

import "http://IUPAC.ecore"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

IUPAC returns IUPAC:
//IUPAC class is mandatory to Object instantiation of EOL or
//Chain.
    {IUPAC}
    EOL*
    ((chain+=Chain)EOL+)*;

Chain returns Chain:
//Chain class is mandatory, to derive a correct IUPAC name.
    {Chain}
        ((branch+=Branch) ("-" branch+=Branch)*)?
  //Provide more branches with different lengths
        encodedName=EncodedName;

enum EncodedName returns EncodedName:
    Methan = "Methan" | Ethan = "Ethan" |
    Propan = "Propan" | Butan = "Butan" |
    Pentan = "Pentan" | Hexan = "Hexan" |
    Heptan = "Heptan" | Octan = "Octan" |
    Nonan = "Nonan" | Decan = "Decan";

Branch returns Branch:
    (position +=Position ("," position += Position)*) "-"
    ("("branch=Branch")")? //Nested branches
```

```
    summaryprefix=SummaryPrefix
    branchName=EncodedBranchName;

enum EncodedBranchName returns EncodedBranchName:
    Methyl = "Methyl" | Ethyl = "Ethyl" |
    Propyl = "Propyl" | Butyl = "Butyl";

SummaryPrefix returns SummaryPrefix:
    prefix=Prefix;

Position returns Position:
    digit=PositionInformation;

enum Prefix returns Prefix:
    Mono = "Mono" | Di = "Di" | Tri = "Tri" | Tetra = "Tetra";


enum PositionInformation returns PositionInformation:
    One = "1" | Two = "2" | Three = "3" | Four = "4" |
    Five = "5" | Six = "6" | Seven = "7" |
    Eight = "8" | Nine = "9" | Ten = "10";

//EOL means end of line.
//Here we realize that every IUPAC name is completed,
//when a new line begins or a comment is attached
//at the end of the IUPAC name.
EOL:
    NEWLINE | SL_COMMENT;

terminal SL_COMMENT:
    "//" !("\n"|"\r")* ("\r"? "\n")?;

terminal NEWLINE:
    ("\r"? "\n");
```

Figure 3.2: Our concrete syntax definition adjusted, after deriving from the IUPAC meta-model.

After implementing the meta-model and OCL constraints, our attention turned to refining the syntax. We utilized our meta-model to generate Xtext code, which we further customized. The syntax of the Xtext code is inspired from the Extended Backus-Naur Form(EBNF). For instance, we encoded position information as numerical values and adjusted the order of IUPAC names.

Each class from our meta-model finds representation in our Xtext code. The red-highlighted elements in Figure 3.2 denote our literals, reflecting the language's vocabulary. These literals are entered within the generated editor as you will see in section 3.4. The previously defined composition relation ensures a specific order for entering the literals. Introducing special characters such as ",", or "-", provide a more concrete representation of our domain. The cardinalities from the relations are represented in Xtext with symbols such as " =, *, or += ".

We also needed to introduce *EOL* as non terminal to reduce to terminal symbols. These are on the one hand the comment symbol "//" and on the other hand the newline symbols ("\r" or "\n"), which are indicating an empty word (ϵ). This separates the IUPAC names from each other in the editor, because We did not classify the literals of the IUPAC name as terminals. The comment symbol allows us to comment our code.

Like in context-free grammar , we can also replace in our Xtext code non terminals, with other non terminals, multiple non terminals or non terminals with literal symbols. In Figure 3.3 we have shown schematically how our Xtext code would look in context-free grammar and then we show how to derive a specific IUPAC name from it.

Classes:

`S`	→	`IUPAC | EOL`
`IUPAC`	→	`ChainEOL`
`Chain`	→	`BranchEncodedName | EncodedName`
`Branch`	→	`Branch-Branch`
`Branch`	→	`Position-SummaryPrefixEncodedBranchName | Position-(Branch)SummaryPrefixEncodedBranchName`
`Position`	→	`Position,Position`
`Position`	→	`PositionInformation`
`SummaryPrefix`	→	`Prefix`

Enumerations:

`EncodedName`	→	`Methan | Ethan | ...`
`EncodedBranchName`	→	`Methyl | Ethyl | ...`
`PositionInformation`	→	`1 | 2 | 3 | ...`
`Prefix`	→	`Mono | Di | Tri | ...`

End of line:

`EOL`	→	`// | ` ϵ

Figure 3.3: Our Xtext code in context-free grammar form. *S* is here the start symbol. A IUPAC name can be terminated with either // or ϵ.

Example:

S → IUPAC
→ ChainEOL
→ Chain
→ BranchEncodedName
→ BranchHeptan
→ Position-SummaryPrefixEncodedBranchNameHeptan
→ Position,Position-SummaryPrefixEncodedBranchNameHeptan
→ Positioninformation,Positioninformation-PrefixMethylHeptan
→ 2,3-DiMethylHeptan

Figure 3.4: From our "Xtext" grammar, we can derive this IUPAC name.

3.4 DSL: Editor

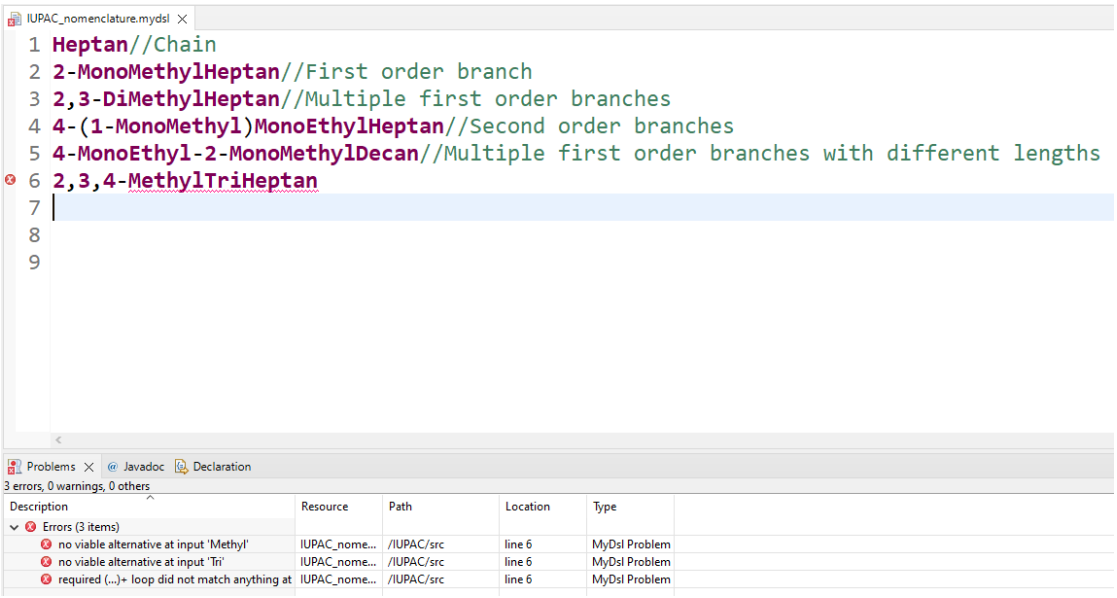

Figure 3.5: Generated editor with static semantics of 3.2 and default concrete syntax of 3.3. It also highlights the negative example and outputs the concrete error.

The generated Editor can do syntax highlighting, performs static validation and code completion, when you type. Figure 3.5 shows the Editor, which accepts the IUPAC names from our previous sections. It also detects the mistake from the last IUPAC name, where prefix and branch name are swapped.

3.5 Possible use cases

One obvious use case is the validation of given IUPAC names. In figure 3.5 is shown, that the generated editor indicates wrong names. Another idea could be an extended grammar with a grouping function. E.g.: The grammar accepts only names with a specific chain length.

4 Discussion and Conclusion

Although it is possible to generate corresponding Xtext code for the meta model shown in 3.1, such crude code is barely maintainable and expandable. Because we always need to update our meta model first. Otherwise newly added variables and data structures are unknown.

The generated Xtext code out of the Ecore model needed to be highly customized due the fact, that by default all terminals are handled as individual tokens without direct concatenation. In the original code the valid name *2-MonoMethylHeptan* was only valid as such: *2-Mono␣Methyl␣Heptan*.

Also the typical problem with results from parser generators and few meaningful error messages occurs here. In figure 3.5 the word *2,3,4-MethylTriHeptan* is wrong[1] because *Methyl* needs to be swapped with *Tri*. The given error messages „no viable alternative at input 'Methyl'“ and „no viable alternative at input 'Tri'“ are hardly usable, especially for novice and intermediate users of such a DSL. A more meaningful message could be here: „Missing summary prefix after list of positions.“.

In summary we made the experience with Ecore, that this technology is suitable for prototype- and test-DSLs. The generated Xtext code is not usable for productive use-cases due to a lot of adaptation work may be necessary. The fact that the code has to be completely rewritten if the model is changed reinforces our opinion that Ecore is hardly suitable for practical use. The generated Xtext code can at most be used as the basis for a DSL. On its own, Xtext is a fairly straightforward language for DSLs and is more useful for daily usage. Although the problem with the less expressive error messages is basically not solvable; it can be compensated by additional error production rules.

[1] The correct name is: *2,3,4-TriMethylHeptan*

5 Reflection

In this section the reflection questions will be answered. The following question list was given:[1]

1. Has the task or your understanding of the task changed over time? If so, how?
2. How appropriate was the planning of the project, especially the intermediate-deadlines? How often did the planning have to be adjusted? Why? What was your role in this?
3. What methods did you use to support the project planning and how did they work?
4. How did the coordination of the project work? What problems were there? How were they solved? What was your role in this?
5. To what extent were you sufficiently competent to cooperate? Which skills did you have to acquire in the course of the project? What did you learn learned from the study project?
6. What have you done to work as efficiently and reliably as possible in the project?
7. What would you do differently if you had to work on the task again with the experience at the end of the project?
8. What challenges came in terms of group dynamics in the team and how did you deal with them?

5.1 Answered questions: Dominik Habermann

1. The main task was in the whole project the same. However some details were changed over time. For example: Topics that should be explained in the tutorial were added and/or altered.

[1] Source of the questions: `https://moodle.ruhr-uni-bochum.de/pluginfile.php/1902788/mod_page/content/5/Reflexion.pdf`

2. The intermediate-deadlines were changed multiple times. One major problem was to find dates for the meetings. The reason was, that there were only few time slots on which everyone was available. My part of this situation was the long drive home. Due to the lack of stable mobile internet, it was not possible to join a meeting.

3. My method to support the project planning was to reserve exclusively fixed points of time in a week.

4. See answer 2 and 3: The main problem was finding a time to meet together. My part of problems in the coordination was, that I don't like to be available the whole time. Usually I mute my phone when I at home.

5. I was able to contribute my knowledge right from the start. I had to learn how to use the program called Eclipse and the Xtext framework; I hadn't worked much with this program until the project started. And the Xtext framework was completely unknown to me up to that point.

6. I gave my team member tutorials and examples of the background topic. Because of private projects, the topic was not new to me. For reliably I installed on my machine an instant messaging client, which I don't used before on this device, and put them to the auto-start list. See answer of question 4.

7. With the experience at the end of the project I would use a different way to finding times for meetings. For example, a more flexible selection of time periods would have helped right from the start.

8. Not too much can be said about the challenges in group dynamics. In summary, there were no real problems regard to this.

5.2 Answered questions: Masuthan Mathiyalakan

1. The requirements themselves have not changed. However, it was not entirely clear at the beginning in which direction the domain should go. There were also adjustments to the definition of the domain, to what extent we wanted to describe it in detail. Here and there, tasks were also adjusted, depending on the time frame.

2. The planning itself was sometimes not clearly structured. Both of us had regular meetings with our supervisor as well as regular group meetings in our core group, but sometimes, due to conflicts with other appointments, we had to cancel our meetings. Most of the time, we could rely on the appointments set by the supervisor. Occasionally, when one of us couldn't attend, the other person conducted the meeting with the supervisor or suggested an alternative appointment when they were unavailable.

3. The communication took place via email. Meetings were held regularly via Zoom. We usually suggested times to each other and when we reached a consensus, we entered the meeting in the calendar.
4. Dominik Habermann had suggested the IUPAC nomenclature domain. Consequently, he focused more on the definition of the domain itself. Since we were supposed to develop a DSL in Xtext, I concentrated on the creation of the meta modelas well as the generation of Xtext code. Because the Xtext code has to be adapted to the domain, Dominik also helped me a lot in this case. The coordination was therefore fluent. We presented every week to each other, what we have done. If anyone had any further questions, someone answered as fast as possible. We had sometimes the problem to find a good date for a meeting with our supervisor. Thus we managed to send our status quo via mail.
5. As I am new to the DSL and IUPAC nomenclature field, I had to familiarize myself with these topics. I incorporated my experience from software development into the project. In addition to understanding the domain itself and creating DSLs using Xtext, I also gained knowledge on how to describe a domain in classes and create a metamodel. I also grasped the close integration of context-free grammar and IUPAC nomenclature.
6. Always informed about the current status and as well as possible, the tasks for the next meetings.
7. Create a clear timetable from the outset as to when which milestone or goal must be completed. Schedule project meetings throughout the semester before the project starts.
8. There were no major problems, so there is nothing worth mentioning.