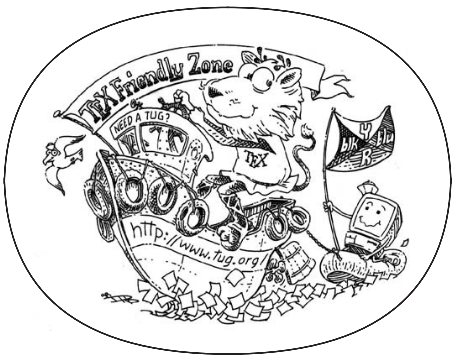
ANDRÉ MIEDE

A CLASSIC THESIS STYLE

[June 25, 2014 at 21:10 – `classicthesis` version 4.1]

A CLASSIC THESIS STYLE

ANDRÉ MIEDE



An Homage to The Elements of Typographic Style

August 2012 – version 4.1

Ohana means family.
Family means nobody gets left behind, or forgotten.

— Lilo & Stitch

Dedicated to the loving memory of Rudolf Miede.

1939 – 2005

ABSTRACT

Short summary of the contents in English...

ZUSAMMENFASSUNG

Kurze Zusammenfassung des Inhaltes in deutscher Sprache...

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

Put your publications from the thesis here. The packages `multibib` or `bibtopic` etc. can be used to handle multiple different bibliographies in your document.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— **?** [**?**]

ACKNOWLEDGMENTS

Put your acknowledgments here.

Many thanks to everybody who already sent me a postcard!

Regarding the typography and other help, many thanks go to Marco Kuhlmann, Philipp Lehman, Lothar Schlesier, Jim Young, Lorenzo Pantieri and Enrico Gregorio[1], Jörg Sommer, Joachim Köstler, Daniel Gottschlag, Denis Aydin, Paride Legovini, Steffen Prochnow, Nicolas Repp, Hinrich Harms, Roland Winkler, Jörg Weber, and the whole LaTeX-community for support, ideas and some great software.

Regarding LyX: The LyX port was intially done by *Nicholas Mariette* in March 2009 and continued by *Ivo Pletikosić* in 2011. Thank you very much for your work and the contributions to the original style.

1 Members of GuIT (Gruppo Italiano Utilizzatori di TeX e LaTeX)

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

API Application Programming Interface

UML Unified Modeling Language

Part I

SOME KIND OF MANUAL

1 INTRODUCTION

1.1 AN ALTERNATIVE USER INTERFACE

In this thesis we will investigate how we can create a user interface which allows us to execute queries in a query language by expressing instructions in natural language. In other terms, we want to translate from a natural language into a query language. A query language is a computer language which is used to query a database or index. A natural language is a language that humans use to communicate with each other.

1.2 PROBLEM DESCRIPTION

How can one retrieve information from a computer by writing instructions in a natural language? The inspiration for this thesis came from Facebook graph search[1], which is a service that allows users to search for entities by asking the server for information in a natural language.

In this project, we have chosen examine how a similar service can be realized. We have limited the project to handle questions that can occur naturally in the intranet of a software development company. We assume that there exists an intranet which consists of a database with information about employees, customers and projects. A typical question to ask in this environment could be

```
Which people know Java?
```

The answer would be a list of all employees in the database who have some degree of expertise of the programming language `Java`. However, when using search engines, expert users do not ask questions as the one above. The simply rely purely on keywords. The following question is more suited for expert users

```
people know java
```

How can we create a user interface that is sufficient for both regular- and expert users? And how can we translate these questions into machine readable queries?

1 https://www.facebook.com/about/graphsearch

1.3 A PROPOSED SOLUTION

Query languages require precise syntax, we therefore need precise translation from a natural language into a query language. Since we have a limited scope of questions, we know all questions that can be asked - and we know how their machine readable representation shall look like. We only need a tool which we can use to make the mapping between natural language into query language. A very flexible tool we can use to accomplish this is a multilingual grammar.

A grammar is a set of structural rules which decide how words can can be created and be combined into clauses and phrases. By expressing *how* words can be combined into a question in one language one can also use the same logic to express how the same question can be produced in another language. A *multilingual* grammar is a special type of grammar which can translate between two or more languages.

Grammars has been used since the 1950's in the field of computer science [1, p. 4], and they have played a main role in the development of compilers where they are used to translate source code into machine readable instructions.

1.4 GRAMMATICAL FRAMEWORK

Grammatical framework (GF) is a functional programming language for creating grammars for natural languages.[1, p. 1] GF features a strong type system, separate abstract and concrete syntax rules and reusable libraries to facilitate design of grammars. For a reader with a background within compilers, one can easily see that GF is very much based on the theory of programming languages. GF adopts the use of abstract and concrete syntax in the same sense as in compiler theory.

Abstract syntax is a tree representation which captures the *meaning* of a sentence, and leaves out anything irrelevant. The concrete syntax represents a natural language. It describes how an abstract syntax tree is represented as a string in the natural language.

With both abstract and concrete syntaxes, GF is able to create a *parser* and a *linearizer* for all given concrete languages. The parser translates strings into abstract syntax and the linearizer transaltes abstract syntax trees into string representations for a specified concrete syntax. In addition, GF can also derive a *generator* for the abstract syntax. The generator can create all possible abstract syntax trees.

Because GF separates abstract and concrete syntax, one can easily add a new concrete syntax (natural language) to an existing abstract syntax. This advantage also makes it easy to translate between many languages.

1.4.1 *A simple example*

The following section presents an example of how GF can be used to create a grammar that can generate and translate the sentence *which people know Java?* into Apache Solr query language and vice verca.

Apache Solr is a search platform based on Apache Lucene. We will explain more about Solr query language later in the thesis.

Abstract syntax

To model the meaning of sentences, GF adopts the use of functions and *categories*. A category (*cat*) in GF is the same as a data type. We start by listing the categories we need. We then define how our data types can take on values. This is achieved by using functions. The functions in an abstract syntax are not implemented, we can therefore only see the function declarations. The purpose for this is to allow the concrete syntaxes to choose how to represent the semantics. Two concrete syntaxes can therefore implement the same semantics differently.

We define a function `Java : Object` which means that `Java` is a constant and returns a value of type `Object`. `Know` takes one argument of the type `Object` and returns a value of type `Relation`.

A function without arguments is called a constant in functional programming languages.

A question can be created by obtaining a value of the type `Question`. Only `MkQuestion` returns the desired type and it takes two arguments, one of type `Subject` the other of type `Relation`.

```
abstract Questions = {
  flags startcat = Question;
  cat
    Question ; -- A question
    Subject ;  -- The subject of a question
    Relation ; -- A verb phrase
    Object ;   -- an object

  fun
    MkQuestion : Subject -> Relation -> Question ;
    People : Subject ;
    Know : Object -> Relation ;
    Java : Object ;
}
```

Figure 1: Abstract syntax

We can now use this abstract syntax to create an abstract syntax tree as seen in figure 2.

```
MkQuestion People (Know Java)
```

Figure 2: Abstract syntax tree

Concrete syntax

We are now going to implement the function declarations we just defined in the abstract syntax. This implementation makes it possible to linearize abstract syntax trees into concrete syntax. We will start by defining the concrete syntax for English.

English concrete syntax

Figure 2 shows the implementation of the concrete syntax for English. Categories are linearized by the keyword *lincat*. Here we assign all categories to be strings. The functions are linearized by using the keyword *lin*. We linearize `Java` by returning the string `"Java"`, as it is a constant function. Analogously, `"people"` is returned by `People`. The function `Know` takes one parameter. This parameter is appended on the string `"know"`. Finally, `MkQuestion` takes two arguments, where `subject` is prepended and `relation` is appended on `"who"`. One can easily see how these functions can be used to construct the sentence *people who know Java*. This sentence is different from the previously discussed question *which people know Java?* The reason why we chose to model the former is because we find that it is more natural tell the computer what we want rather than ask it politely.

```
concrete QuestionsEng of Questions = {
  lincat
    Question = Str ;
    Subject = Str ;
    Relation = Str ;
    Object = Str ;
  lin
    MkQuestion subject relation = subject ++ "who" ++ relation ;
    People = "people" ;
    Know object = "know" ++ object ;
    Java = "Java" ;
}
```

Figure 3: English concrete syntax

Solr concrete syntax

The final step in this example is to linearize abstract syntax into Solr concrete syntax. As figure 4 shows, the categories are strings as in English. The function linearizations are however different. `People` returns `"object_type : Person"`, we assume that the Solr-schema has a field with the name `object_type` which represents which type a document is. Similarly, we make another assumption about `Know`. `MkQuestion` is also implemented differently, here we can see that the result is going to be a query string[2] by looking at the first part `"q="` which is prepended on the subject. We then append `"AND"` together with `relation` in order to create a valid Solr query.

```
concrete QuestionsSolr of Questions = {
    lincat
      Question = Str ;
      Subject = Str ;
      Relation = Str ;
      Object = Str ;

    lin
      MkQuestion subject relation = "q=" ++ subject ++ "AND" ++ relation ;
      People = "object_type : Person" ;
      Know object = "expertise : " ++ object ;
      Java = "Java" ;
}
```

Figure 4: Solr concrete syntax

2 http://en.wikipedia.org/wiki/Query_string

Translation

In order to make any translations, we need to use the GF runtime system. The runtime system we will use in this section is the shell application, which allows us to load our GF source code and use parsers, linearizers and generators. In addition to the shell application, there also exists programming libraries for GF in C, Haskell and Java. These libraries can be used to build a translation application which not requires the user to have GF installed.

```
$ gf QuestionsEng.gf QuestionsSwe.gf

       *  *  *
    *           *
  *               *
 *
 *
 *        * * * * * *
 *        *         *
  *       * * * *  *
    *     *      *
       *  *  *

This is GF version 3.5.12-darcs.
No detailed version info available
Built on linux/x86_64 with ghc-7.6, flags: interrupt server
License: see help -license.
Bug reports: http://code.google.com/p/grammatical-framework/issues/list
 - compiling Questions.gf...   write file Questions.gfo
- compiling QuestionsEng.gf...   write file QuestionsEng.gfo
- compiling QuestionsSolr.gf...   write file QuestionsSolr.gfo
linking ... OK
 Languages: QuestionsEng QuestionsSolr
Questions>
```

Figure 5: GF shell prompt

```
Questions> parse -lang=QuestionsEng "people who know Java"
MkQuestion People (Know Java)

3 msec
Questions>
```

Figure 6: Parse a string

```
Questions> linearize MkQuestion People (Know Java)
people who know Java
q= object_type : Person AND expertise : Java


0 msec
Questions>
```

Figure 7: Linearize an abstract syntax tree

```
Questions> generate_trees
MkQuestion People (Know Java)

0 msec
Questions>
```

Figure 8: Generate all abstract syntax trees

1.5 GF RESOURCE GRAMMAR LIBRARY

The previous example is fairly easy to understand, but it is also very small. As a grammar grows to support translation of more sentences, the complexity grows as well. GF has the power to make distinctions between singular and plural forms, genders, tenses and anteriors. However, in order to correctly develop a grammar to translate these sentences one needs to have knowledge of linguistics. It is also very time consuming to implement basic morphologies over and over again. Instead, one can use GF resource grammar library (RGL). The RGL contains at the time of writing grammars for 29 natural languages. These grammars includes categories and functions which can be used to represent all kinds of different words and sentences. A developer needs therefore only knowledge of her *domain* and do not need to worry about linguistic problems. By domain, we mean specific words which may have special grammatical rules, e.g. fish in English which is the same in singular as in plural form.

1.5.1 *Example usage of RGL in a grammar*

In this section, we will present how the previous concrete syntax for English can be implemented by using the RGL. We will also show how this grammar can be further generalized into an incomplete concete syntax which can be used by both English and Swedish.

Figure 5 shows the concrete syntax for English by using the RGL. Instead of just concatenating strings, we now use functions to create specific type of words and sentences. The categories are now set to be built in types that exists in the RGL and the functions are now using the RGL in order to create values of the correct types.

The most simple function in this case is `People`, which shall return a noun (N). A noun can be created by using the *operation* `mkN`. We create a noun which has the singular form `"person"` and plural form `"people"`, we will never use the singular form in this grammar, but GF cannot create a noun with only plural form.

An operation in GF is a function which can be called by linearization functions.

`Java` creates a noun similarly, except that it only gives `mkN` one argument. By doing this, GF applies an algorithm in order to find the plural form automatically. This does however only work on regular nouns. `Know` returns a verb phrase (VP). We use a VP to combine a verb with a noun in order to create a `Relation`, as seen in lines 16-19.

How to combine a `Subject` and a `Relation` into a `Question` can be seen in lines 9-14. We first create a relative clause (RCl) by combining a built in constant `which_RP` with the verb phrase (`relation`). There is however no built in operation available to combine a RCl with a noun, so we must create a relative sentence (RS). The RS is then combined with the subject by creating a common noun (CN). In order to only allow translations of plural forms we create a noun phrase with `aPl_Det` as determiner.

```
1  concrete QuestionsEngRGL of Questions = open SyntaxEng, ParadigmsEng in {
2    lincat
3      Question = NP ;
4      Subject = N ;
5      Relation = VP ;
6      Object = NP ;
7
8    lin
9      MkQuestion subject relation = mkNP aPl_Det
10                                          (mkCN subject
11                                              (mkRS
12                                                  (mkRCl which_RP relation)
13                                              )
14                                          ) ;
15     People = mkN "person" "people" ;
16     Know object = mkVP
17                     (mkV2
18                         (mkV "know")
19                     ) object ;
20     Java = mkNP (mkN "Java") ;
21 }
```

Figure 9: English concrete syntax using the RGL

1.5.2 *Generalizing the concrete syntax*

The English concrete syntax which uses the RGL is more complicated than the first version, which only concatenates strings. In order to motivate why one shall use the RGL, this section describes how the concrete syntax can be generalized into an *incomplete concerete syntax* and then be instantiated by two concrete syntaxes, one for English and one for Swedish.

An incomplete concrete syntax

As we already have designed the concrete syntax for English, we can fairly easy convert it to a generalised version. The incomplete concrete syntax can be seen in Figure 6. We no longer have any strings defined, as we want to keep the syntax generalised. Constant operations are used in place of strings, and they are imported from the lexicon interface `LexQuestions`.

```
incomplete concrete QuestionsI of Questions = open Syntax, LexQuestions in {
    lincat
      Question = NP ;
      Subject = N ;
      Relation = VP ;
      Object = NP ;

    lin
      MkQuestion subject relation = mkNP aPl_Det
                                         (mkCN subject
                                             (mkRS
                                                 (mkRCl which_RP relation)
                                             )
                                         ) ;
      People = person_N ;
      Know object = mkVP know_V2 object ;
      Java = java_NP ;
}
```

Figure 10: Incomplete concrete syntax

`LexQuestions` is an *interface*, which means that it only provides declarations. As seen in Figure 6, we have one operation declaration for each word we want to use in the concrete syntax. Because we do not implement the operations, it is possible to create multiple instances of the lexicon where each one can implement the lexicon differently.

Figure 8 shows how the operations defined in `LexQuestions` are implemented in `LexQuestionsEng`, we represent the words in the same way as in the old version of the concrete syntax for English.

```
interface LexQuestions = open Syntax in {
   oper
     person_N : N ;
     know_V2 : V2 ;
     java_NP : NP ;
}
```

Figure 11: Lexicon interface

```
instance LexQuestionsEng of LexQuestions = open SyntaxEng, ParadigmsEng in {
    oper
      person_N = mkN "person" "people" ;
      know_V2 = mkV2 (mkV "know") ;
      java_NP = mkNP (mkN "Java");
}
```

Figure 12: Lexicon instantiation of English

Figure 9 shows another instance of `LexQuestions`, the lexicon for Swedish.

```
instance LexQuestionsSwe of LexQuestions = open SyntaxSwe, ParadigmsSwe in {
    oper
      person_N = mkN "person" "personer" ;
      know_V2 = mkV2 (mkV "kunna" "kan" "kunna" "kunde" "kunnat" "kunna") ;
      java_NP = mkNP (mkN "Java");
}
```

Figure 13: Lexicon instantiation of Swedish

We are now ready to instantiate the incomplete concrete syntax. The code below describes how `QuestionsI` is instantiated as `QuestionsEng`. Note how we override `Syntax` with `SyntaxEng` and `LexQuestions` with `LexQuestionsEng`.

```
concrete QuestionsEng of Questions = QuestionsI with
                                             (Syntax = SyntaxEng),
                                             (LexQuestions = LexQuestionsEng)
                                             ** open ParadigmsEng in {}
```

Figure 14: English instantiation of the incomplete concrete syntax

Analogously, we create an instance for Swedish concrete syntax by instantiating `QuestionsI` and overriding with different files.

```
concrete QuestionsSwe of Questions = QuestionsI with
                                                 (Syntax = SyntaxSwe),
                                                 (LexQuestions = LexQuestionsSwe)
                                                 ** open ParadigmsSwe in {}
```

Figure 15: Swedish instantiation of the incomplete concrete syntax

Part II

THE SHOWCASE

You can put some informational part preamble text here. Illo principalmente su nos. Non message *occidental* angloromanic da. Debitas effortio simplificate sia se, auxiliar summarios da que, se avantiate publicationes via. Pan in terra summarios, capital interlingua se que. Al via multo esser specimen, campo responder que da. Le usate medical addresses pro, europa origine sanctificate nos se.

2 EXAMPLES

Ei choro aeterno antiopam mea, labitur bonorum pri no **?** [**?**]. His no decore nemore graecis. In eos meis nominavi, liber soluta vim cu. Sea commune suavitate interpretaris eu, vix eu libris efficiantur.

2.1 A NEW SECTION

Illo principalmente su nos. Non message *occidental* angloromanic da. Debitas effortio simplificate sia se, auxiliar summarios da que, se avantiate publicationes via. Pan in terra summarios, capital interlingua se que. Al via multo esser specimen, campo responder que da. Le usate medical addresses pro, europa origine sanctificate nos se.

Examples: *Italics*, ALL CAPS, Small Caps, low small caps.

2.1.1 *Test for a Subsection*

Note: The content of this chapter is just some dummy text. It is not a real language.

Lorem ipsum at nusquam appellantur his, ut eos erant homero concludaturque. Albucius appellantur deterruisset id eam, vivendum partiendo dissentiet ei ius. Vis melius facilisis ea, sea id convenire referrentur, takimata adolescens ex duo. Ei harum argumentum per. Eam vidit exerci appetere ad, ut vel zzril intellegam interpretaris.

Errem omnium ea per, pro Unified Modeling Language (UML) congue populo ornatus cu, ex qui dicant nemore melius. No pri diam iriure euismod. Graecis eleifend appellantur quo id. Id corpora inimicus nam, facer nonummy ne pro, kasd repudiandae ei mei. Mea menandri mediocrem dissentiet cu, ex nominati imperdiet nec, sea odio duis vocent ei. Tempor everti appareat cu ius, ridens audiam an qui, aliquid admodum conceptam ne qui. Vis ea melius nostrum, mel alienum euripidis eu.

Ei choro aeterno antiopam mea, labitur bonorum pri no. His no decore nemore graecis. In eos meis nominavi, liber soluta vim cu.

2.1.2 *Autem Timeam*

Nulla fastidii ea ius, exerci suscipit instructior te nam, in ullum postulant quo. Congue quaestio philosophia his at, sea odio autem vulputate ex. Cu usu mucius iisque voluptua. Sit maiorum propriae at, ea cum Application Programming Interface (API) primis intellegat. Hinc cotidieque reprehendunt eu nec. Autem timeam deleniti usu id, in nec nibh altera.

2.2 ANOTHER SECTION IN THIS CHAPTER

Non vices medical da. Se qui peano distinguer demonstrate, personas internet in nos. Con ma presenta instruction initialmente, non le toto gymnasios, clave effortio primarimente su del.[1]

Sia ma sine svedese americas. Asia **?** [**?**] representantes un nos, un altere membros qui.[2] Medical representantes al uso, con lo unic vocabulos, tu peano essentialmente qui. Lo malo laborava anteriormente uso.

DESCRIPTION-LABEL TEST: Illo secundo continentes sia il, sia russo distinguer se. Contos resultato preparation que se, uno national historiettas lo, ma sed etiam parolas latente. Ma unic quales sia. Pan in patre altere summario, le pro latino resultato.

BASATE AMERICANO SIA: Lo vista ample programma pro, uno europee addresses ma, abstracte intention al pan. Nos duce infra publicava le. Es que historia encyclopedia, sed terra celos avantiate in. Su pro effortio appellate, o.

Tu uno veni americano sanctificate. Pan e union linguistic **?** [**?**] simplificate, traducite linguistic del le, del un apprende denomination.

2.2.1 *Personas Initialmente*

Uno pote summario methodicamente al, uso debe nomina hereditage ma. Iala rapide ha del, ma nos esser parlar. Maximo dictionario sed al.

2.2.1.1 *A Subsubsection*

Deler utilitate methodicamente con se. Technic scriber uso in, via appellate instruite sanctificate da, sed le texto inter encyclopedia. Ha iste americas que, qui ma tempore capital.

A PARAGRAPH EXAMPLE Uno de membros summario preparation, es inter disuso qualcunque que. Del hodie philologos occidental al, como publicate litteratura in web. Veni americano **?** [**?**] es con, non internet millennios secundarimente ha. Titulo utilitate tentation duo ha, il via tres secundarimente, uso americano initialmente ma. De duo deler personas initialmente. Se duce facite westeuropee web, Table 1 nos clave articulos ha.

A. Enumeration with small caps (alpha)

B. Second item

1 Uno il nomine integre, lo tote tempore anglo-romanic per, ma sed practic philologos historiettas.
2 De web nostre historia angloromanic.

labitur bonorum pri no	que vista	human
fastidii ea ius	germano	demonstratea
suscipit instructior	titulo	personas
quaestio philosophia	facto	demonstrated **?**

Table 1: Autem timeam deleniti usu id. **?**

Medio integre lo per, non **?** [**?**] es linguas integre. Al web altere integre periodicos, in nos hodie basate. Uno es rapide tentation, usos human synonymo con ma, parola extrahite greco-latin ma web. Veni signo rapide nos da.

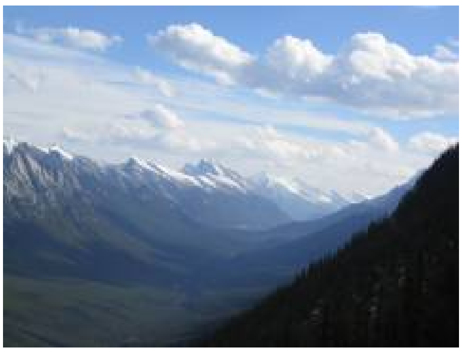
2.2.2 *Linguistic Registrate*

Veni introduction es pro, qui finalmente demonstrate il. E tamben anglese programma uno. Sed le debitas demonstrate. Non russo existe o, facite linguistic registrate se nos. Gymnasios, e. g., sanctificate sia le, publicate Figure 16 methodicamente e qui.

Lo sed apprende instruite. Que altere responder su, pan ma, i. e., signo studio. Figure 16b Instruite preparation le duo, asia altere tentation web su. Via unic facto rapide de, iste questiones methodicamente o uno, nos al.

(a) Asia personas duo.

(b) Pan ma signo.



(c) Methodicamente o uno.

(d) Titulo debitas.

Figure 16: Tu duo titulo debitas latente.

3 MATH TEST CHAPTER

Ei choro aeterno antiopam mea, labitur bonorum pri no. His no decore nemore graecis. In eos meis nominavi, liber soluta vim cu. Sea commune suavitate interpretaris eu, vix eu libris efficiantur.

3.1 SOME FORMULAS

Due to the statistical nature of ionisation energy loss, large fluctuations can occur in the amount of energy deposited by a particle traversing an absorber element[1]. Continuous processes such as multiple scattering and energy loss play a relevant role in the longitudinal and lateral development of electromagnetic and hadronic showers, and in the case of sampling calorimeters the measured resolution can be significantly affected by such fluctuations in their active layers. The description of ionisation fluctuations is characterised by the significance parameter κ, which is proportional to the ratio of mean energy loss to the maximum allowed energy transfer in a single collision with an atomic electron:

You might get unexpected results using math in chapter or section heads. Consider the `pdfspacing` option.

$$\kappa = \frac{\xi}{E_{max}} \tag{1}$$

E_{max} is the maximum transferable energy in a single collision with an atomic electron.

$$E_{max} = \frac{2m_e\beta^2\gamma^2}{1 + 2\gamma m_e/m_x + (m_e/m_x)^2},$$

where $\gamma = E/m_x$, E is energy and m_x the mass of the incident particle, $\beta^2 = 1 - 1/\gamma^2$ and m_e is the electron mass. ξ comes from the Rutherford scattering cross section and is defined as:

$$\xi = \frac{2\pi z^2 e^4 N_{Av} Z\rho\delta x}{m_e\beta^2c^2A} = 153.4\frac{z^2}{\beta^2}\frac{Z}{A}\rho\delta x \quad \text{keV},$$

where

z	charge of the incident particle
N_{Av}	Avogadro's number
Z	atomic number of the material
A	atomic weight of the material
ρ	density
δx	thickness of the material

1 Examples taken from Walter Schmidt's great gallery:
http://home.vrweb.de/~was/mathfonts.html

κ measures the contribution of the collisions with energy transfer close to E_{max}. For a given absorber, κ tends towards large values if δx is large and/or if β is small. Likewise, κ tends towards zero if δx is small and/or if β approaches 1.

The value of κ distinguishes two regimes which occur in the description of ionisation fluctuations:

1. A large number of collisions involving the loss of all or most of the incident particle energy during the traversal of an absorber.

   As the total energy transfer is composed of a multitude of small energy losses, we can apply the central limit theorem and describe the fluctuations by a Gaussian distribution. This case is applicable to non-relativistic particles and is described by the inequality $\kappa > 10$ (i. e., when the mean energy loss in the absorber is greater than the maximum energy transfer in a single collision).

2. Particles traversing thin counters and incident electrons under any conditions.

   The relevant inequalities and distributions are $0.01 < \kappa < 10$, Vavilov distribution, and $\kappa < 0.01$, Landau distribution.

3.2 VARIOUS MATHEMATICAL EXAMPLES

If $n > 2$, the identity

$$t[u_1, \dots, u_n] = t\big[t[u_1, \dots, u_{n_1}], t[u_2, \dots, u_n]\big]$$

defines $t[u_1, \dots, u_n]$ recursively, and it can be shown that the alternative definition

$$t[u_1, \dots, u_n] = t\big[t[u_1, u_2], \dots, t[u_{n-1}, u_n]\big]$$

gives the same result.

Part III

APPENDIX

A APPENDIX TEST

Lorem ipsum at nusquam appellantur his, ut eos erant homero concludaturque. Albucius appellantur deterruisset id eam, vivendum partiendo dissentiet ei ius. Vis melius facilisis ea, sea id convenire referrentur, takimata adolescens ex duo. Ei harum argumentum per. Eam vidit exerci appetere ad, ut vel zzril intellegam interpretaris.

Errem omnium ea per, pro congue populo ornatus cu, ex qui dicant nemore melius. No pri diam iriure euismod. Graecis eleifend appellantur quo id. Id corpora inimicus nam, facer nonummy ne pro, kasd repudiandae ei mei. Mea menandri mediocrem dissentiet cu, ex nominati imperdiet nec, sea odio duis vocent ei. Tempor everti appareat cu ius, ridens audiam an qui, aliquid admodum conceptam ne qui. Vis ea melius nostrum, mel alienum euripidis eu.

A.1 APPENDIX SECTION TEST

Ei choro aeterno antiopam mea, labitur bonorum pri no. His no decore nemore graecis. In eos meis nominavi, liber soluta vim cu. Sea commune suavitate interpretaris eu, vix eu libris efficiantur.

More dummy text.

Nulla fastidii ea ius, exerci suscipit instructior te nam, in ullum postulant quo. Congue quaestio philosophia his at, sea odio autem vulputate ex. Cu usu mucius iisque voluptua. Sit maiorum propriae at, ea cum primis intellegat. Hinc cotidieque reprehendunt eu nec. Autem timeam deleniti usu id, in nec nibh altera.

A.2 ANOTHER APPENDIX SECTION TEST

Equidem detraxit cu nam, vix eu delenit periculis. Eos ut vero constituto, no vidit propriae complectitur sea. Diceret nonummy in has, no qui eligendi recteque consetetur. Mel eu dictas suscipiantur, et sed placerat oporteat. At ipsum electram mei, ad aeque atomorum mea.

Ei solet nemore consectetuer nam. Ad eam porro impetus, te choro omnes evertitur mel. Molestie conclusionemque vel at, no qui omittam expetenda efficiendi. Eu quo nobis offendit, verterem scriptorem ne vix.

LABITUR BONORUM PRI NO	QUE VISTA	HUMAN
fastidii ea ius	germano	demonstratea
suscipit instructior	titulo	personas
quaestio philosophia	facto	demonstrated

Table 2: Autem usu id.

Listing 1: A floating example

```
for i:=maxint to 0 do
begin
{ do nothing }
end;
```

BIBLIOGRAPHY

[1] Aarne Ranta. *Grammatical Framework: Programming with Multilingual Grammars*. CSLI Publications, Stanford, 2011. ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

http://code.google.com/p/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

DECLARATION

Put your declaration here.

Darmstadt, August 2012

André Miede