funCode 1.0 Technical Report

Markus Lepper[1]
Baltasar Trancón y Widemann[1] [2]
Michael Oehler[3]

[1] semantics GmbH, Berlin
[2] Nordakademie Elmshorn
[3] Universität Osnabrück

Abstract

Harmonic analysis annotations can be represented in a computer model of a piece of music by plain text strings. But whenever automated processings like analysis, comparison or retrieval are intended, a formal definition is helpful. This should cover not only the syntactic structure, but also the semantics, i.e. the intended meaning, and thus adheres to the technique of *mathematical remodelling* of existing cultural phenomena. The resulting models can serve as a basis for automated processing, but also help to clarify the communication and discussion among humans substantially.

The funCode project proposes such a definition in four layers, which address different problems of encoding and communication (relation of symbol sequences to staff positions; combining functions; chord roots; interval structure and voice leading). Only one of them is specific to functional (Riemannian) theory and can possibly be replaced to represent scale degree theory.

The proposal is configurable to different interval specification methods and open to localisation. Syntax and semantics are defined and implemented using Prolog, which, due to its well-founded semantics, implies an unambiguous specification of the funCode formalism.

This text has been released 2022-02-16T11h34 .

The source files and possibly more recent releases shall be found in the web source code repository `https://sourceforge.net/projects/funcode`.

This text can be cited by the DOI found in `https://osnadocs.ub.uni-osnabrueck.de`

Contents

Listings

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Harmonic Analysis and its Encoding

Since over hundred years harmonic analysis systems developed in two different traditions: ”Roman number analysis” or ”scale degree theory” (German: ”Stufentheorie”) on one side, and ”functional analysis” on the other. Both traditions have been developed further by many theorists, mainly in Europe and North America. The former has been founded by Weber (1817) and most prominently employed by Schenker (1906) — it is prevailing in the US and the UK. The second is based on the theories of Hugo Riemann, but the form of the currently prevailing systems in Germany and other European countries is determined by the proposals of Grabner (1923) and Maler (1931), see Imig (1970) for a survey.[1]

The many different variant systems in both above-mentioned main schools have in common that each comes with its own *labelling system*. This is always collection of basic symbols together with rules for their syntactic combinations. A major use case is to construct *labels* by these means and to attach them to different time points in the flow of a piece of music. The intended meaning is that these labels shall describe (in one way or the other) the mental processes, feelings or ”situations” in human reception of this piece.

Another use case is a stand-alone sequence of these labels, meaning any piece of music which would match, or just standing for a particular formula of harmonic progression.

The upper part (a) of Figure 1.1 shows a rather typical and rather arbitrarily chosen example of such a harmonic labelling from a recent publication (Heetderks, 2015), describing the reception of a particular piece of music in Roman number theory. Typically, these labels are arranged below a (simplified) notation of the music in two dimensions: A vertical stack of *tracks* which extend horizontally and label the time points, as described above. This realisation is called *conventional two-dimensional arrangement (C2DA)* in the following. It has been added to the layout of publications often by hand-writing, even still in the late twentieth century. Nowadays it can be realised in digital publishing by graphical type setting commands, but no semantic digital encoding exists.

The simultaneity of tracks can serve different purposes. Prevailing use case is to show different ways of reception, i.e. different harmonic interpretations, which can be ”valid” or ”adequate” or even “effective” simultaneously, or where one of them reflects the feelings diachronously while another reflects the feelings in retrospect. This is common with *modulations*, i.e. harmonic sequences which alter the feeling for the tonic centre. But also different opinions of several authors can be mapped to different tracks, e.g. when comparing different famous analyses of famous works like the first eight bars of ”Tristan und Isolde”.

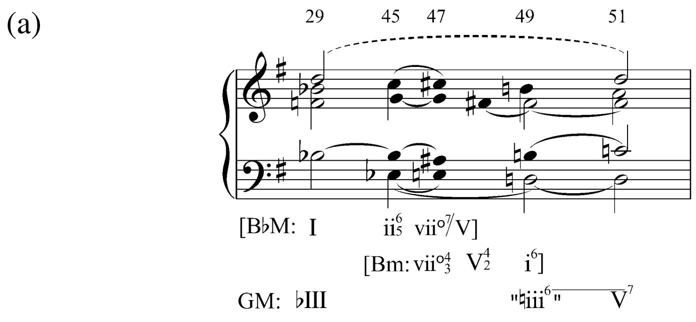
1.2 funCode, an Attempt for Standardisation

Most labels in Roman number theories and many in functional theories are graphically complicated combinations of sub- and superscripts, special character, strike-throughs etc. Nowadays these can easily be realised by digital type setting systems. But this is different from any *semantic* encoding. Only this would allow to automate comparison, sorting and retrieval of label sequences and translation between variants of notation systems. Esp. in the context of digital processing, such a definition is required: Therefore the authors of the recent “Annotated Beethoven Corpus” Neuwirth et al. (2018) added a formal definition of their scale degree labelling system, which specifies at least its syntax.

---

[1] Recently the North-America ”(Neo-)Riemannian” theories claim to be an offspring of functional analysis, but concentrate on particular aspects of Riemann’s theories, as criticised by Kopp (2011, footnotes 1 and 3).

In the last third of the twentieth century, alternative systems beyond this dichotomy arose, based on more mathematical analysis of sets of pitch classes, etc., but these are not in the scope of this project



(b)

B:T S56+ DD/79-

H:D/7_9- .8 t3_

G:tP Tg3_ D7

(c) `B♭M:I ii56 >vii˚7/V <{Bm:vii˚34 V24 i5}<<GM:♭III ~ ~ ♮iii6 V7`

(d) `B:T S56+ >DD/79- <{H:D/79- .8 t3_}<< G:tP ~ ~  Tg3_ D7`

Figure 1.1: Analysis from Schubert D 960 by Heetderks (2015), using sub-tracks.
(a) = staff notation and Roman number analysis from the publication; (b) = translation into functional style; (c) and (d) = application of the funCode L-1 linear encoding to (a) and (b).

But already much earlier, in pure inter-human communication, a lack of standardisation means a lack of clarity: Many functional encoding systems differ in details which have never been precisely specified, – in Roman number theory it will be similar. Even nowadays authors must still explicitly clarify their labelling system in advance. The first footnote in the example article (Heetderks, 2015) starts “In this article, major triads will be indicated by . . . ”

funCode is a computer language which reflects the way of labelling in functional harmonic theories. Due to its nature as computer language, it is unambiguously specified and thus reduces substantially the risk of misunderstandings and the effort for further clarification.

It is based on the labelling system developed by Grabner (1923) and Maler (1931), the most current system for functional annotations in Germany and other countries of the European continent, called *GM style* in the following.

funCode is intended (a) as a target language for translating existing analyses, to make them accessible for automated processing; (b) as a language onto which other symbol systems can be mapped, to unambiguously specify their semantics; and (c) as a language to be used in its own right, as a new variant of functional labelling.

Nevertheless, our project does *not* aim at inventing anything new from scratch. Contrarily, to be useful for the community, funCode is highly configurable, trying to be adaptable to as many different traditions of labelling as possible. Since all these instantiations are equally well-defined, automated translation should be feasible in most cases. As an attempt to cover different and widely used labelling systems, the funCode project adheres to the principles of *mathematical re-modelling* of established cultural techniques. (Lepper, 2021)

1.3 Design Principles of the funCode Approach

In the tradition of mathematical *re*-modelling, funCode does not aim at inventing anything new, but only tries to unify and simplify a whole bunch of historically grown annotation styles, which are currently in wide use — to make them accessible for automated processing and to clarify their semantics unambiguously for human discourse.

Not intending to invent something new, nevertheless in mathematical re-modelling always a certain “clean-up” takes places, which straightens out some “rough edges” of the historically grown and (from the standpoint of informatics) suboptimally defined language features. Most such corrections are *canonical extensions*: They remove a particular restriction which evolved in practice as “professional blinker” but which is not necessary from the mathematical viewpoint. *Computational thinking* as proposed by Broy (2011) means to learn from automated processing for the preciseness and handiness of human communication.

Thus some basic properties have been clearly marked as indispensable:

D-1 Transposition-invariant symbols: Every identical harmonic pattern must deliver identical symbol sequences, e.g. appearing in a key of c major, c sharp major and c flat major. The harmonic substance which shall be encoded must be readable independently from the transposition and always completely explicit.[2]

D-2 Syntax and semantics must be easy readable and writable by humans and machines.

D-3 Context information must be minimised to the ergonomically sensible. (E.g. `D7` always means a *minor* seventh, an abbreviation following convention and practical usability, but `T7-` and `T7+` must always be qualified explicitly.)

D-4 Localisation and further adaptability: Every harmonic notation system has its merits, and many scholars have been used to a particular system for decades, which they want to continue to use. Since in funCode all possible adaptations are formalised, complete automatic translation between these variants is feasible.

1.4 The Euler Net as Semantic Domain

Following the usual practice, a computer language is defined by its syntax and semantics. The former is specified using a standard grammar defining framework, like "recursive decent parsing" or "LALR tables" etc. The latter is defined by mapping the allowed syntactical constructs into some *semantic domain*, the elements of which represent their different intended meanings.

In case of harmonic labelling systems, very different kinds of semantics can be appropriate. For instance, a label can stand for "a particular position in the listener's mental map of harmonics". A very different semantics is modelled by "a set of pitch classes which realises the function indicated by the label". For instance `SpD` and `DDD` may represent the same set of pitch classes according to the latter semantics (namely a triad of A-major, when the labels are resolved relative to the tonic C), but stand for very different points in the harmonic inner world of the listener and different paths reaching this point, namely the dominant to the relative chord of the subdominant vs. the dominant of the dominant of the dominant.

Only the latter semantics, called *physical semantics* in the following, can currently be calculated by algorithms and are specified in the following chapter.

The domain into which the label sequences are mapped for defining their (physical) semantics is the *Euler net* (often called "Tonnetz"). This is a two-dimensional vector space of pitch or interval classes — these are pitches or intervals modulo octave. The first coordinate stands for the exponent with which (pure) fifths are applied, the second coordinate for pure (major) thirds. First proposed by Euler (1774), it has since been used in very different variants of music theory.

Figure 1.2 shows some typical variants of a graphical representation of the Euler net: The left version has orthogonal axes for the coordinates of the fifths and the major thirds, thus clearly showing the underlying construction. This form is especially useful when a third orthogonal axis for the pure seventh is added, see Vogel (1975). Its labelling uses one of the many variants to indicate the syntonic comma: The `E,` reached from `C` by a natural third "is lower" that the `E` reached by four fifths, which is again lower than the `E'` reached by going one third down and eight fifths up, etc.[3]

Nowadays the variant on the right side of the Figure is often preferred: Its didactic advantage is that every set of labels which looks like a minimal triangle indeed represents a triad, which is not the case in orthogonal variant. On the other hand, major and minor thirds seem to be on equal terms, which contradicts the construction.

Please note that very different variants of value systems and thus semantics can be expressed by the (initially mere syntactical) construct of an Euler net: Originally it was used to discuss tuning problems, and the axes stood for the application of the concrete above-mentioned intervals (= the proportions $3/2$ and $5/4$) to some arbitrarily chosen starting point, the points in the space thus representing concrete frequencies.

But these points can also be used to represent notated pitch classes; the net as a whole can also be used to represent classes of intervals. When representing intervals, the origin $(0, 0)$ stands for the prime interval; when representing frequencies or notated pitch classes, the meaning of $(0, 0)$ is fixed, but arbitrarily chosen. Furthermore,

---

[2] "Aus praktischen Erwägungen ist es wünschenswert, dass sich die Akkordstruktur (Intervallaufbau) unmittelbar aus der Funktions- oder Stufenbezeichnung ablesen läßt." (Hussong, 2005, pg. 99) ("For practical reasons it is desirable that the chord structure (interval structure) is immediately evident from the symbol for the function or scale degree.") In particular this should be *context free* and *transposition invariant*. Commonly, the function "tP" is written as "♭III" in C major, but as "♮III" in C sharp major, which clearly violates these principles. This is a typical example for the confusion between "external representation" (=note writing=syntactic sphere) and intended semantics, often found in music theory.

[3] Since the nodes of the Euler net represent classes of pitches modulo octave, the correct statement is of course "taking one representative each, with minimal distance, the representative of `E,` is lower than that of `E`" — etc.

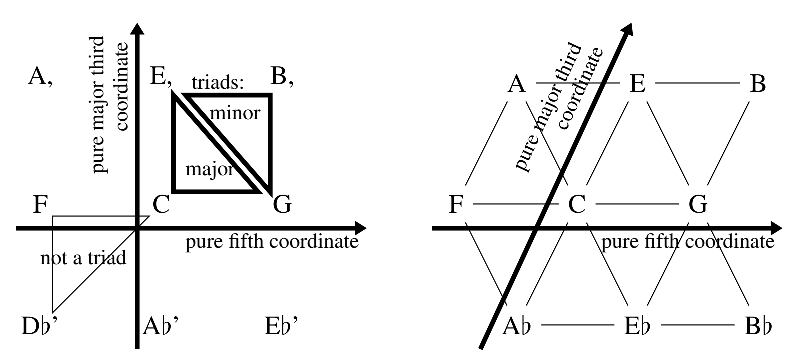


Figure 1.2: Graphical variants of the Euler net

different identities can be applied to the axes: In modern applications often a cycle is closed after 12 steps on the fifths axis and/or after 3 (major) thirds.[4]

In the following, the Euler space in unlimited (no modulos are defined), its points stand for pitch classes, and $(0, 0)$ is mapped (arbitrarily) to "C". The Euler space is used to model the *psychological concept* of harmonic position: While listening to a harmonic progression, the listeners move their feeling of a "current tonal position" and of the "current tonic centre" through the Euler space.

Please note that the axes use "pure" intervals on the conceptual level. This is independent of the concrete sounding frequency proportions, which in most cases nowadays will be equally tempered: The concept and the concrete experience of tonal spaces is based on pure intervals and works reliably and consistently, independent of intonation and tuning deviations.[5]

In the Prolog modelling, the constructor for coordinates in the Euler space is euler/2, where integers are allowed as arguments. Line 381 in the Listings defines the addition of two vectors, line 385 the summing up of a whole list of vectors.

1.5 Prolog for Execution and Specification

The following chapter presents the syntax and semantics of funCode as Prolog code. In contrast to most other programming languages, Prolog is a declarative language, the syntax of which closely follows that of Horn clauses, a fully mathematical device (Iso, 1996).[6] Thus the following text serves at least four purpose: It is an implementation which can be used with a Prolog interpreter (we use SWI-Prolog threaded, 64 bits, version 8.3.29) to parse, evaluate and translate funCode labelling (see also chapter 3). This allows (a) automated checking and processing of functional analyses, as well as (b) testing the specification itself for completeness and consistency. It is (c) a compact description of syntax and semantics for human readers and (d) an unambiguous specification in a strict mathematical sense.

Thus the Prolog specification of funCode does for functional ("Riemannian") labels the same as Nápoles López and Fujinaga (2020) do for the Roman number style, – indeed it does a bit more, since the "declarative way of reading" of the published Prolog code works as a well-founded public specification of the realised semantics.[7]

Physically, this technical report exists as one single *compound document* in the "**`d2d`**" format (Lepper and Trancón y Widemann, 2011). This text contains the description given in LaTeX, the core code, the code of all test routines (printed here as an appendix), and other minor text layers.

The licence of all contained texts is CC-SA-BY-NC.

In this print, the Prolog source text of the implementation core is given in the contained listing boxes, with consecutive line numbers, and referred to in the following explanations.

The web site contains the common d2d source text and different renderings of the different text layers.

---

[4] For a survey on literature about the Euler net see Cohn (2012, pg. 28, 29); see also Cohn (2011) and Gollin (2011).

[5] "Es sei genügend bewiesen, daß sich das geistig erfaßte Tonverhältnis oftmals von dem akustisch gegebenen Verhältnis unterschiede." Es gibt eine "Seelische Aktivität des Zurechthörens." ("It has been proven that the mentally perceived relation often differs from the acoustic relation". There is a "mental activity of adjusting in hearing.") (Vogel, 1975, pg. 147) See also the detailed discussion by Ploeger (1990, pg. 47pp)

[6] Therefore also the Java Virtual Machine specification (Lindholm et al., 2018) heavily employs Prolog rules.

[7] These both purposes conflict in some concerns: The practical use of the implementation requires detailed error messages and thus further parameters for their generation, which obfuscates the source text as a pure specification.

Chapter 2

Core Specification of funCode

2.1 Overall Architecture

funCode is a language which re-models different cultural techniques for labelling music with harmonic symbols, according to functional harmonic theory. The input format to funCode is a sequence of ASCII characters.[1] Its ”physical” semantics (as defined above) are reified by different finite maps, which relate particular time points in the flow of a piece of music to pitch classes and sets of pitch classes. These time points are called *score positions* in the following, and modelled by a contiguous sequence of integer numbers starting with 1. Score positions are taken as given — by which means the relation to the music or score is established is out of scope of the core project, but see section 3.4.

The syntax of the input is specified by Table 2.1. The calculation of the semantics, i.e. the above-mentioned maps is specified by the Prolog code published in this chapter.

As mentioned above, the motivation doing this kind of labelling lies in the “psychological” semantics, where each label describes a mental situation of reception. This is out of scope of our modelling project, see Imig (1970) and Gollin and Rehding (2011) for surveys on the different theories.

2.2 Implementation Principles and Housekeeping Routines

Basic implementation technique is to use the Prolog data base directly for holding the parsing results and thus all funCode information. In a first phase the text input is parsed and the results are stored by calls to **asserta**/2 etc. In a second phase inquiry functions can be applied to the data base to get the semantic data for a particular score position, see section 2.8 and Listing 2.28 below.

All calls to parsing and inquiry have as a first parameter an atom serving as ”document identifier”, to link the input and the outputs of separate parser calls. The Listing 2.1 defines in line 19 how the global data base is cleared for a particular document id. All atoms used as *data constructors* in this data base are underlined in the following listings. The predicates consistent/1 in Listing 2.3 test the correctness of their usage.

The parsing process is realised by a *Definite Clause Grammar (DCG)*, a standard in modern Prolog systems: The rules take the form

$$\textit{non-terminal}\ (\ \textit{arglist}\ )\ \longrightarrow \textit{body}.$$

The body is again a sequence of non-terminals, more precisely: a sequence of calls to clauses (with or without parameters) which represent non-terminals. Definitions of and calls to these non-terminals are *expanded* by the Prolog system on loading the source text by two further “invisible” arguments, representing the (rest) input to parse. Therefore the arities of the clauses defined by this notation are larger by two than visible.

In the body, sequence of “normal” clauses can be interspersed by setting them in braces “`{...}`”. After the expansion of heads and calls in the bodies, the DCGs are evaluated like any other Prolog code. This gives a strong expressive power to this very simple parsing formalism, because all back-tracking facilities are active as usual.

When parsing a funCode input, let “current score position” be the integer number to which the next recognised label shall be assigned. Each syntactic element (functional label, virtual root, sub-track, “space” or “idem” symbol, etc.) which is recognised by the code in the input source, is memorised by a data object stored to the Prolog data base. It is identified by the parameter Node:int, which is incremented after every such storage. The parsing procedures for all these items perform only *tail recursion*, therefore these numbers increase strictly in source text order, which makes debugging and testing easier — see the parameter Node in all the grammar rules items/8 starting

---

[1] Future extension to a larger Unicode character set is feasible but not necessary.

```
%! parse_atoms(++D:atom, ++Input:list of atoms) is det.
% The parameter D is here and in the following always a kind of ”Document Identifier”,
% which relates the inputs to the (globally stored) results of the parsing process.
% Node index and score positions are one(1)-based.
% Containing track of the top-level track is the non-existent ”node number 0”.
parse_atoms(D,Input) :-
fun_node_retract(D), phrase(parse_track(D, 1, 0, 1, []), Input),
\+ fun_error(D,_,_,_,_,_).

%! parse_string(++D:atom, ++Input:string) is det.
parse_string(D,Input) :- atom_chars(Input,List), parse_atoms(D,List).

:- dynamic(fun_node/5), dynamic(relative_root/3),
   dynamic(track_end/4), dynamic(fun_heureka/4),
   dynamic(error_pos/4), dynamic(fun_error/6), dynamic(fun_warning/6).

%! fun_node_retract(++D:atom) is det.
% Delete all data base entries for this ”Document Identifier”.
fun_node_retract(D) :-
    retractall (fun_node(D,_,_,_,_)), retractall ( relative_root (D,_,_)),
    retractall (track_end(D,_,_,_)), retractall (fun_heureka(D,_,_,_)),
    retractall ( fun_error (D,_,_,_,_,_)), retractall (fun_warning(D,_,_,_,_,_)).

% builds a map from node to node, by index, for later retrieving the tonal reference.
%! store_reference(++D:atom, ++FirstRelative:int, ++LastRelative:int, ++Basis:int) is det.
store_reference(_D, From,To,_) :-
    From > To, !.
store_reference(D,From,To,Target) :-
    under_write(D,From,Target),
    succ(From,F), store_reference(D,F,To,Target).
under_write(D,From,_Target) :- relative_root (D,From,_),!.
under_write(D,From,Target) :- assertz( relative_root (D,From,Target)).

%Builds a map from track (identified by node number) to last physical node number
% and last score position.
%! store_track_end(++D:atom, ++Track:int, ++Node:int, ++ScorePos:int) is det.
store_track_end(D, Track, Node,ScorePos) :-
    assertz(track_end(D,Track,Node,ScorePos)).
```

Listing 2.1: Global Data Base Management. Parsing and Inquiry Calls are linked by an atom acting as ”Document Identifier”.

analysis ::= *trackTitle*$^?$(*tonicCenter* **:**)$^?$(*funSeq* | **<*****{** *analysis* **}**)*
(**<**$^+$*analysis*)$^?$
trackTitle ::= **"** *any_character_not_doublequote** **"**
tonicCenter ::= *whiteKey* (**b** | **#**)*
whiteKey ::= **A** | **B** | **C** | **D** | **E** | **F** | **G**
funSeq ::= (*funPred*$^?$ *funSound funSucc*$^?$
| *funPred* **[***rootAndMode*$_{\mathrm{N}}$**]** *funSucc*$^?$ | **[***rootAndMode*$_{\mathrm{N}}$**]** *funSucc*
| *funSounds* | **-** | **~** | **>** | **!**)$^+$
funPred ::= **(***funSeq* **:**$^?$**)**
funSucc ::= **(:***funSeq***)**
funSound ::= *rootAndMode suppress*$^?$ *intervals*$^?$ | *intervals*
funSounds ::= *funSound* (**&***funSound*)*
rootAndMode ::= (**S** | **s** | **T** | **t** | **D** | **d**) (**P** | **p** | **G** | **g** | **D** | **d** | **S** | **s**)*
suppress ::= **/** | **//**
intervals ::= (**.** | *intervalDecorated*)$^+$
intervalDecorated ::= *interval* (**/** | **,** | *iModifier*$^?$ $\_\overset{??}{\wedge}\_^{?}$)$^?$
interval ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14**
iModifier ::= **+*** | **-***

Table 2.1: Complete syntax of all funCode layers, not regarding customisation/localisation

```
%! error/warning(++D:atom, ++Track:OPT int, ++Node: OPT int, ++ScorePos:OPT int,
%   ++Text:String ,++Args:OPT List) is  det.
% stores an error  message/warning with the given text and (partial)  coordinates.
error(D, Track,  Node,ScorePos,Text,Args) :-
    assertz(fun_error(D,Track,Node,ScorePos,Text,Args)).
warning(D, Track, Node,ScorePos,Text,Args) :-
    assertz(fun_warning(D,Track,Node,ScorePos,Text,Args)).

set_error_pos(D, Track,  Node,ScorePos) :-
     retractall (error_pos(D,_,_,_)),  assertz(error_pos(D,Track,Node,ScorePos)).
error(D, Text,Args) :-
    error_pos(D,Track,Node,ScorePos), assertz(fun_error(D,Track,Node,ScorePos,Text,Args)).
warning(D, Text,Args) :-
    error_pos(D,Track,Node,ScorePos), assertz(fun_warning(D,Track,Node,ScorePos,Text,Args)).
```

Listing 2.2: Global Data Base of Errors and Warnings. Parsing and Inquiry Calls are linked by an atom acting as ”Document Identifier”.

at line 110. (Only a weaker version of this fact is required by the operation of the parsing algorithm, namely that all nodes between two given nodes are addressable.) The nodes for tracks and track items are thus together directly identified by this "node number" (together of course with the above-mentioned document id.)

Contrarily, in all parsing rules and nodes the parameter `Score:int` holds the current score position which can jump back when a sub-track starts and back or forth when it ends.

Figure 2.1 shows an "object oriented" view to the prolog data base containing the parsing results: the nodes representing tracks and those representing track contents (sum nodes, idem nodes, etc.) are all wrapped into `fun_node/5` facts. Each such contains additionally the score position it refers to and the track it belongs to. For a track node the score position is the start of the track; a virtual sound "**[...]**" which serves as a relative reference point but does not appear at any score position in the music gets the special value `virtual`.

All inquiries for the data of a particular functional label work on the number of the representing node, see section 2.8.

Also each track is identified by the number of the node by which it is represented. A track node contains its name (if present) and its tonic centre (if present) in form of its source text and its Euler coordinate, see line 97.

The contents of a track follows immediately this representing node, i.e. the very first item in the track is represented by the subsequent node number, and both nodes carry the same score position.

A sum node contains a list of sound nodes, with a minimal length of one. Each sound node represents one functional symbol and contains the Euler coordinate of the root of the intended chord, the set of all its pitch classes, the Euler coordinate of its bass and melody tone (if specified), the source text of root and mode ("ram_source") and intervals ("int_source", "int_inherited"), and the code for root suppressing. (The first fields serve as convenience cache for the subsequent retrieval calls; the last three fields are needed for values to be inherited by the direct successor label.)

Additionally there is the relation defined by the fact `relative_root /3` which can link a node to another which defines the tonic context, relative to which its roots shall be resolved. This relation is realised as a collection of facts with document id and node numbers. `store_reference(D,From,To,Target)` in line 26 assigns the node "Target" as relative basis to all those nodes from "From" to inclusively "To" which are not yet assigned. Due to this relation the relation "containing track" contained in `fun_node/5` is *redundant* — it can be derived by searching. Nevertheless it is reified in the implementation, see the dashed arrows in Figure 2.1.

The top-level track is always represented by node number one. The predicate `track_end/4` additionally holds the last physical node number and covered by a track (including its sub-tracks) and the last score position, see line 37, line 123 and line 379. (Both values are stored *inclusively*.) These both values for track number 1 give the values for the whole specification.

2.3 Parsing Contexts

As mentioned above, the top-level parsing procedures get the document id as a parameter, the next node number to assign and the current score position, i.e. to which time point the next parsed sound label shall be attached.

Additionally it gets the currently growing track, identified by its node number.

In the top-level parsing process, the stack of the nested parsing contexts is managed explicitly, through the parameter `Stack: list`, i.e. is *not* mapped onto the Prolog execution stack. This is due to numerous crucial context conditions relating nodes independent from the parse tree structure, e.g. between adjacent sub-tracks.

When a first sound label has been processed, the parser changes its state because between adjacent sound labels there are additional inheritance relations. Therefore the "last parsed sound" becomes a new parsing parameter, see e.g. line 343

2.4 Parsing of Tracks and Sub-Tracks

Listing 2.4 shows the code of the top-most parsing rule which defines the global structure of a track: An optional name, an optional tonic centre (i.e. the tonic relative to which the following functional symbols shall be resolved) and a sequence of items, which are functional labels, relative regions, tab stop settings, sub-tracks etc. This corresponds roughly to the rule for the non-terminal *analysis* in Table 2.1. The parsing of these items is realised by the code in Listing 2.4 and Listing 2.9, calling some auxiliary routines. (The parsing of sound items has some more state parameters and is realised in Listing 2.10 and described later.)

Figure 1.1 lines (a) and (b) from above show realistic examples of the C2DA of sound labels, organised in vertically stacked and horizontally extending tracks; Figure 2.2 shows accordingly all four axes ("layers") supported in funCode: L-1 is the sequence of tracks; in each of these L-2 is the sequence of labels. One label can consist of more than one functional symbols, to represent compound sounds, which is axis L-3, and from one label to

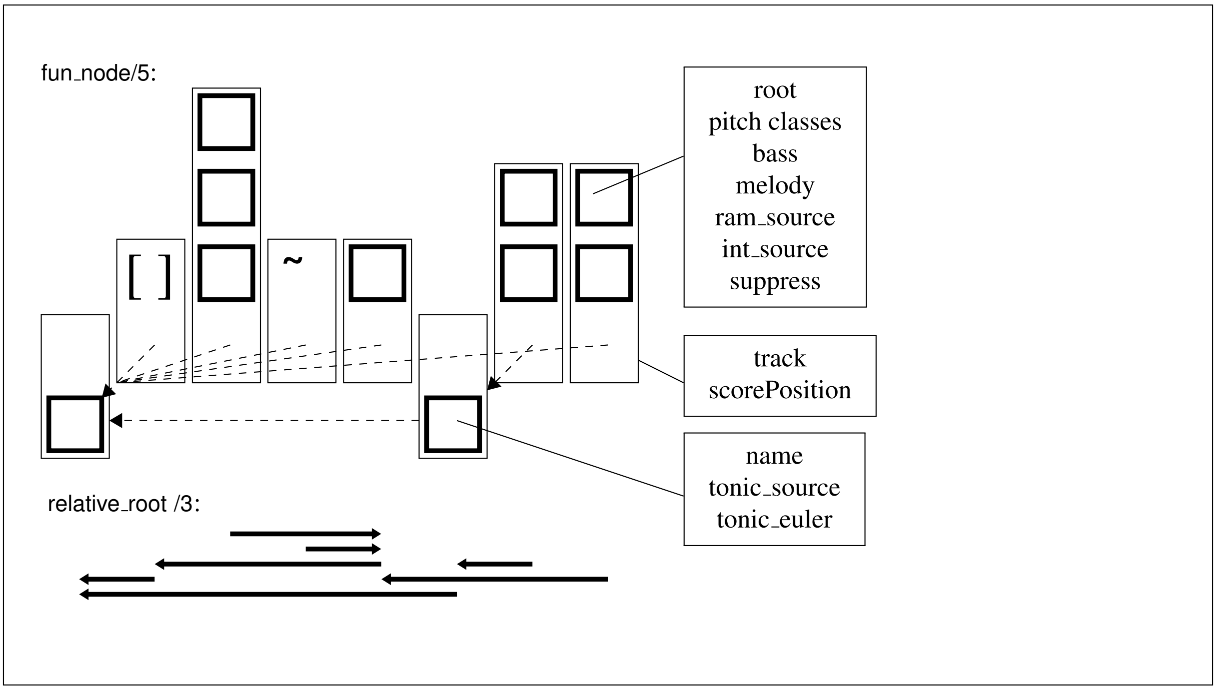


Figure 2.1: Data base of Parsing Results, "object style" view

the other the same root (i.e. a repetition of basically "the same chord") can exchange one chord component (= sounding interval) by another, using axis L-4. Lines (c) and (d) in Figure 1.1 and the last line in Figure 2.2 show how these C2DAs are encoded in funCode.

Let *sub-track* be a track which inherits vital parameters from another track, called its *parent track*. In C2DA the parent track is found by going upward from the start of the sub-track to the first reached non-empty track. In funCode sub-tracks are constructed by the following rules:

- The operator ">" sets a tab stop at the current score position.
- A pair of braces "`{...}`" encloses the source text which defines a sub-track. It starts its labelling with the current score position. After the contents of the braces has been translated, the parsing process is resumed for the interrupted track definition as if the braces and their contents were not present. The interrupted track is the sub-track's parent track. The sub-track inherits certain context conditions which have been valid at the parsing state of the parent track when the sub-tracks source code begins.
- When the opening brace is preceded by one or more "<", then the sub-track starts its labelling not at the current score position, but at an earlier tab stop. The number of "<" signs indicates how many tab stops are skipped backwards.
- One or more "<" signs without an opening brace start a new sub-track without the possibility to resume its parent track. This has been realised as an own special case because it is quite frequent in practical applications, when describing tonal *modulations*.

The parsing of these symbols is integrated into the parsing of the *items*, which are the contents of a track, see the code starting at line 110.

Whenever a new sub-track is started, a node entry is generated in the Prolog data base (see line 97) and the identifying number of this node is passed in the argument Track to the parsing procedures, by which all further generated data base entries will be linked to this track, see e.g. line 268.

Figure 2.3 shows examples of funCode sources and the intended tracks.

The first example shows a sub-track with an explicit new tonal centre **e:E**. Any possible (sub-)sub-track contained therein must never "start earlier", because then this context would be inherited by the above-mentioned funCode rule, but not visible in C2DA. Therefore we decided that no tab stop is inherited by a sub-track from its parent track. This prevents (sub-)sub-tracks starting left of the sub-track.

Tab stops to the right of the starting point of the sub-track *could* be inherited, e.g. in the second example the tab stop before **E** to the sub-track starting with "**<<{h:H...**". But such a tab stop could only become sensibly active not before the current score position has overtaken it, here: not before the **J** had been parsed. Therefore the meaning of the "<" operator would become context dependent, which could become very confusing for the users.

```
consistent(track ([], Center_source,Center_euler)) :-
      !, consistent(track("dummy", Center_source, Center_euler)).
consistent(track(Name,Center_source,euler(Q,T))) :-
      !, string(Name), Center_source=[_|_], integer(Q), integer(T).
consistent(track(Name,[],[])):-
      !, string(Name).

consistent(sum(SS)) :-
      !, is_list (SS), SS = [_|_], maplist(consistent_sound, SS).

consistent(sound(euler(Q,T),Pitches,Bass,Mel,Ram_source,Int_source,Int_inherited,Suppress)) :-
      !, integer(Q), integer(T), is_list (Pitches),
      integer_or_undef(Bass), integer_or_undef(Mel),
       is_list (Ram_source),is_list(Int_source), is_list ( Int_inherited ),
      member(Suppress, [0,1,2]).

consistent( virtual (Source,euler(Q,T))) :- !, is_list (Source), integer(Q), integer(T).

consistent( interrupted_track (T,S,TT)) :- !, integer(T), integer(S), is_list (TT), maplist(integer,TT).
consistent( left (I)) :- !, integer(I).
consistent( right (I)) :- !, integer(I).

consistent_sound(S) :- functor(S, sound, _), consistent(S).
integer_or_undef(undef) :- !.
integer_or_undef(I) :- integer(I).
```

Listing 2.3: Type and Consistency Tests on the Data Base "Item" Entries and on Auxiliary Data Structures.

Therefore no tab at all is inherited by a sub-track. Only the start point of each track (which can also be regarded as an inherited information) is included in the initial set of tab stops in every track, see line 98.

Please note that the graphic layout in the last lines of Figure 2.3 only represents the intended horizontal positions of the track items and is not a valid C2DA: The last track "`K L`" seems to inherit its context from position "`J`" but indeed inherits from "`E`"; it is a sibling and not a sub-track of the second track. The problem of a sensible layout is dealt with in section 3.5.

Every track starts with an optional trackNameSpec, an arbitrary character sequence. This can serve as an identifier, but it may also specify the roles of the different tracks — this is out of scope of funCode 1.0.[2]

This is followed by an optional tonicCenterSpec. This gives the tonal centre relative to which all following functional symbols in the sound labels will be evaluated. Major or minor mode is not included in this data, because all interval sizes are given explicitly, see section 2.7 below.

The topmost track *must* specify a tonic centre; all sub-tracks without a tonic centre inherit it from their parent tracks. This is checked not before the inquiry phase, see section 2.8.

Please note that the corresponding non-terminal *tonicCenter* in Table 2.1 allows arbitrary many accidentals: While in *notated music* at most one accidental is allowed for a key signature, this is different in *analysis*: Lewin (2006, pg. 193) discusses Lorenz and a "real" E♭♭ key and (pg. 196) even an "E♭♭♭ chord".

funCode 1.0 can be used with different semantics:

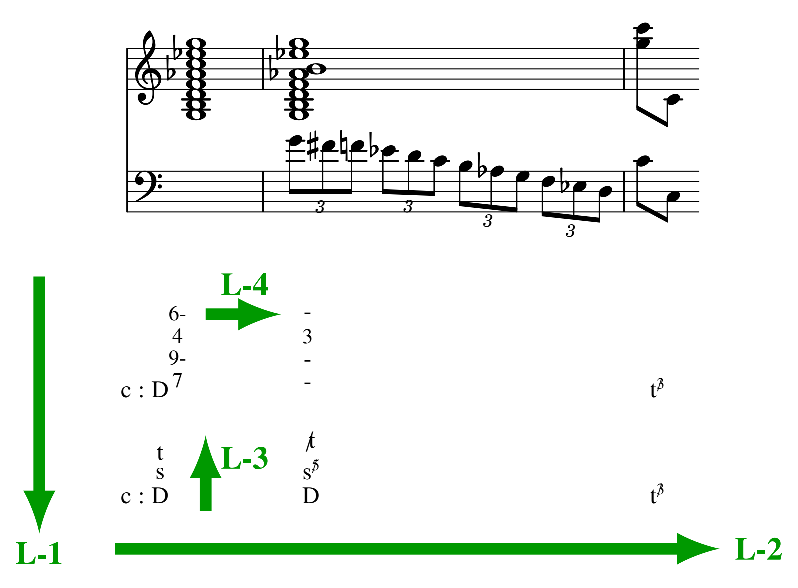
Not specifying any commata means that all resulting Euler coordinates are meant as classes module the syntonic comma; specifying comma values with the grammar rule comma (starting at line 187) allows to apply a semantics without this modulo, specifying the starting point for the label evaluation explicitly away from the central axis of fifths. This must be enabled by the global style parameter fun_initiaSyntonica, see Listing 2.18.

---

[2] For a catalogue of different roles of the tracks see the forthcoming journal publication.

```
%!parse_track(++D:int, ++Node:int, ++Track:int, ++Score:int, ++Stack:list)
% creates a new track-node as a sub-track of the current track (which may
% be == 0 for the top-most track!) and starts parsing of the contents
% (i.e. the opening "{" or "<" has already been consumed)
parse_track(D, Node, Track, Score, Stack) ⟶
    optWhiteSpace, trackNameSpec(Name),
    optWhiteSpace, tonicCenterSpec(Center_source,Center_norm,Center_acc,Center_comma),
    { set_error_pos(D,Node,Track,Score),
      ( (Center_comma=[_|_], \+fun_initiaSyntonica) ->
        error(D,
              "comma␣not␣allowed␣with␣tonic␣center␣by␣style␣parameter␣fun_initiaSyntonica",
              []);
        true),
      ( (Center_acc=[_,_|_], \+fun_accidensRepetendum) ->
            error(D, "more␣than␣one␣accidens␣requires␣fun_accidensRepetendum",[]); true ),
      append([Center_norm,Center_acc,Center_comma],Center_2),
      append([Center_source,Center_comma],Center_3),
       tonic_center_to_euler (Center_2, Center_euler),
      succ(Node,N),
      assertz(fun_node(D,Node,Track,Score,track(Name,Center_3,Center_euler))) },
    items(D,N, Node, Score, [Score], Stack).

optWhiteSpace() ⟶['␣'], !, optWhiteSpace().
optWhiteSpace() ⟶[].

%! items(++D:atom, ++Node:int, ++Track:int, ++Score:int, ++Tabs:list, ++Stack:list)
%@param D:atom      Document Identifier, linking parsing and inquiry functions
%@param Node:integer Id of the next physical node object to generate
%@param Track:integer Id of the currently growing track
%@param Score:integer Score Position for the next recognised label
%@param Tabs:list    Set of currently set tab positions
%@param Stack:list    Represents the nesting of tracks and relative regions
items(D, Node, Track, Score, Tabs, Stack) ⟶
    backtabs(P), ['{'], !,
    { set_error_pos(D,Node,Track,Score), poptabs(D, [Score|Tabs], P, StartScoreIndex) },
    parse_track(D, Node,Track,StartScoreIndex, [interrupted_track(Track,Score,Tabs)|Stack]).
items(D, Node, Track, Score, Tabs, Stack) ⟶
    backtabs(P), {P>0}, !,
    { set_error_pos(D,Node,Track,Score), poptabs(D, Tabs, P, StartScoreIndex) },
    parse_track(D, Node,Track,StartScoreIndex,Stack).

% assume that the contents of each track starts directly after the track's own node!
items(D, Node, Track, Score, _Tabs, [ interrupted_track (T,S,TT)|Stack]) ⟶
    ['}'], !,
    { succ(PN,Node), succ(Track,SN), succ(PS,Score),
      store_reference(D,SN,PN,Track), store_track_end(D,Track,PN,PS) },
    items(D, Node, T, S, TT, Stack).
items(D, Node, Track, Score, Tabs, Stack) ⟶
    ['>'], {Tabs=[Score|_]}, !,
    { warning(D, Node, Track, Score, "multiple␣set␣of␣same␣tab␣stop", []) },
    items(D, Node, Track, Score, Tabs, Stack).
items(D, Node, Track, Score, Tabs, Stack) ⟶
    ['>'], !, items(D, Node, Track, Score, [Score|Tabs], Stack).
items(D, Node, Track, Score, Tabs, Stack) ⟶
    ['!'], !,
    { store_heureka(D,Track,Node,Score) },
    items(D, Node, Track, Score, Tabs, Stack).
```

Listing 2.4: Top Level of Parsing: Tracks and Sub-Tracks



```
c:D79-46- ..3.  t3/ <c:D&s&t D&s5/&t/ t3/
```

Figure 2.2: All four axes of funCode employed to describe Mahler, II/1, m.327pp

```
a:A    B    >C    D    {e:E    F     G}   H    I   <{J  K}   L
⟹
a:A    B     C    D     H      I     L
                        e:E    F     G
             J    K

a:A  B    >C    D    >E    F     G    <<{h:H  I   J}   <K  L
⟹
a:A  B     C    D     E    F     G
           h:H  I     J
                      K    L
```

Figure 2.3: Examples of sub-track specifications. The upper case letters stand for sound labels; the lines before the arrow are source text; the two-dimensional arrangement after the arrows shows the intended score positions. (The second example is not a correct C2DA, see the text.)

2.5 Parsing of Sound Labels and Relative Regions

According to the grammar in Table 2.1, any track can contain sub-tracks as described above and relative regions by `(..)`, see section 2.5. Further there are the space symbol "`~`"and the idem symbol `-`. In conventional handwriting, the former corresponds to leaving the paper blank under the next score position, the latter to a long dash "—" meaning that the preceding formula continues ruling.

But the central contents of any track are the sound labels assigned to the score position. The first sound label recognised puts the parser into a different state, see lines 330 and 334: The rule sound_items/9 parses *adjacent* sound labels and thus implements the inheritance of intervals from chord to chord, see axis L-4 in Figure 2.2.

A feature much more frequently applied in functional analysis than in scale degree theory are the *relative regions*: Sequences of sound labels are evaluated not relative to the overall central tonal centre but to the root pitch of a neighbouring label, which thereby serves as a kind of "local tonal centre".

This notated using round parentheses "`(...)`". Conventionally the relative regions are related to their right neighbour in source text order, which is their successor in temporal score positions. funCode 1.0 implements a canonical extension which allows relations to their predecessor. This corresponds to the nonterminals *funSucc* and *funPred* in Table 2.1.

```
% backtabs(--NumOfTabs:int)
backtabs(N) ⟶['<'], !, backtabs(M), {succ(M, N)}.
backtabs(0) ⟶ [].

% poptabs(++D:DocId (for error msg), ++Input:list, ++index:int,--FoundTab:int)
poptabs(_D, [I | _], 0, I) :- !.
poptabs(D, [_ | Tabs], P, I) :- !, succ(O, P), poptabs(D, Tabs, O, I).
poptabs(D, [], _P, I) :- error(D, "undefined␣tab␣stop,␣too␣many␣<␣signs",[]),
                          I=700. %% only for allowing further error detection.

% trackNameSpec(--S:string or [])
trackNameSpec(S) ⟶['"'], !, trackNameChars(N, []), { string_chars(S,N) } .
trackNameSpec([]) ⟶[].
trackNameChars(N, N) ⟶['"'], !.
trackNameChars(R, N) ⟶[C], !, { append(N,[C],L) }, trackNameChars(R,L).

store_heureka(D,Track,Node,Score) :-
    fun_heureka(D,Track,_,_), !,
    error(D,Node,Track,Score, "More␣than␣one␣heureka/!␣operators", []).
store_heureka(D,Track,Node,Score) :-
    assertz(fun_heureka(D,Track,Node,Score)).
```

Listing 2.5: Auxiliary Routines for Parsing a Track

a)	`A (B C D) E`	Roots of B, C and D are defined relative to the root of E
b)	`A (B C D:) E`	Same as line a)
a)	`A (B C D) [E]`	Same as line a), but E does not sound = does not correspond to a score event
c)	`A (:B C D) E`	Roots of B, C and D are defined relative to the root of A
d)	`A ((B C) D) E`	Roots of B and C are relative to the root of D, which is relative to E
e)	`A (B (:C D)) E`	Roots of C and D are relative to the root of B, which is relative to E
f)	`A (((B) C) D) E`	Root of B is relative to C is relative to D is relative to E
g)	`A ((B) C) (D) E`	Same as line f)
h)	`A (B) (C) (D) E`	Same as line f)
i)	`A (B) ((C) D) E`	Same as line f)

Table 2.2: Relative Regions. The upper case letters stand for sound labels.

Table 2.2 shows typical examples.

Please note that the reference relation is independent of the parentheses and the parsing tree: lines f) to h) have the same meaning, and the format in h) is especially useful for sequences of falling fifth, i.e. of dominants.

The state of nesting relative regions including their starting score positions is memorised in the Stack parameter, interspersed with the state of nested tracks. The fact that a right-looking region has immediately been closed is also memorised by the special stack item rightJustClosed.

Whenever a *closing* parenthesis is reached, the stack is checked and shortened accordingly. Then all generated items from the start of the region to its end are linked to the succeeding or preceding score position by calling store_reference/4 — see lines line 299, line 327, and line 26. This procedure stores the relation only for those nodes which have not been processed earlier, so the information resulting from deeper nested relative regions is not overwritten.

Table 2.3 shows illegal inputs. Not all of them can be excluded with simple grammar rules, thus they are allowed according to the grammar in Table 2.1. The forbidden cases can be characterised as

- Line a) is excluded already by the syntax rules in Table 2.1.
- After a right-looking closing parenthesis no closing parenthesis may follow, neither right-looking (line c)) nor left-looking (line f)).

```
% −−K:source text −−L:norm pitch class −−M:accidentals −−N:commata
tonicCenterSpec(K,L,M,N) ⟶fullKey(K,L,M), comma(N), [:], !.
tonicCenterSpec(Source,L,M,N) ⟶whiteKey(K,L), accidental(M), {append(K,M,Source)}, comma(N), [:], !.
tonicCenterSpec ([],[],[],[])   ⟶ [].

:− dynamic(whiteKey/4).
:− dynamic(fullKey/5).

 select_locale_for_key (Loc) :−
      retractall (whiteKey(_,_,_,_)),
     atom_concat('whiteKey_',Loc,WKA),
     WCode =.. [WKA,A,B,C,D],
     assertz( (whiteKey(A,B,C,D) :− WCode) ),
      retractall ( fullKey (_,_,_,_,_)),
     atom_concat('fullKey_',Loc,FKA),
     FCode =.. [FKA,A,B,C,D,E],
     assertz( ( fullKey (A,B,C,D,E) :− FCode) ).

:−  select_locale_for_key ('EN').

fullKey_EN(_K,_L,_M) ⟶{ fail }.
whiteKey_EN( [K], [L])  ⟶[K],  {upcase_atom(K,L), member(L, ['A', 'B', 'C', 'D', 'E', 'F', 'G']) }, !.

accidental (['#'|L]) ⟶['#'],  !,  accidental_sharp ([], L).
accidental (['b'|L]) ⟶['b'],  !,   accidental_flat  ([], L).
accidental ([])  ⟶ [].
accidental_sharp(N, ['#'|L]) ⟶['#'],  !,  accidental_sharp(N,L).
accidental_sharp(N, N)   ⟶ [].
 accidental_flat (N, ['b'|L]) ⟶['b'],  !,   accidental_flat (N,L).
 accidental_flat (N, N)   ⟶ [].

comma([','|L]) ⟶[','], !, comma_down([], L).
comma(['\''|L]) ⟶['\''], !, comma_up([], L).
comma([])  ⟶ [].
comma_down(N, [','|L]) ⟶[','], !, comma_down(N, L).
comma_down(L, L) ⟶[].
comma_up(N, ['\''|L]) ⟶['\''], !, comma_up(N, L).
comma_up(L, L) ⟶ [].

key_pitch('A', euler( 3,   0)).
key_pitch('B', euler( 5,   0)).
key_pitch('C', euler( 0,   0)).
key_pitch('D', euler( 2,   0)).
key_pitch('E', euler( 4,   0)).
key_pitch('F', euler(−1,   0)).
key_pitch('G', euler( 1,   0)).

key_pitch('#', euler( 7,   0)).
key_pitch('b', euler(−7,   0)).
key_pitch(',', euler(−4,   1)).
key_pitch('\'', euler( 4,  −1)).

 tonic_center_to_euler   ([],[])   :−  !.
 tonic_center_to_euler (M,V) :−
     maplist(key_pitch,M,Terms),
     eusum_list(Terms, V).
```

Listing 2.6: Parsing the Tonic Centre of a Track

```
% (++Lang:atom, ++Text:String, ++Norm,++Acc:List of Chars)
def_locale_for_fullKey_upLowCap(Lang,Text,Norm,Acc) :-
    atom_concat("fullKey_",Lang,Nonterm),
    def_locale_for_key_upLowCap(Nonterm, Text, [Norm,Acc]).
def_locale_for_whiteKey_upLowCap(Lang,Text,Norm) :-
    atom_concat("whiteKey_",Lang,Nonterm),
    def_locale_for_key_upLowCap(Nonterm, Text, [Norm]).

% creates a rule  like ...
% fullKey_DE([g, i , s],  ['G'],  [#],  [g, i , s|A], B) :-  !,  B=A.
% ++Nonterm:Atom, ++Text:String, ++NormAcc:list
def_locale_for_key_upLowCap(Nonterm, Text,NormAcc) :-
    string_lower(Text, LText),
    string_chars(LText,ALText),
    def_locale_key([Nonterm, ALText |NormAcc]),
    string_upper(Text, UText),
    string_chars(UText,AUText),
    def_locale_key([Nonterm, AUText |NormAcc]),
    (  (ALText=[_| LTail ],  LTail  \=[])
    -> (AUText=[UH|_], ACText = [UH|LTail],
        def_locale_key([Nonterm,ACText|NormAcc])
      )  ;  true  ).
% ++Nonterm:Atom (like "whiteKey_NL") ++Input,Norm,Acc:List of Chars
def_locale_key([Nonterm, Input| NormAcc]) :-
    Head =.. [Nonterm, Input| NormAcc],
    Parser = (Head --> Input,  !),
    dcg_translate_rule(Parser, ParserCode),
    assertz(ParserCode).
```

Listing 2.7: Localisation of the Tonic Centre of a Track

```
:- def_locale_for_fullKey_upLowCap('DE', "Cis", ['C'],['#']).
:- def_locale_for_fullKey_upLowCap('DE', "Ces", ['C'],['b']).
:- def_locale_for_whiteKey_upLowCap('DE', "C", ['C']).

:- def_locale_for_fullKey_upLowCap('DE', "Dis", ['D'],['#']).
:- def_locale_for_fullKey_upLowCap('DE', "Des", ['D'],['b']).
:- def_locale_for_whiteKey_upLowCap('DE', "D", ['D']).

:- def_locale_for_fullKey_upLowCap('DE', "Es", ['E'],['b']).
:- def_locale_for_whiteKey_upLowCap('DE', "E", ['E']).

:- def_locale_for_fullKey_upLowCap('DE', "Fis", ['F'],['#']).
:- def_locale_for_whiteKey_upLowCap('DE', "F", ['F']).

:- def_locale_for_fullKey_upLowCap('DE', "Gis", ['G'],['#']).
:- def_locale_for_fullKey_upLowCap('DE', "Ges", ['G'],['b']).
:- def_locale_for_whiteKey_upLowCap('DE', "G", ['G']).

:- def_locale_for_fullKey_upLowCap('DE', "Ais", ['A'],['#']).
:- def_locale_for_fullKey_upLowCap('DE', "As", ['A'],['b']).
:- def_locale_for_whiteKey_upLowCap('DE', "A", ['A']).

:- def_locale_for_whiteKey_upLowCap('DE', "H", ['B']).
:- def_locale_for_fullKey_upLowCap('DE', "B", ['B'],['b']).
```

Listing 2.8: German Localisation of the Tonic Centre

```
C:sG  { sG tP }  ( T DD )
⟹
C:sG   (T DD)                    |  C:sG   ( T DD )
       sG tP                     |         sG  tP

C:sG  ( { T DD }  T  SP)
⟹
C:sG  (T SP)         | C:sG  ( T SP   )    |  C:sG  (T SP)
       T DD          |         T DD        |        (T DD)
                     |                     |

A   ( B  { C D E F } G H )  I
⟹
A   ( B   G   H )  I
          C   D    E   F
```

Figure 2.4: Different nestings of sub-track and relative region. The C2DA rendering must be printed and read very carefully w.r.t. the horizontal positions; different alternatives are shown. The last example uses abstract labels and shows a situation to avoid.

a)	`A (:B C D:) E`	Cannot look to both sides	line 305
b)	`A (B C:) (:D) E`	Cyclic dependencies between **`C`** and **`D`**	line 277
c)	`A (B (C D)) E`	Identical reference root for different nesting levels?	line 315
d)	`A (:(:B) C)`	Same as line c)	line 271
e)	`A ((:B) C) D`	No reference base for B	line 282
f)	`A (:B (C)) D`	No reference base for C	line 310

Table 2.3: Illegal Attempts for Relative Regions.

- Before a left-looking opening parenthesis no opening parenthesis may precede, neither left-looking (line d)) nor right-looking (line e)).
- a right-looking closing parenthesis may not be followed by a left-looking opening one (line b)).

Because the directions of information flow and of parser operation are different for all these cases, they must be recognised by the code explicitly; Table 2.3 shows the code line numbers.

A virtual function is enclosed in square brackets "`[...]`". This is a function which serves only as local reference point but does not correspond to a real sound in the labelled music. In many cases it corresponds to a sound the listener is *expecting*. It is modelled by $\mathit{rootAndMode}_{N}$ in Table 2.1 and by entering a virtual/2 item in the database, see line 268 and line 322.

A virtual item requires an adjacent relative region which refers to it, either from left or from right or from both sides. Therefore it is accepted only when a right-looking relative region has been closed immediately before (line 319) or when a left-looking region follows (line 264).

2.6 Parsing Root and Mode of a Chord

With the first recognised sound label (according to the parser procedure funsounds/4, see lines 330 and 334) the funCode parser switches its state by switching to the parsing procedure sound_items/9. This has one argument more than items/8, namely PreSounds, which is used to inherit root and intervals between two adjacent functional sound formulas.

Only in this state the "idem" item "`-`" can be used, which means that the same label is applied to the current score position as to its predecessor (line 356). The "space" item "`~`" can be applied inside (line 350) and outside this mode (line 370). It always corresponds to "paper left blank" in traditional handwriting. The parsing of both items carries on the default information in PreSounds for inheritance.

As soon as no more sound labels are recognised, the parser leaves this state, see line 365.

In functional analysis the combination of more than one functional sound into one sound label (which corresponds to one complex sounding chord at one particular score position) is not often but sometimes the adequate formula (see the lower track in Figure 2.2 and both examples in Figure 2.5). The parsing of sounds generates thus a sum/1 item in the data base which contains the list of the parsed sound descriptions of type sound/8.

```
items(D, Node, Track, Score, Tabs, Stack) ⟶
    ['['], { set_error_pos(D,Node,Track,Score) }, funRootN(D,Source,Euler) , [']'],
    optWhiteSpace, ['(', ':'], !,
    { succ(Node, N),
      assertz(fun_node(D, Node,Track,virtual, virtual (Source,Euler))) },
      items(D, N, Track, Score, Tabs, [ left (N)|Stack] ).

items(D, Node, Track, Score, Tabs,  [ left (Node)|Stack] ) ⟶
    ['(', ':'], !, {error(D, Node,Track,Score,
                          "adjacent␣left-looking␣parentheses",[])},
    items(D, Node, Track, Score, Tabs, [ left (Node),left (Node)|Stack] ).
    %% continue only to detect subsequent errors!

items(D, Node, Track, Score, Tabs,  [rightJustClosed|Stack] ) ⟶
    ['(', ':'], !, {error(D, Node,Track,Score,
                          "adjacent␣right-␣and␣left-looking␣parentheses",[])},
    items(D, Node, Track, Score, Tabs, [ left (Node)|Stack] ).  %% continue only to detect subsequent errors!

items(D, Node, Track, Score, Tabs,  [ right (Node)|Stack] ) ⟶
    ['(', ':'], !, {error(D, Node,Track,Score,
                          "adjacent␣right-␣and␣left-looking␣open␣parentheses",[])},
    items(D, Node, Track, Score, Tabs, [ left (Node),right(Node)|Stack] ).
    %% continue only to detect subsequent errors!

items(D, Node, Track, Score, Tabs, Stack) ⟶
    ['(', ':'], !, items(D, Node, Track, Score, Tabs, [ left (Node)|Stack] ).

items(D, Node, Track, Score, Tabs, [rightJustClosed|Stack]) ⟶
    ['('], !, items(D, Node, Track, Score, Tabs, [ right (Node)|Stack] ).

items(D, Node, Track, Score, Tabs, Stack) ⟶
    ['('], !, items(D, Node, Track, Score, Tabs, [ right (Node)|Stack] ).

items(D, Node, Track, Score, Tabs, [ right (Start )| Stack]) ⟶
     right_close , !,
    { succ(Last,Node), store_reference(D,Start,Last,Node) },
    items(D, Node, Track, Score, Tabs, [rightJustClosed|Stack] ).

 right_close ⟶[':', ')'], !.
 right_close ⟶[')'].

items(D, Node, Track, Score, Tabs, [ left (_)| Stack]) ⟶
    [':', ')'], !, {error(D, Node,Track,Score,
                          "relative␣region␣cannot␣look␣to␣both␣sides",[])},
    items(D,Node,Track,Score,Tabs,Stack). %% continue only for detecting subsequent errors!

items(D, Node, Track, Score, Tabs, [rightJustClosed, left (_N2)|Stack]) ⟶
    [')'], !, {error(D, Node,Track,Score,
                  "adjacent␣right-␣and␣left-looking␣close␣parentheses",[])},
    items(D, Node, Track, Score, Tabs, Stack).   %% continue only to detect subsequent errors!

items(D, Node, Track, Score, Tabs, [rightJustClosed, right (_N2)|Stack]) ⟶
     right_close , !, {error(D, Node,Track,Score,
                           "adjacent␣right-looking␣parentheses",[])},
    items(D, Node, Track, Score, Tabs, Stack).   %% continue only to detect subsequent errors!
```

Listing 2.9: Parsing of Relative Regions

```
items(D, Node, Track, Score, Tabs, [rightJustClosed|Stack]) ⟶
    ['['], { set_error_pos(D,Node,Track,Score) }, funRootN(D,Source,Euler), [']'], !,
    { succ(Node,N),
      assertz(fun_node(D, Node,Track,virtual, virtual (Source,Euler))) },
    items(D, N, Track, Score, Tabs, Stack).

items(D, Node, Track, Score, Tabs, [ left (Start)| Stack] ) ⟶
    [')'], !,
    { succ(Ref, Start), succ(PreNode, Node), store_reference(D,Start,PreNode, Ref) },
    items(D,Node,Track,Score,Tabs,Stack).

items(D, Node, Track, Score, Tabs, [rightJustClosed|Stack]) ⟶
    funSounds(D,Sounds), !,
    start_sound_items(D, Node, Track, Score, Tabs, Stack, Sounds).

items(D, Node, Track, Score, Tabs, Stack) ⟶
    funSounds(D,Sounds), !,
    start_sound_items(D, Node, Track, Score, Tabs, Stack, Sounds).

start_sound_items(D, Node, Track, Score, Tabs, Stack, Sounds) ⟶
    { succ(Node, J), succ(Score, S) ,
      assertz(fun_node(D, Node,Track,Score,sum(Sounds))) },
    sound_items(D, J, Track, S, Tabs, Stack, Sounds).

sound_items(D, Node, Track, Score, Tabs, Stack, PreSounds) ⟶
    { set_error_pos(D, Node,Track,Score) },
    funSounds(D,PreSounds, Sounds), !,
    { succ(Node, J), succ(Score, S) ,
      assertz(fun_node(D, Node,Track,Score,sum(Sounds))) },
    sound_items(D, J, Track, S, Tabs, Stack, Sounds).

sound_items(D, Node, Track, Score, Tabs, Stack, PreSounds) ⟶
    ['~'], !,
    { succ(Node, J), succ(Score, S) ,
      assertz(fun_node(D, Node,Track,Score,space)) },
    sound_items(D, J, Track, S, Tabs, Stack, PreSounds).

sound_items(D, Node, Track, Score, Tabs, Stack, PreSounds) ⟶
    ['-'], !,
    { succ(Node, J), succ(Score, S) ,
      assertz(fun_node(D, Node,Track,Score,idem)) },
    sound_items(D, J, Track, S, Tabs, Stack, PreSounds).

sound_items(D, Node, Track, Score, Tabs, Stack, PreSounds) ⟶
    ['␣'], !, sound_items(D, Node, Track, Score, Tabs, Stack, PreSounds).

sound_items(D, Node, Track, Score, Tabs, Stack, _PreSounds) ⟶
    items(D, Node, Track, Score, Tabs, Stack).
```

Listing 2.10: Parsing of Sound Items

$$
\begin{array}{llll}
X \in \{\mathtt{T},\mathtt{S},\mathtt{D},\mathtt{G},\mathtt{P},\uparrow\} & & x \in \{\mathtt{t},\mathtt{s},\mathtt{d},\mathtt{g},\mathtt{p},\downarrow\} & \\
\ldots X\mathtt{G}\ldots & \rightsquigarrow & \ldots X\mathtt{g}\uparrow\ldots & \\
\ldots X\mathtt{P}\ldots & \rightsquigarrow & \ldots X\mathtt{p}\uparrow\ldots & \ldots\left\{\begin{matrix}\uparrow\\ \downarrow\end{matrix}\right\}\left\{\begin{matrix}\mathtt{S}\\ \mathtt{s}\\ \mathtt{D}\\ \mathtt{d}\end{matrix}\right\}\ldots \;\rightsquigarrow\; \mathtt{error}(\text{“}\mathit{Superfluous}\text{”}) \\
\ldots x\mathtt{g}\ldots & \rightsquigarrow & \ldots x\mathtt{G}\downarrow\ldots & \\
\ldots x\mathtt{p}\ldots & \rightsquigarrow & \ldots x\mathtt{P}\downarrow\ldots &
\end{array}
$$

Table 2.4: Normalisation of Functions Codes

```
items(D, Node, Track, Score, Tabs, Stack) ⟶
    ['␣'], !, items(D, Node, Track, Score, Tabs, Stack).

items(D, Node, Track, Score, Tabs, Stack) ⟶
    ['~'], !,
    { succ(Node, J), succ(Score, S),
      assertz(fun_node(D, Node,Track,Score,space)) },
    items(D, J, Track, S, Tabs, Stack).

%% ASSSUME toplevel track = node #1 and its contents starts with 2.
items(D, Node, 1, Score, _Tabs, []) ⟶
    [], { succ(PN,Node), store_reference(D,2,PN,1),
          succ(PS,Score), store_track_end(D, 1, PN, PS)
          }.
```

Listing 2.11: Parsing of Miscellaneous Track Items and End of Input

There exist two parsing functions funSounds/4 and funSounds/5, without and with a preceding label to inherit from (Lines 405 and 410). Both step through the input and recognise functional symbols (according to the nonterminal *funSound* in Table 2.1), calling the parsers funSound/5 if there is a predecessor for inheriting from, or funSound/4 (line 422) if not.

The former comes in two variants: One covers the case that *rootAndMode* are present in the source text (line 438). In this case, inheritance of intervals is only defined if the maximum difference between source text of both roots is the character case of the last character of their source text, as checked by check_inherit_intervals (line 487).

The other case is that only intervals are present in the source text (line 454). In this case also root and mode are inherited, plus possibly further intervals.

The root pitch class and the mode (major/minor) of a function sound is parsed according to the nonterminal *rootAndMode* from Table 2.1, realised by the code starting in line 529.

For *rootAndMode* as well as for *intervals*, the *source text* is collected by the parser and stored in the data base item, for diagnosis and possible later reconstruction of the input text.

Then the sequence of functional symbols is *normalised* to make explicit the mode changes, see Table 2.4. During this normalisation process, superfluous mode conversions are detected and rejected. This can be controlled by the global style parameter “fun_emotioFugax”, see Listing 2.18.

Then the Euler value of the root is calculated (line 843) and is stored in the database node, only as a “convenience cache” for later retrieval, see section 2.8.

2.7 Parsing of Intervals

2.7.1 Interval Modifier Syntax, Inheritance and Defaults

Every function (the root of which is determined by the rules from the preceding section) can appear in the labelled music score as a concrete chord with different components. There are default components, added to any root implicitly, and possibly further components like additional notes or suspensions, which must be notated with every label explicitly. Each such component is identified by its *pitch class*. This in turn is identified (in most cases) by the smallest possible interval of one of its representatives above some representative of the root pitch class.

Basic design principle D-3, see section 1.3 above, says that all information should be decodable with as little context knowledge as sensible. Consequently, in *funSound* all interval sizes must be notated explicitly. This is in contrast to scale based systems, where a central major/minor decision rules the sizes of all possible chord components on all possible scale degrees.

```
euplus(euler(Q1, T1), euler(Q2, T2), euler(Q3, T3)) :–
    plus(Q1, Q2, Q3),
    plus(T1, T2, T3).

eusum_list( List , Sum) :–
    foldl (euplus, List , euler(0, 0), Sum).
```

Listing 2.12: Underlying Euler Arithmetics

```
funRootN(D,RootText,Euler) ⟶{fun_emotioFugax}, root_and_mode(RootText),!,
  {  normalize(D,RootText,Ram_norm),
     maplist(base_interval, Ram_norm, Terms),
     eusum_list(Terms, Euler)
  }.

funRootN(D,RootText,Euler) ⟶
    root_and_mode(RootText),!,
    { normalize(D,RootText,Ram_norm),
      last(Ram_norm, X),
      ( (X='↑'; X='↓') ->
        error(D, "Superfluous␣mode␣change,␣e.g.␣use␣[Tg],␣not␣[TG]",
             [RootText]); true ),
      maplist(base_interval, Ram_norm, Terms),
      eusum_list(Terms, Euler)
    }.

%!funSounds(−−Sound:list)
funSounds(D,[Sound | AndSounds]) ⟶
    funSound(D,Sound),
    andFunSounds(D,AndSounds).

%!funSounds(++LastSound:list −−Sound:list)
funSounds(D,[LastSound|AndLastSounds], [Sound | AndSounds]) ⟶
    funSound(D,LastSound,Sound),
    andFunSounds(D,AndLastSounds,AndSounds).

andFunSounds(D,[], [S | Z]) ⟶andFunSounds(D,[S | Z]), !.

andFunSounds(D,[S0|Z0], [S | Z]) ⟶[&], funSound(D,S0, S), !, andFunSounds(D,Z0, Z).
andFunSounds(_D,_,[]) ⟶[].

andFunSounds(D,[S | Z]) ⟶['&'], funSound(D,S), !, andFunSounds(D,Z).
andFunSounds(_D,[]) ⟶[].
```

Listing 2.13: Parsing of Harmonic Function Roots and Compound Sounds

```
% case: only ram−expr present, no predecessor
funSound(D,sound(Root,Pitches,Bass,Mel,Ram_source,I_source,Suppresses)) ⟶
     root_and_mode(Ram_source),!,
     { process_ram(D,Ram_source,Root,Mode,Is_dominant)} ,
     suppresses(Suppresses),
     intervals_opt (D,I_source),
     {   filter_no_inherit_interval   (D, I_source, I_clean ),
       process_intervals(D, I_clean ,  Pitches,  Bass, Mel, Mode, Is_dominant, Suppresses)
     }.

% error case: only  intervals  present,  but  no predecessor
funSound(D,sound(euler(0,0),[],undef,undef,['D','S','D','S'], [], 2)) ⟶
     intervals (D,_I_source),   !,
     { error(D, "attempt␣to␣inherit␣interval␣with␣no␣chord␣preceding", []) }.

% case: predecessor and ram−expr present
funSound(D,sound(_,_,_,_,Old_ram_source,Old_i_source,_),
            sound(Root,Pitches,Bass,Mel,Ram_source,Completed_i_source,Suppresses)) ⟶
     root_and_mode(Ram_source),!,
     { process_ram(D,Ram_source,Root,Mode,Is_dominant)} ,
     suppresses(Suppresses),
     intervals_opt (D,I_source),
     {  check_inherit_intervals (Old_i_source, Inherit_i_source ,  Old_ram_source, Ram_source),
        inherit_intervals  (D,I_source, Inherit_i_source ,Completed_i_source),
       process_intervals(D, Completed_i_source, Pitches, Bass, Mel, Mode, Is_dominant, Suppresses)
     }.

% special case: predecessor witho NO intervals, her only  one   '.'   like    "T&d .&s"
funSound(D,sound(Root,Pitches,Bass,Mel,Old_ram_source,[],Suppresses),
            sound(Root,Pitches,Bass,Mel,Old_ram_source,[],Suppresses)) ⟶
     intervals (D,[ inherit ]),   !.

% case: only intervals  present,  predecessor required  ( intervals  at  least  one  '.'   !):
funSound(D,sound(Root,_,_,_,Old_ram_source,Old_i_source,Suppresses),
            sound(Root,Pitches,Bass,Mel,Old_ram_source,Completed_i_source,Suppresses)) ⟶
     intervals (D,I_source),   !,
     {  inherit_intervals  (D,I_source,Old_i_source,Completed_i_source),
       extract_mdom(Old_ram_source,Mode,Is_dominant),
       process_intervals(D, Completed_i_source, Pitches, Bass, Mel, Mode, Is_dominant, Suppresses)
     }.

process_ram(D,Source,Root,Mode,Is_dominant) :−
     normalize(D,Source,Ram_norm),
     maplist(base_interval,  Ram_norm, Terms),
     eusum_list(Terms, Root),
     extract_mdom(Source,Mode,Is_dominant).
```

Listing 2.14: Parsing of and Inheriting From Harmonic Labels

```
% GLOBAL input: default_intervals, suppress_default_intervals
%! processIntervals(++I_source: list,--Pitches:list,--Bass:atom,--Mel:atom,++Mode:atom,
%!   ++Is_dominant:atom, ++Suppresses:int)
process_intervals(D, I_source,Pitches,Bass,Mel,Mode,Is_dominant,Suppresses) :-
        extract_base_and_mel(D, I_source, 1, undef, undef, BassIndex, MelIndex),
         default_intervals ( Default_intervals ),
        suppress_default_intervals (Suppress_default_intervals ),
        add_defaults(Suppresses, All_source,I_source, Default_intervals ,I_source,
                          Suppress_default_intervals ),
         interval_pitches (All_source,  Pitches,  Mode,Is_dominant),
         single_pitch (Mode,Is_dominant,I_source,BassIndex,Bass),
         single_pitch (Mode,Is_dominant,I_source,MelIndex,Mel).

single_pitch (_,_,_,undef,undef) :-!.
single_pitch (Mode,Is_dominant,I_source,Index,[Index,Euler]) :-
     nth1(Index,I_source,Source),
      interval_pitch (Source,Euler,Mode,Is_dominant).

% returns either a list (copy of the predecessors source text) or a non-list:
% ++Pitches0:parsed source text, --pitchesPre:list for inheritance operator, ++Old_ram_source, ++Ram_source
 check_inherit_intervals (Pitches0, PitchesPre, Old_ram_source, Ram_source) :-
     reverse(Old_ram_source, [OH|OT]),
     reverse(Ram_source, [H|T]),
     OT = T,
     to_upper2(OH,OHU), to_upper2(H,HU), OHU=HU, !,
     PitchesPre=Pitches0.
 check_inherit_intervals (_,PitchesPre,_,_)   :- PitchesPre=noIntervalInheritance.

to_upper2('↑','↑').
to_upper2('↓','↑').
to_upper2(A,B) :- upcase_atom(A,B).

%!  inherit_intervals (++NewIntSource:list, ++OldIntSource:list,  --Combined:list) is det
 inherit_intervals (D, [ inherit |R1], [ [I,Mod,B,Mel|_] |R2], [ [I,Mod,B,Mel,inherit] |R3]) :- !,
      inherit_intervals (D, R1,R2,R3).
 inherit_intervals (D, [ inherit |R1], [], R1) :- !,
     error(D, "attempt␣to␣inherit␣interval␣from␣undefined␣stack␣position",
           [[ inherit |R1 ]]).
 inherit_intervals (_D, [], _, []) :- !.    %% CUT kann evtl nach error aufraeumen wieder raus?
 inherit_intervals (D, [I1|R1], [_|R2], [I1|R3]) :- !,   inherit_intervals (D, R1,R2,R3).
 inherit_intervals (D, [I1|R1], noIntervalInheritance, [I1|R3]) :- !,
      inherit_intervals (D, R1,noIntervalInheritance,R3).
 inherit_intervals (D, [I1|R1], [], [I1|R3]) :- !,   inherit_intervals (D, R1,[],R3).
 inherit_intervals (D, [ inherit |R1],noIntervalInheritance,R1) :-
     error(D, "attempt␣to␣inherit␣interval␣from␣different␣root").

  filter_no_inherit_interval   (_D ,[],[])  :-!.
  filter_no_inherit_interval   (D,[ inherit |Ri],Ro) :-!,
     error(D, "attempt␣to␣inherit␣interval␣with␣no␣chord␣preceding", []),
        filter_no_inherit_interval   (D,Ri,Ro).
  filter_no_inherit_interval   (D,[I|Ri],[I|Ro]) :-  filter_no_inherit_interval   (D,Ri,Ro).
```

Listing 2.15: Inheritance of Interval Source Text

```
:- dynamic(root_and_mode/3).
:- dynamic(root_and_mode_2/3).

 select_locale_for_function (Loc) :-
     retractall (root_and_mode(_,_,_)),
    atom_concat('root_and_mode_',Loc,KA),
    Code =.. [KA,A,B,C],
    assertz( (root_and_mode(A,B,C) :- Code) ).

:-  select_locale_for_function ('DE').

root_and_mode_DE(['D'|Succ]) ⟶['D'], !, root_and_mode_2_DE(Succ).
root_and_mode_DE(['d'|Succ]) ⟶['d'], !, root_and_mode_2_DE(Succ).
root_and_mode_DE(['S'|Succ]) ⟶['S'], !, root_and_mode_2_DE(Succ).
root_and_mode_DE(['s'|Succ]) ⟶['s'], !, root_and_mode_2_DE(Succ).
root_and_mode_DE(['T'|Succ]) ⟶['T'], !, root_and_mode_2_DE(Succ).
root_and_mode_DE(['t'|Succ]) ⟶['t'], !, root_and_mode_2_DE(Succ).

root_and_mode_2_DE(['D'|Succ]) ⟶['D'], !, root_and_mode_2_DE(Succ).
root_and_mode_2_DE(['d'|Succ]) ⟶['d'], !, root_and_mode_2_DE(Succ).
root_and_mode_2_DE(['S'|Succ]) ⟶['S'], !, root_and_mode_2_DE(Succ).
root_and_mode_2_DE(['s'|Succ]) ⟶['s'], !, root_and_mode_2_DE(Succ).
root_and_mode_2_DE(['G'|Succ]) ⟶['G'], !, root_and_mode_2_DE(Succ).
root_and_mode_2_DE(['g'|Succ]) ⟶['g'], !, root_and_mode_2_DE(Succ).
root_and_mode_2_DE(['P'|Succ]) ⟶['P'], !, root_and_mode_2_DE(Succ).
root_and_mode_2_DE(['p'|Succ]) ⟶['p'], !, root_and_mode_2_DE(Succ).
root_and_mode_2_DE([]) ⟶ [].

suppresses(2) ⟶['/', '/'], !.
suppresses(1) ⟶['/'], !.
suppresses(0) ⟶ [].
```

Listing 2.16: Parsing of Function Roots

```
% (++Language:atom, ++InputText:String, ++NormalizedText:String
def_locale_for_function_head (Lang,Text,Norm) :−
    atom_concat("root_and_mode_",Lang,Nonterm),
    atom_concat("root_and_mode_2_",Lang,Nonterm2),
    def_locale_function (Nonterm, Nonterm2, Text, Norm).
 def_locale_for_function_tail  (Lang,Text,Norm) :−
    atom_concat("root_and_mode_2_",Lang,Nonterm2),
    def_locale_function (Nonterm2, Nonterm2, Text, Norm).
def_locale_for_function_both (Lang,Text,Norm) :−
    atom_concat("root_and_mode_",Lang,Nonterm),
    atom_concat("root_and_mode_2_",Lang,Nonterm2),
    def_locale_function (Nonterm, Nonterm2, Text, Norm),
    def_locale_function (Nonterm2, Nonterm2, Text, Norm).

% Back conversion (for pretty  printing  etc.)  STILL MISSING
% ++Nonterm:Atom (like "root_and_mode_2_NL") ++Input:String, ++ANorm:Atom(einstellig!)
% creates a rule  like ...
%   root_and_mode_FX([ANorm|A]) ⟶[ Input ], root_and_mode_2_FX(A).
def_locale_function (Nonterm, Nonterm2, Input, ANorm) :−
    Call  =..  [Nonterm2, Succ],
    string_chars (Input,  [ First | Tail ]),
    upcase_atom(First,FU),
    downcase_atom(First,FD),
    upcase_atom(ANorm,NU),
    downcase_atom(ANorm,ND),
    HeadU =.. [Nonterm, [NU|Succ]],
    HeadD =.. [Nonterm, [ND|Succ]],
    ParserU = (HeadU ⟶( [FU|Tail],  !,  Call)  ),
    ParserD = (HeadD ⟶( [FD|Tail],  !,  Call)  ),
    dcg_translate_rule (ParserU, ParserCodeU),
    dcg_translate_rule (ParserD, ParserCodeD),
    assertz(ParserCodeU),
    assertz(ParserCodeD).

% creates a rule  like ...
% root_and_mode_DE(X, [Input|A], B) :−
%   !,  root_and_mode_2_DE(X, [Expansion|A], B).
def_abbreviation_for_function (Lang,Input,Expansion) :−
    atom_concat("root_and_mode_",Lang,Nonterm),
    atom_concat("root_and_mode_2_",Lang,Nonterm2),
    string_chars (Input,CInput),
    string_chars (Expansion,CExpansion),
    append(CInput,A,CAInput),
    append(CExpansion,A,CAExpansion),
    Head =.. [Nonterm,X,CAInput,B],
    Head2 =.. [Nonterm2,X,CAInput,B],
    Call  =..  [Nonterm2,X,CAExpansion,B],
    assertz(Head:−(!,Call)),
    assertz(Head2:−(!,Call)).
```

Listing 2.17: Localisation of Function Roots

Thus all explicit intervals (in contrast to the default intervals, see below) are specified by appending a sequence of instances of the non-terminal *intervalDecorated*, see Table 2.1 above, to the root specification. Each consists of a number, given the traditional numeric name of the interval, plus possibly a sequence of modifiers. The syntax of the modifiers is pluggable, see the example definitions in Listing 2.21, and their semantics are defined by the rules interval_pitch /4, see the example definitions in Listing 2.23.

Every *intervalDecorated* is parsed into a list of four items: the basic name, encoded as a natural number, the sequence of modifiers (verbatim the source text), and whether the interval has been marked as bass or melody tone, which is decoded by the parser procedure base_mel/4 in line 687.

To this explicit input, *two levels of defaults* are applied:

Firstly **explicit inheritance of intervals** is possible under the following conditions: (a) An immediate predecessor chord exists, i.e. a sound/7 node at the same position in the containing sum/1 node as the current chord. (This is tested by the parser state switching from the procedure items/8 to sound_items/9).

(b) The current node has either no own root specification (i.e. exists as interval specifications only), or has verbatim the same text as that predecessor, not regarding the case of the very last character. (This is tested by an explicit call in line 443 to the procedure check_inherit_intervals /4 in line 487.)

Under these conditions the *inherited interval* operator "`.`" may appear at a particular position in the list of intervals. Then the (analysed) source text of the interval specification at this very position in the predecessor chord is copied. Therefore a further condition must hold, that (c) the list of explicit interval specifications of the predecessor is long enough. (This list may of course itself include inherited intervals). This is checked by the procedure inherit_intervals /4 in line 500.

The complicated condition (b) is necessary to support typical inheritance notations like

"**`D89 7.`**"

as well as

"**`TG56+ Tg..`**"

In many historical analytical texts, different interval numbers at the same position of the (vertically printed) interval lists of adjacent chord symbols are meant as *voice leading*. So the sequence "**`D488 3..  .79-`**" reads as: "The four is a suspension going (one tone down) to the three, and afterwards one octave goes down to the seventh while another voice goes from the octave up to the ninth." *These semantics are currently not supported by funSound.* Instead, the semantics of all chord symbols are merely *mathematical sets of pitch classes*, without any information about octave register and voice leading. But the weaker semantics of course hold as a generalisation, and assuming interval_defaults_conventional (see below) the sequence above reads as: "The pitch class of the fourth interval sounds with the first chord and vanishes with the second, when the third starts sounding. In the third chord seventh and ninth sound additionally (while the octave does in no place add a pitch class, since the root's pitch sounds anyhow.)"

A special case is when the preceding sound does not specify any interval and the current sound is written as a single dot "`.`". This means an inheritance of the whole sound (similar to the "idem" operator "`-`" one level above) and is realised in line 449. A typical example is the second system in Figure 2.5. This notation is a *canonical continuation*: If the chord has no own root specification but only intervals, it inherits root and mode from its immediate predecessor anyhow, see line 454. If the predecessor specifies more than zero intervals, than the same number of "`.`" characters is required to inherit them all. So one dot has the same effect for a predecessor with one or with zero explicit intervals.

Second transformation: When **default intervals** are defined, then these are added to the set of interval pitch classes. This works as follows: Before parsing any funCode label expression, the set of default intervals and suppressing rules must be defined, for instance by calling one procedure from Listing 2.24.

The code in line 787 steps through the default intervals, calling the code at line 808 which steps through the explicit interval input, calling the code at line 804 which steps through the suppressing rules. Whenever an explicit interval is found with the same numeric name as the tested default interval, then the latter is not included. Whenever a *suppressing rule* matches the currently tested pair of default and explicit interval, then the default interval is included neither. Only after all explicit intervals and suppressing rules have been tested without such a veto, the default interval is included in the set of calculated intervals, see line 798.

The examples in Listing 2.24 have been chosen for demonstration purposes: any explicit "**`2`**" interval with *any* modifier suppresses the default "**`1`**" interval; an explicit "**`4`**" with empty modifier suppresses the "**`3`**", but a "**`4+`**" would not. (On the right side of the suppressed interval an "any" operator is not supported because you always should know which intervals are included in the default set!-)

Before these intervals are evaluated, the top level suppress operators "`/`" and "`//`", which in the grammar directly follow *rootAndMode*, see Table 2.1, are parsed (lines 425, 441, and 546) and evaluated: The first is the classical "Verkürzung" (i.e. suppressing the sound of the root note, conventionally written like $\not{D}$) which is realised by inserting the sequence "**1/**" into the explicit intervals, which leads to a suppression of any default "**1**", see line 780.

"**//**" is a canonical continuation by funCode which suppresses *all* default intervals, realised in line 777, see Figure 2.5 for typical applications.

"**/**" appended to an interval number as a (predefined) modifier does not include any interval with this (numeric) name , but it suppresses all default intervals starting with this (numeric) name. (Parsing is in line 685 and evaluation in line 700.) So

"**T3/**"

describes a sound without third, and with interval_defaults_conventional it holds that

"**T/3/5/**" = "**T//**"

2.7.2 Interval Evaluation

The call to interval_pitches /4 evaluates all interval specifications (explicit, inherited or default) into a set of interval classes relative to the root.

The mapping from numeric interval names and modifiers is pluggable; Listing 2.23 shows example definitions. The evaluation of interval sizes can depend on the mode of the chord and on the fact whether it is a dominant. This allows shortcut definitions for "**3**" and "**7**" without the need of explicit modifiers, see line 725 and line 742.

A prominent example which requires this degree of flexibility is the definition of the dominant's minor seventh: It is highly controversial whether it shall be considered the root of the subdominant added to the dominant, or the minor third of the double dominant, or even a natural seventh, etc.[3]

In funCode this can be modelled very flexible. This sequence of definitions

```
modifier(['*']) --> ['*'], !.
interval_pitch([7,['-']|_],      euler(-2,0), _, _), !.
interval_pitch([7,['*']|_],      euler(2,-1), _, _), !.
```

supports two variants of sevenths and allows to select explicitly between them, by labelling e.g. "**D7-**" or "**D7***". Further adding

```
modifier(['v']) --> ['v'], !.
interval_pitch([7,['v']|_],      vogel(0,0,1), _, _), !.
```

would additionally allow a "natural seventh" according to Vogel (1975), as soon as the three dimensional vector space vogel/3 had been defined, together with an arithmetic integration of euler/2.

2.7.3 Two-Digits Interval Names

Figure 2.6 shows that the numeric interval name "**10**" is necessary even in rather conservative harmonics: As a suspension of the "**9-**", conflicting with the "normal" major "**3**" of the dominant.

Two further chord components can be defined by adding further thirds atop of the ninth, namely "**11**" and "**13**". Whether they are supported as first class chord components strongly depends on the underlying harmonic theory. Since they are identical modulo octave with "**4**" and "**6**", there are theories which only allow them as *suspensions*, i.e. as "nonchord tones" ("akkordfremd"). Others, like Jazz lead sheet notation, support them as chord components on their own, built by increasing the "tower of thirds".

Anyhow, the existence of "**15**" is supported by no single theory, because it collapses to a simple "**8**" and even "**1**" when talking of pitch *classes*.

On the other hand, *if* "**11**" and "**13**" are supported, then "**12**" and "**14**" (corresponding to "**5**" and "**7**") can also be allowed as their suspensions. The rules in Listing 2.19 enable four sensible combinations. (The semantics of all these intervals must be defined in a separate step anyhow, see Listing 2.23 below.)

Technically, the interval number "**10**" does not cause any ambiguity: There is a one-digit interval code "**1**" but no "**0**". This is different with the higher numbers. To resolve ambiguities, all parsers are *greedy* and the comma operator "**,**" can be inserted. There are not many situations where the interval number "**1**" is necessary, but imagine a suspension like "**T2-4**". The resolution of a double suspension can be written as "**T2-4 13**" if and only

---

[3] See among many others Hauptmann (1873, pg. 114), Riemann (1918, pg. 142), Imig (1970, pg. 86), Vogel (1975, pg. 92, referring to Leipniz and Euler), and Hewitt (2000, pg.112) for different standpoints.

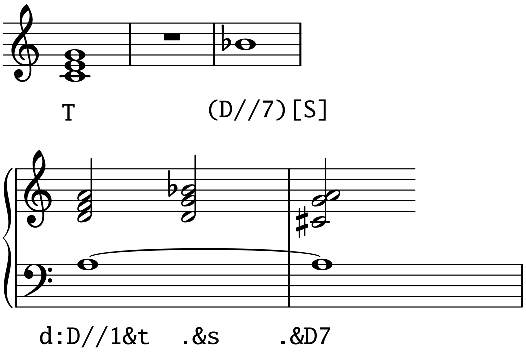


Figure 2.5: Labelling the concepts of “Klangvertretung” (“sound substitution”) and organ point

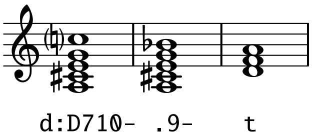


Figure 2.6: Interval Code “**`10`**” is necessary is even in rather conservative harmonics

if a thirteenth interval is syntactically not permitted. Otherwise one has to write “**`T2-4 1,3`**”. This is an example of the *semantic* layer determining the *syntactic*, a situation often found in DSLs.

2.7.4 Bass and Melody Specification

The parsing of the interval source text memorises in the third and fourth position of the generated data structure (list) whether the indicators for bass tone “_” and/or melody “^” are appended, see base_mel/4 in line 687. The one-based index and the Euler value are stored.

The procedure extract_base_and_mel, called from line 471 and defined in line 704, steps through all explicitly defined *and inherited* intervals, extracts the indices of the (at most one) base and melody indications, and stores them into (positions three and four of) the sound/7 data structure. It raises errors if more than one melody or bass is found.

It is crucial that for any inherited interval the complete source text is copied and translated anew. Therefore in the sequence “**`T5+_^7 t.`**” both bass and melody pitch class are the same for both chords, and “**`T5+_^7 t.3_`**” yields an error (“more than one bass tones”).

2.8 Retrieval of Information from the Constructed Data Base

Once a funCode source text has been parsed and translated into a data base, the “physical semantics” for every label can be retrieved. (Other ways of processing are described in sections 3.5 and 3.7.)

For this purpose, the *realising node number* must be known of the sum/1 node which holds the parsing result of the label. How to reach this is out of scope of this section. But when the document id D, the containing track (by the node number T of its realising node), and the score position S are given, then

funnode(D, N, T, S, sum(_))

delivers the realising node number N of the entry for this combination. There can maximally be one such entry.

The procedure all_tracks /2 in line 868 delivers all track nodes with their realising node number and their starting score position, which may help to locate the node for the interesting label.

```
:- dynamic(fun_extremaIdempotentes/0),
   dynamic(fun_initiaSyntonica/0), dynamic(fun_accidensRepetendum/0),
   dynamic(fun_emotioFugax/0),
   dynamic(fun_praesentatio_lineaVacuaUtSpatium/0),
   dynamic(fun_praesentatio_bassusUtSpatium/0).


set_style (D, false) :-  retractall (D) ,!.
set_style (D, true) :-  retractall (D), assertz(D).

get_style (D, S) :- D, !, S=true.
get_style (_D, S) :- S=false.

% Indicates whether bass and melody pitch indication may be repeated:
% fun_extremaIdempotentes :- false.

% Indicates whether initial key indications can specify syntonic commata:
% fun_initiaSyntonica :- false .

% Indicates whether multiple accidentals are allowed with track tonic centre:
% fun_accidensRepetendum :- false.

% Indicates whether modes may change explicitly with non-sounding functions:
% fun_emotioFugax :- false.

% When a track spans a whole staff and is non-empty before and after the
% limiting line breaks but completely empty between them, it is represented
% as a vertical gap. Otherwise the vertical space will be compressed.
% fun_praesentatio_lineaVacuaUtSpatium :- false.


% see also enabling of intervals > 10 by the ”allow_intervals_ ...” procedures.
```

Listing 2.18: General Style Parameters

```
%Supported combinations:
%11
%11 12
%11    13
%11 12 13 14

:- dynamic(interval_allowed/2).

interval_allowed (_,_) :- false.

no_intervals_larger_10 (D) :-  retractall ( interval_allowed (D,_)).

allow_interval_11 (D) :-
    no_intervals_larger_10 (D),
    asserta(interval_allowed(D,11)).

allow_interval_11_with_suspension(D) :-
    no_intervals_larger_10 (D),
    allow_interval_11 (D),
    asserta(interval_allowed(D,12)).

allow_intervals_11_13 (D) :-
    no_intervals_larger_10 (D),
    allow_interval_11 (D),
    asserta(interval_allowed(D,13)).

allow_intervals_11_13_with_suspension(D) :-
    no_intervals_larger_10 (D),
    allow_interval_11_with_suspension(D),
    asserta(interval_allowed(D,13)),
    asserta(interval_allowed(D,14)).
```

Listing 2.19: Enabling Interval Codes Larger Than 10

```
 intervals (D,[I|T])  ⟶ interval (D,I),  !,  intervals_comma(D,T).
intervals_comma(D,L) ⟶[',' ], !,  intervals (D,L).
intervals_comma(D,L) ⟶intervals_opt(D,L).
 intervals_opt (D,L)  ⟶ intervals (D,L),  !.
 intervals_opt (_D ,[])  ⟶ [].

 interval (_D, inherit )  ⟶['.'],  !.

 interval (_D,[10,Modif,B,M]) ⟶['1','0'], !,  modifier(Modif),  base_mel(B,M).
 interval (D,[11,Modif,B,M]) ⟶['1','1'], {interval_allowed(D,11)}, !,  modifier(Modif),  base_mel(B,M).
 interval (D,[12,Modif,B,M]) ⟶['1','2'], {interval_allowed(D,12)}, !,  modifier(Modif),  base_mel(B,M).
 interval (D,[13,Modif,B,M]) ⟶['1','3'], {interval_allowed(D,13)}, !,  modifier(Modif),  base_mel(B,M).
 interval (D,[14,Modif,B,M]) ⟶['1','4'], {interval_allowed(D,14)}, !,  modifier(Modif),  base_mel(B,M).
 interval (_D,[1,Modif,B,M]) ⟶['1'], !,  modifier(Modif),  base_mel(B,M).
 interval (_D,[2,Modif,B,M]) ⟶['2'], !,  modifier(Modif),  base_mel(B,M).
 interval (_D,[3,Modif,B,M]) ⟶['3'], !,  modifier(Modif),  base_mel(B,M).
 interval (_D,[4,Modif,B,M]) ⟶['4'], !,  modifier(Modif),  base_mel(B,M).
 interval (_D,[5,Modif,B,M]) ⟶['5'], !,  modifier(Modif),  base_mel(B,M).
 interval (_D,[6,Modif,B,M]) ⟶['6'], !,  modifier(Modif),  base_mel(B,M).
 interval (_D,[7,Modif,B,M]) ⟶['7'], !,  modifier(Modif),  base_mel(B,M).
 interval (_D,[8,Modif,B,M]) ⟶['8'], !,  modifier(Modif),  base_mel(B,M).
 interval (_D,[9,Modif,B,M]) ⟶['9'], !,  modifier(Modif),  base_mel(B,M).

:− discontiguous(modifier/3).
modifier(suppress) ⟶['/'], !.

base_mel(true,true) ⟶['_', '^'],  !.
base_mel(true,true) ⟶['^',  '_'], !.
base_mel(true,false) ⟶['_'], !.
base_mel(false,true) ⟶['^'],  !.
base_mel(false,false) ⟶ [],  !.
```

Listing 2.20: Syntax of Intervals and Generic Modifiers

```
modifier (['+'|T]) ⟶['+'], !,  modifier_plus(T).
modifier (['-'|T]) ⟶['-'], !,  modifier_minus(T).
modifier ([])  ⟶ [].
modifier_plus (['+'|T]) ⟶['+'], !,  modifier_plus(T).
modifier_plus ([])  ⟶ [].
modifier_minus(['-'|T]) ⟶['-'], !,  modifier_minus(T).
modifier_minus ([])  ⟶ [].
```

Listing 2.21: Syntax of Specific Interval Modifiers, Pluggable

```
interval_pitches  ([],[], _,_).
interval_pitches ([ [_,suppress, _, _] |B], D, Mode,IsD) :- !, interval_pitches (B, D, Mode,IsD).
interval_pitches ([A|B], [C|D], Mode, IsD) :-
    interval_pitch (A,C,Mode,IsD), interval_pitches(B, D, Mode, IsD).

extract_base_and_mel(_D, [], _, B, M, B, M) :- !.
extract_base_and_mel(D, [ [_,_,false,false|_] | R], Index, B, M, Bout, Mout) :- !,
    succ(Index,J), extract_base_and_mel(D, R, J, B, M, Bout, Mout).

extract_base_and_mel(D, [ [_,_,true,Mval|_] | R], Index, undef, M, Bout,Mout) :- !,
    extract_base_and_mel(D, [ [_,_,false,Mval] |R], Index, Index, M, Bout, Mout).
extract_base_and_mel(D, [ [_,_,true,Mval|_] | R], Index, Bset, M, Bout,Mout) :- !,
    error(D, "double␣bass␣pitch␣selection", [Bset, Index]),
    extract_base_and_mel(D, [ [_,_,false,Mval] |R], Index, Bset, M, Bout, Mout).

extract_base_and_mel(D, [ [_,_,false,true|_] | R], Index, B, undef, Bout,Mout) :- !,
    succ(Index,J), extract_base_and_mel(D, R, J, B, Index, Bout, Mout).
extract_base_and_mel(D, [ [_,_,false,true|_] | R], Index, B, Mset, Bout, Mout) :-
    error(D, "double␣melody␣pitch␣selection", [Mset, Index]),
    extract_base_and_mel(D, [ [_,_,false,false] |R], Index, B, Mset, Bout, Mout).
```

Listing 2.22: Interval Semantics and Base and Melody indication

Once the realising node number N for a sum/1 node is known, the function call

`all_results (D,N, R,P,B,M)`

(see line 892 in Listing 2.28) delivers

- *R* = the root pitch class of the only functional label, or that of the first one if there is a "`..&..&..`" construct.
- *P* = the set of all pitch classes, unified over the sum expression "`..&..&..`".
- *B* = the explicitly set pitch class of the bass, or undef if none has been specified in the source text.
- *M* = the explicitly set pitch class of the melody note, or undef if none has been specified in the source text.

For the bass note's pitch class there are three cases:

- It may be given explicitly, so B\= undef.
- Otherwise the "first root" R (which is always a defined Euler value) may be taken as such, but only if it is a member in P!
- Otherwise there is no information about the bass pitch class.

The implementation has to determine the reference pitch, relative to which all intervals must be resolved. For this it first finds the reference node as stored in relative_root /3, which is either a track/3 or another sum/1 node. To this node find_root /3 is applied. This procedure (see line 925) analyses the node object: A track node has either its own tonic centre, or inherits it from its parent track. For a sound or virtual node, its root is added to its respective reference point; the latter is calculated by recursive application of find_root /3.

Technically, some error conditions are not discovered before the corresponding retrieval is performed:

- More than one function labels in a sum expression specify a bass pitch or a melody pitch. (The style parameter fun_extremaldempotentes/1 allows to specify the same pitch class more than once, as in "**`D1_&T5_`**", see line 915.)
- The reference point (whether sounding or virtual) of a relative region is part of a sum of more than one label. (This is currently not supported, but see section 3.3.)

Currently these error messages are added to the Prolog data base and the inquiry fails.

For practical application, the code should further be enhanced e.g. by automatic navigation when retrieving an idem label, i.e. delivering the values of its immediate predecessor. The current implementation simply fails for the still unimplemented cases.

```
%! interval_pitch (++IntervalSize : int , ++Modifier: list , --Result:euler, ++majorMinor,++isDom).
interval_pitch ([1,  []| _],            euler (0,0),  _,  _).

interval_pitch  ([2,[ '+']|_],          euler (2,0),  _,  _) :- !.
interval_pitch  ([2,[ '-']|_],          euler(-1,-1), _,  _).

interval_pitch  ([3,[]| _],             euler (0,1),  major, _) :- !.
interval_pitch  ([3,[]| _],             euler(1,-1),  minor, _) :- !.
interval_pitch  ([3,[ '+']|_],          euler (0,1),  _,  _) :- !.
interval_pitch  ([3,[ '-']|_],          euler(1,-1),  _,  _).

interval_pitch  ([4,[]| _],             euler(-1,0),  _,  _) :- !.
interval_pitch  ([4,[ '+']|_],          euler (2,1),  _,  _) :- !.
interval_pitch  ([4,[ '-']|_],          euler(0,-2),  _,  _).

interval_pitch  ([5,[]| _],             euler (1,0),  _,  _) :- !.
interval_pitch  ([5,[ '+']|_],          euler (0,2),  _,  _) :- !.
interval_pitch  ([5,[ '-']|_],          euler(-2,-1), _,  _).

interval_pitch  ([6,[ '+']|_],          euler(-1,1),  _,  _) :- !.
interval_pitch  ([6,[ '+', '+']|_],     euler (2,2),  _,  _) :- !.
interval_pitch  ([6,[ '-']|_],          euler(0,-1),  _,  _).

interval_pitch  ([7,[]| _],             euler(-2,0),  major, is_dominant) :- !.
interval_pitch  ([7,[ '+']|_],          euler (1,1),  _,  _) :- !.
interval_pitch  ([7,[ '-']|_],          euler(-2,0),  _,  _).

interval_pitch  ([8, M|_],   S,  _,  _)   :-  interval_pitch (  [1,M|_],  S,  _,  _).
interval_pitch  ([9, M|_],   S,  _,  _)   :-  interval_pitch (  [2,M|_],  S,  _,  _).
interval_pitch  ([10,M|_],  S,  Mode, _)  :-  interval_pitch (  [3,M|_],  S,  Mode, _).
interval_pitch  ([11,M|_],  S,  _,  _)    :-  interval_pitch (  [4,M|_],  S,  _,  _).
interval_pitch  ([12,M|_],  S,  _,  _)    :-  interval_pitch (  [5,M|_],  S,  _,  _).
interval_pitch  ([13,M|_],  S,  _,  _)    :-  interval_pitch (  [6,M|_],  S,  _,  _).
interval_pitch  ([14,M|_],  S,  Mode,Is_Dominant) :-  interval_pitch (  [7,M|_],  S,  Mode, Is_Dominant).
```

Listing 2.23: Interval Semantics (Pluggable)

```
:- dynamic(default_intervals/1), dynamic(suppress_default_intervals/1).

set_interval_defaults_none :-
     retractall ( default_intervals (_)),  retractall (suppress_default_intervals(_)),
    asserta(default_intervals ([])),  asserta(suppress_default_intervals ([])).

% every 2 suppresses the default 1, even 2++;
% 4 suppresses 3, but 4+ doesn't;
% both 6- and 6+ suppress default 5, but not augmented 6++:
set_interval_defaults_conventional :-
     retractall ( default_intervals (_)),  retractall (suppress_default_intervals(_)),
    asserta(default_intervals ([  [1,[]],   [3,[]],   [5,[]]  ])),
    asserta(suppress_default_intervals([  [2,any,  1,[]],  [4,  [],  3,[]],  [6,['-'],5,[]],
                                          [6,['+'],5,[]]  ])).

% as above; additionally  6+ suppresses default 7- but not 7+ (if it were a default):
set_interval_defaults_hypothetical :-
     retractall ( default_intervals (_)),  retractall (suppress_default_intervals(_)),
    asserta(default_intervals ([  [1,[]],   [3,[]],   [5,[]],   [7,['-']] ])),
    asserta(suppress_default_intervals([  [2,any,  1,[]],  [4,  [],  3,[]],  [6,['-'],5,[]],
                                          [6,['+'],5,[]],  [6,['+'],7,['-']] ])).

:- set_interval_defaults_conventional .
```

Listing 2.24: Default Chord Components (Pluggable)

2.9 Summary of funCode Canonical Continuations

funCode brings some light extensions beyond GM-style notation, all of them canonical continuations:

- Writing “`D3/`” to suppress the default of a particular interval.
- Writing “`D//`” to suppress the default of all intervals.
- Writing “`TpD`” instead of “`(D)[Tp]`”.
- Writing “`Tp(:D)`” analogous to “`(D)Tp`”.
- Finding the reference point of a relative section independently from its parentheses, as in “`(D)(D)(D)t`”.

```
%! add_defaults(++Suppresses:int, ..
add_defaults(2, I_source,I_source,_,_,_) :- !.
add_defaults(1, All_source,I_source, Default_intervals ,I_source,Suppress_default_intervals) :- !,
    add_defaults(All_source, I_source,
                 Default_intervals , [[1, suppress,foo,foo]|I_source], Suppress_default_intervals).
add_defaults(0, All_source,I_source, Default_intervals ,I_source,Suppress_default_intervals) :-
    add_defaults(All_source,I_source, Default_intervals ,I_source,Suppress_default_intervals).

% add_defaults(--Result:list,?Akku:list,++Defaults: list ,++ Explicit : list ,++Suppress_rules:list)
add_defaults(Akku, Akku, [], _Explicit , _Suppress_rules) :- !.  %% ATTENTION cut should be redundant??

add_defaults(Result, Akku, [I|Rest], Explicit , Suppress_rules) :-
    add_one_default(Res, Akku, I, Explicit ,Suppress_rules,Suppress_rules),
    add_defaults(Result, Res, Rest,Explicit ,Suppress_rules).

% result, akku, ondeDefault, explicit , suppress_rules, suppress_rules-backup
% ( explicit = number, modifiers, isBase, isMel):
% steps for ONE default interval through all explicit intervals and all rules:
% discard default due to explicit interval with the same number:
%   ("..| _]" stands for the optional ", inherit ]")
add_one_default(Akku, Akku, [I,_], [ [I,_,_,_|_]| _], _Suppress_rules,_) :-!.
% accept default since explicits all tested:
add_one_default([ [I,M,notBase,notMel]|Akku], Akku, [I,M], [], _Suppress_rules,_) :-!.

% discard default due to suppression rule:
add_one_default(Akku, Akku, [I,M], [ [J,N,_,_|_]| _], [[J, N, I, M]|_ ], _Suppress_rules) :-!.
add_one_default(Akku, Akku, [I,M], [ [J,_,_,_|_]| _], [[J, any, I, M]|_ ], _Suppress_rules) :-!.
% check next suppress rule (with same explicit interval ):
add_one_default(Result, Akku, Int, Explicit , [_|Rest], Suppress_rules) :- !,
  add_one_default(Result, Akku, Int, Explicit , Rest, Suppress_rules).

% check against next explicit , with all suppress rules restored:
add_one_default(Result, Akku, Int, [ _Explicit |Rest], [] ,Suppress_rules) :-
  add_one_default(Result, Akku, Int, Rest, Suppress_rules, Suppress_rules).
```

Listing 2.25: Evaluate Default Chord Components

```
%! normalize(++Input:list,--Output:list)
normalize(_D,[X],  Y) :- !,  Y=[X],
     is_valid_function (X).

normalize(X,[A, B | R],  [A, C | S]) :-
    normalize(A, B, C, D),  !,
    normalize(X,[D | R],  S).

normalize(D,[A, B | R],  [A | S]) :-
    \+ fun_emotioFugax,
    member(A, ['↑', '↓']),
    member(B, ['S', 's', 'D', 'd']),
    !,
    error(D, "Superfluous␣mode␣change,␣e.g.␣use␣TgD,␣not␣TGD", [ [A,B|R] ]),
    normalize(D,[B | R],  S).

normalize(D,[A | R],  [A | S]) :-
     is_valid_function (A),
    normalize(D,R, S).

 is_valid_function (X) :- fun_upper(X),  !.
 is_valid_function (X) :- fun_lower(X).

normalize(X, 'G', 'g', '↑') :- fun_upper(X).
normalize(X, 'P', 'p', '↑') :- fun_upper(X).
normalize(X, 'g', 'G', '↓') :- fun_lower(X).
normalize(X, 'p', 'P', '↓') :- fun_lower(X).

%! fun_upper(++X:atom)
fun_upper(X) :- member(X, ['S', 'T', 'D', 'P', 'G', '↑']), !.
fun_lower(X) :- member(X, ['s', 't', 'd', 'p', 'g', '↓']), !.

% calculation of the root of the chords:
base_interval('T', euler( 0,   0)).
base_interval('t', euler( 0,   0)).
base_interval('D', euler(+1,   0)).
base_interval('d', euler(+1,   0)).
base_interval('S', euler(-1,   0)).
base_interval('s', euler(-1,   0)).
base_interval('↑', euler( 0,   0)).
base_interval('↓', euler( 0,   0)).
base_interval('P', euler(+1,  -1)).
base_interval('p', euler(-1,  +1)).
base_interval('G', euler( 0,  -1)).
base_interval('g', euler( 0,  +1)).
```

Listing 2.26: Aux Routines for Roots

```
%mode(Symbols, Mode) :-
%     append(_,[Last],Symbols),
%     (fun_upper(Last),  !,  Mode = major ; fun_lower(Last),  !,  Mode = minor).

extract_mdom(Symbols, Mode, Is_dominant) :-
     append(_,[Last],Symbols), !,
     extract_mdom2(Last,Mode,Is_dominant).

extract_mdom2('D', major, is_dominant) :- !.

extract_mdom2(Root, major, []) :- fun_upper(Root),  !.

extract_mdom2(_Root, minor, []).
```

Listing 2.27: Analysing the Last Character of the Root Specification

```
 all_tracks (D,Tracks) :-
     findall ([Node,Scorepos,T],
              (fun_node(D,Node,_,Scorepos,T), T=track(_,_,_)), Tracks).

all_messages(D, Messages) :-
     findall ([Node,Track,Scorepos,Msg,Args],
              ( fun_error (D,Node,Track,Scorepos,Msg,Args) ;
                fun_warning(D,Node,Track,Scorepos,Msg,Args) ),
              Messages).

% track_end is unclusive, and so is the result
last_score_pos(D, Last) :-
     findall (P, track_end(D,_,_,P), ScorePoss),
     max_list (ScorePoss,Last).


% allResults(DocId++,SumNodeId++,Root--,Pitches--,Bass--,Melody--)
% Deliver the most important semantic data encoded with the label
% identifed by the given realizing node number.
% Bass and Melody are of type euler, if specified exactly once.
% If not specified then undef. If more than once then "error ..."
% Root is that of the first functional symbol in "A&B&.."
% Possibly it is NOT member in pitches!

 all_results (D,Index,Root,Pitches,Bass,Melody) :-
    fun_node(D,Index,Track,Score,sum(Sounds)),
    set_error_pos (D,Index,Track,Score),
     relative_root (D,Index,Sup),
    find_root (D,Sup,RelBase),
    Sounds=[sound(FirstRoot,_,_,_,_,_,_)|_],
    euplus(RelBase,FirstRoot,Root),
     extract_results (D,Sounds,RelBase,[],undef,undef,PitchesBag,Bass,Melody),
    \+ fun_error(D,_,_,_,_,_),
    sort (PitchesBag,Pitches).

 extract_results (D,[sound(MyRoot,MyPitches,MyBass,MyMelody,_,_,_)|Rest],RelBase,
                  PrePitches,PreBass,PreMelody,Pitches,Bass,Melody) :-
    !, euplus(MyRoot,RelBase,Root),
    combine(D,"bass",Root,MyBass,PreBass,NewBass),
    combine(D,"melody",Root,MyMelody,PreMelody,NewMelody),
    combine_set(Root,MyPitches,PrePitches,NewPitches),
     extract_results (D,Rest,RelBase,NewPitches,NewBass,NewMelody,Pitches,Bass,Melody).
 extract_results (_D ,[], _,Pitches,Bass,Melody,Pitches,Bass,Melody).

combine(_D,_Text,_RelBase,undef,Pre,Pre) :- !.
combine(_D,_Text,RelBase,[_, Euler],undef,New) :- !, euplus(RelBase, Euler,New).
combine(_D,_Text,RelBase,[_,MyEuler],Euler,Euler) :-
    fun_extremaIdempotentes, euplus(RelBase, MyEuler,Euler), !.
combine(D,Text,_RelBase,[_,MyEuler],Euler,Euler) :-
    format(string (Msg), "more␣than␣one␣w␣indication␣in␣sum␣expression", [Text]),
    error (D,Msg,[MyEuler, Euler]).

combine_set(RelBase,[My|Rest],Pre,Result) :- !,
    euplus(RelBase,My,NewPitch), combine_set(RelBase, Rest,[NewPitch|Pre],Result).
combine_set(_RelBase,[],Pre,Pre).
```

Listing 2.28: Retrieving the Results

```
%! find_root (D++,Index++,Euler−) is det
%@param Index: the node which represents the context/tonic reference point
find_root (D,Index, Euler) :−
    fun_node(D,Index,_,_,Node), find_root (D,Index,Node, Euler).

%! find_root (D++,Index++,Node++,Euler−) is det
% assume 1 is the topmost track
find_root (_,_,track(_,_,euler(Q,T)),euler(Q,T)) :− !.
find_root (D,1,track(_,_,_), euler(99,99)) :− !,
    error(D, "top␣track␣tonic␣centre␣is␣undefined", []).

find_root (D,Index,track(_,_ ,[]), Euler) :− !,
     relative_root (D, Index,Sup), find_root (D,Sup,Euler).
find_root (D,Index,sum([sound(Root,_,_,_,_,_)]), Euler) :−
    !, relative_root (D, Index,Sup), find_root (D,Sup,SupRoot),
    euplus(Root,SupRoot,Euler).
find_root (D,Index,sum([_,_|_R]), euler(99,0)) :−
    !, error(D,"cannot␣refer␣to␣a␣sum␣of␣more␣than␣one␣functions",[Index]).
find_root (D,Index, virtual (_Source,Root),Euler) :−
    !, relative_root (D, Index,Sup), find_root (D,Sup,SupRoot),
    euplus(Root,SupRoot,Euler).
find_root (D,Index,_, euler(99,0)) :−
    error(D,"cannot␣resolve␣this␣node␣as␣a␣reference␣point",[Index]).
```

Listing 2.29: Retrieving the Results – Continued

Chapter 3

Extensions

3.1 Errors and Warnings

As mentioned above, the guiding principles for writing code for practical application vs. readable specification may conflict: For the first purpose, (a) comprehensive error diagnosis is sensible, and (b) in one single parser application as many input errors as possible should be detected. This contradicts the coding principles for specification, which aim at lean and well readable code describing the allowed inputs.

The architecture chosen here is to generate an error message whenever the parsing or evaluation process cannot be continued. This message is stored as a fact with the document id, see line 42 and line 44. Then some dummy data is substituted to make possible some continuation of the processing, to detect further input errors for the user's convenience. Therefore any result which includes only a single error message must be completely discarded. This is reflected by line 8.

Warnings are generated when a result is allowed under the specification, but there is some reasonable doubt that the user really intended it, see for instance line 125.

To shorten the list of parameters passed down to the elementary auxiliary procedures, the current coordinates to be included in error messages are stored as the global fact fun_error_pos/4, set in line 47 and used in the shortened message generating routines in line 49 and line 51.

all_messages/1 defined in line 872 retrieves a list of all errors and warnings.

3.2 Localisation and Configuration

funCode comes with five areas for customisation:

- Global style parameters, affecting parsing or processing behaviour. These are summarised in Listing 2.18 on page 34 and explained in this report in their evaluation contexts.
- Syntax of interval modifiers and semantics of interval numeric names and modifiers. These must be plugged in anyhow, see Listing 2.21 on page 36 and Listing 2.23 on page 38.
- Abbreviations for often used sequences of characters and digits.
- Localisation of names and abbreviations, namely pitch class names for a track's tonic centre, and
- letter codes for functions.

In this report, all listings dealing with customisation and configuration are marked with a

thick green frame.

Pitch class names for tonic centres come in two flavours: names of "white keys", which can be followed by "`b`" or "`#`", and names of "full keys" which can only be followed by commata `,` and `'` like **`As`** or **`Eses`**.[1] Both categories *must* be followed by a colon "`:`" in the grammar of Table 2.1 and are thus unambiguously recognisable.

Listing 2.7 on page 21 shows some methods which define new parsers for a localised grammar: They allow to parse a lower case, an upper case and a capitalised version of the given text, yielding identical semantic values. Other kinds of definition macros are of course thinkable. Listing 2.8 on page 21 shows an example application. After loading the file, execute "**`listing(whiteKey_DE)`**" and "**`listing(fullKey_DE)`**" to see the concrete results.

---

[1] That this may be necessary has been mentioned above, see Lewin (2006, pg. 193).

```
:- dynamic(root_and_mode_FX/3).
:- dynamic(root_and_mode_2_FX/3).
:-  def_locale_for_function_both ('FX',"Od", 'D').
:-  def_locale_for_function_both ('FX',"Ud", 'S').
:-  def_locale_for_function_head ('FX',"T", 'T').
:-   def_locale_for_function_tail  ('FX',"L", 'G').
:-   def_locale_for_function_tail  ('FX',"R", 'P').
root_and_mode_2_FX([]) ⟶ [].
```

Listing 3.1: Fancy Names for Function Roots

“`Fr`”	↦	“`D5-_7`”	*// French Sixth*
“`It`”	↦	“`D/5-_7`”	*// Italian Sixth*
“`Gr`”	↦	“`D/5-_79-`”	*// German Sixth*

Table 3.1: English Names for Chords of the Augmented Sixth

It is important that the sequential order of the definition calls is significant: each definition of a parser contains a “cut” and is added by **assertz**/1, so when one possible input is a prefix of another, this longer one must precede.

The selection of the input language for tonic centres goes by calling select_locale_for_key /1 as defined in line 164 and called in line 174. It works by re-routing every call of whiteKey/4 to (e.g.) whiteKey_EN/4 and every call to fullKey /5 to fullKey_EN/5, by overwriting the rules contained in the Prolog data base.

The same technique is applied by select_locale_for_function /1 as defined in line 521, which re-routes root_and_mode/3 to any root_and_mode_XY/3. The hypothetical fancy format in Listing 3.1 combines the wording of the basic functions by Marschner (Imig, 1970, pg. 109) with the “triad transformations” by Hyer (1989, pg. 162pp) for the mediantic derivations. So `udl` is `sg` and `OdR` is `DP`.

The same caveats as with tonic centres apply also here, and there are two further:

- The implementation relies on the fact that the standard names “`T`”, “`t`”, etc., have only one single character.
- Only the *normalised* name is stored in each sound node. The reverse translation is not yet supported: The mapping from standard names to locale names must be stored and be applied in all output operations like the LaTeX back-end.

Abbreviations for often used and characteristic sequences of function code letters, possibly followed by interval numbers, can be defined using procedure def_abbreviation_for_function /3, as defined in line 586. (Its implementation relies on the way the DCG rules are expanded and thus relies on reverse engineering.)

Listing 3.2 shows an example. It shows that the names of the functions and the abbreviations must be ordered for the parser in the same way as described above for the tonic centres. Line 2093 on page 73 shows possible applications.

Table 3.1 shows the definitions for the English names of the chords of the augmented sixth.

It is crucial that the replacement text is inserted in the parsing process completely mechanically. So the user must be aware whether numbers are inserted. The sequence

“`c:sND`”

covers *two* score positions, because its expansion is

“`c:sG3_D`”

Similar the suspension

“`c:sN2- 1`”

will not work as expected, because the information “`3_`” will not be inherited to the second position. What is meant is probably

“`c:sN2- .1`”

which is expanded to “`c:sG3_2- 3_1`”.

```
:– dynamic(root_and_mode_DE2/3).
:– dynamic(root_and_mode_2_DE2/3).
:–  def_abbreviation_for_function (’DE2’, "DV", "D/79-").
:–  def_abbreviation_for_function (’DE2’, "Dhv", "D5-7").
:–  def_locale_for_function_both (’DE2’,"D", ’D’).
:–  def_abbreviation_for_function (’DE2’, "sN", "sG3_").
:–  def_locale_for_function_both (’DE2’,"S", ’S’).
:– def_locale_for_function_head (’DE2’,"T", ’T’).
:–   def_locale_for_function_tail  (’DE2’,"G", ’G’).
:–   def_locale_for_function_tail  (’DE2’,"P", ’P’).
root_and_mode_2_DE2([]) ⟶[].
```

Listing 3.2: Additional Abbreviations with German Function Names

3.3 Possible Future Features

- Riemann’s short notation for modulations:

```
c: T tP=D s
```

for

```
c: T >tP <as:D s
```

meaning

```
c: T tP
     as:D s
```

- Wildcard notation like

```
 D?_
```

to indicate all possible chord inversions (=any pitch may be bass).

- Indication of enharmonic notation like

```
 f:T      DD/79-
      f#: D/3˜5˜79-
```

to indicate by the decoration “˜” that the chord engraved as “b+d+f+a♭” must be read as “b+d+e♯+g♯” for the second interpretation.

- The possible combinations of sub-tracks by “`{..}`” and relative regions by “`(..)`” are not yet completely analysed.
- Perhaps relative regions and the sum construct could swap their hierarchical position, so that in a sequence of sums, different reference points can be active. The source could look like

“ `T&Tp (D&D DD&s)` ”

… or …

“ `D & (d)[D]` ”

… or even …

“ `[Tp] (:D&D DD)&(s:)  &[Dp]` ”

but it is still totally unclear whether there are practical applications and how complicated the implementation.

- Directed by a style parameter, adjacent inverses like “`..DS..`” or “`..Pp..`” could raise a warning or even an error.

3.4 Encoding of Score Positions

As mentioned above, the core specification in the preceding chapter takes the *score positions* as given and assigns to them all recognised labels consecutively.

A very primitive encoding of the score positions, which may be sufficient for many purposes, can easily added to the funCode syntax definitions:

A text like “`step=1/8`”, prepended to a funCode specification, can indicate the temporal distance of equidistant score positions. The following label sequence can skip some of them, whenever necessary, using the labels “˜” and “-”.

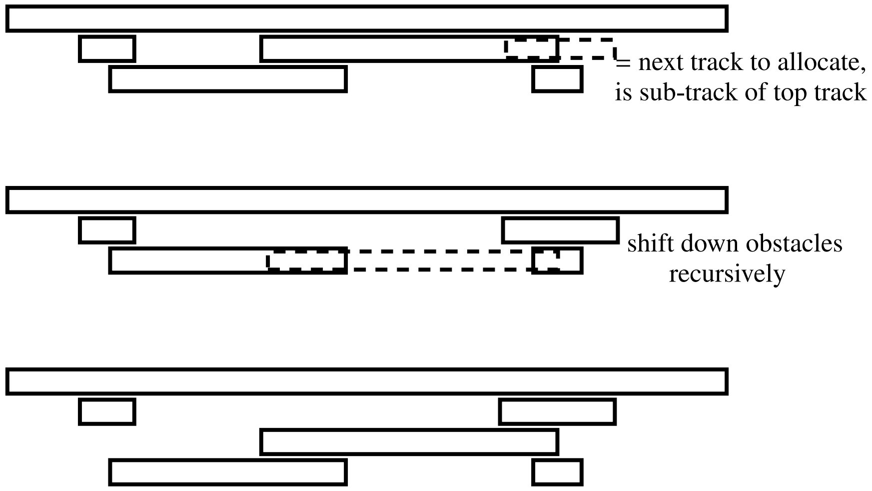


Figure 3.1: Example Application of the Layout Algorithm

Alternatively or additionally, the character "**|**", not yet used in funCode's syntax, can indicate bar lines, i.e. end of measures of the score under labelling. Applying both would look like
`step=1/4:  T ~ D - | s D T -`

3.5 Track Layout Algorithm

In nearly all historical sources, the *conventional two-dimensional arrangement (C2DA) sub-tracking* works by horizontal juxtaposition: A sub-track not carrying a tonic centre specification inherits such from its parent track. This is found by searching vertically upwards for the first track which overlaps the start position of this sub-track.

Therefore the rendering of the second example in Figure 2.3 on page 18 is *not* a correct C2DA: The track starting with "**K**" seems to inherit from the track starting with "**h:**", but the funCode source means it to inherit from "**a:**".

A general rule is, that no sub-track may be separated from its parent track by a sibling which overlaps its starting point.

This is achieved by the layout algorithm in Listing 3.3: Allocation means to assign a *row number* to each track, counting top down. Whenever a track has been successfully assigned, its sub-tracks are processed with increasing starting position (from left to right). Whenever the horizontal starting position of the first such sub-track is occupied (which includes the violation of a minimal distance gap, the width of which is given by the parameter Gap), then this obstacle is moved down one line by re-allocation, which starts this process recursively. See Figure 3.1 for an example. It is easily seen that this algorithm is correct: Every previously allocated sub-track is correctly connected to its parent track; the newly entered sub-track may shift these down, but never starts earlier than its already allocated siblings, so it cannot damage the connections to their respective parents.

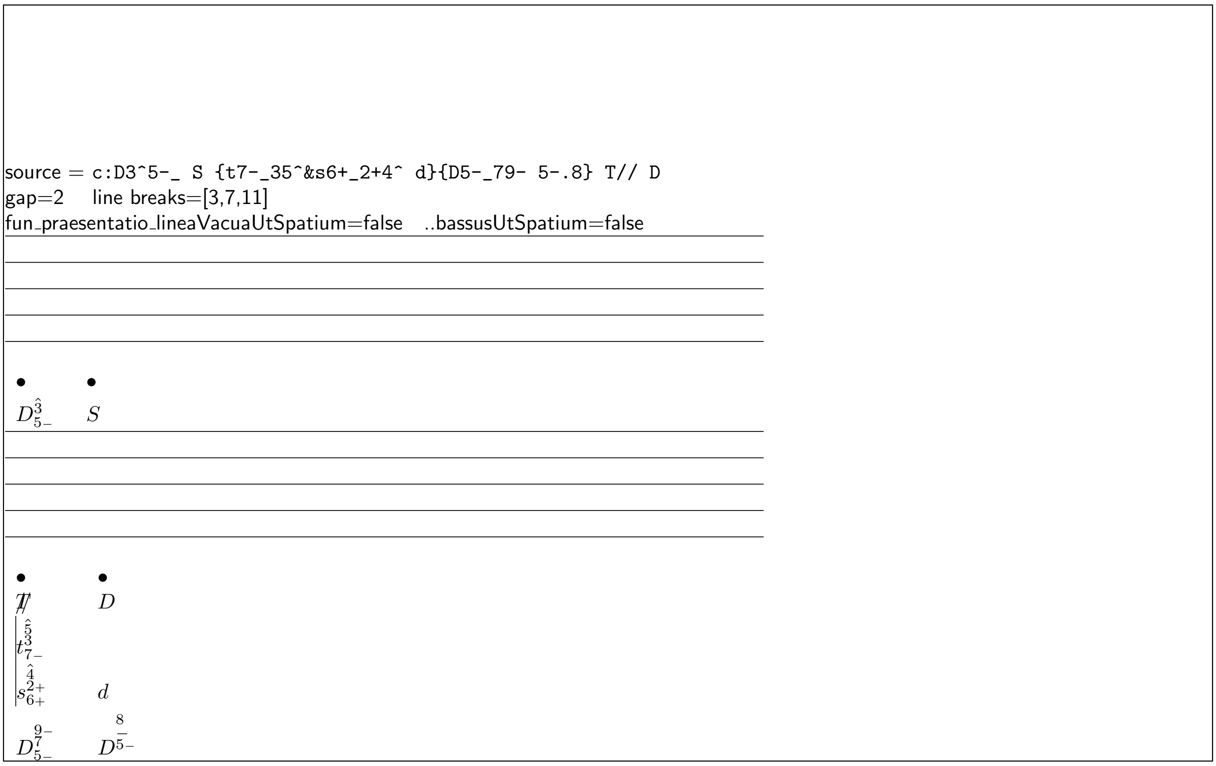
3.6 Alternative Top-Level Syntax

As an alternative to the linear encoding of the sub-track relation, using "<", "**{..}**", etc., a two-dimensional text input similar to the conventional arrangement C2DA could be considered. Its realisation is not easy: The x-coordinate must be finer than the mere "score positions", but additionally represent the different nestings of sub-tracks and relative regions, see the first examples in Figure 2.4.

When decoding such an input, a numeric style parameter "gap width" is required: With the opposite task, the calculation of a graphical layout as described in section 3.5, it gives the minimum distance between two tracks, required for sharing the same y coordinate. Here it gives the maximum distance of two non-whitespace entries on the same y coordinate to be recognised as part of the same track.

```
% tracks are identified by their node number, as above.
%!track_positions(+Track:int,-Start:int,-End:int)
%!sub_tracks(+Track:int,-Subs:list)
% assume the list of subtracks is sorted by ascending start position
%!min_gap(++D,Width:int)
%!track_row(+Track:int,-Row:int)

% assume top most track is always in node #1.
calculate_layout(D,Gap) :-
    retractall(track_row(D,_,_)),
    asserta(track_row(D,1,1)),
    sorted_sub_tracks(D,1,Subs),
    alloc(D,Gap,2,Subs).

alloc(_D,_Gap,_Row,[]) :- !.

alloc(D,Gap,Row,[Sub|Rest]) :-
    fun_node(D,Sub,_,ScorePos,track(_,_,_)),
    plus(Start,Gap,ScorePos),
    make_space(D,Gap,Row,Start),
    asserta(track_row(D,Sub,Row)),
    sorted_sub_tracks(D,Sub,Subs),
    plus(Row,1,NextRow),
    alloc(D,Gap,NextRow,Subs),
    alloc(D,Gap,Row,Rest).

sorted_sub_tracks(D,Track,Subs) :-
    findall(Scorepos-Node,fun_node(D,Node,Track,Scorepos,track(_,_,_)),Pairs),
    reverse(Pairs,Reversed),
    keysort(Reversed,Sorted),
    pairs_values(Sorted,Subs).

make_space(D,Gap,Row,Start) :-
    findall(T, conflicting(D,Row,Start,T),Conflicting),
    make_space2(D,Gap,Row,Conflicting).

make_space2(_D,_Gap,_Row,[]) :- !.
make_space2(D,Gap,Row,Conflicting) :-
    maplist([X,Y]>>(fun_node(D,X,_,Y,_)),Conflicting,Starts),
    min_list(Starts,Min),
    plus(NextStart,Gap,Min),
    plus(Row,1,NextRow),
    make_space(D,Gap,NextRow,NextStart),
    forall(member(C,Conflicting),(retract(track_row(D,C,Row)),asserta(track_row(D,C,NextRow)))).

conflicting(D,Row,Start,Track) :-
    track_row(D,Track,Row),
    % ATTENTION track_end values are inclusive.
    track_end(D,Track,_,End),
    End >= Start.
```

Listing 3.3: Track Layout Allocation



```
source = c:D3^5-_ S {t7-_35^&s6+_2+4^ d}{D5-_79- 5-.8} T// D
gap=2    line breaks=[3,7,11]
fun_praesentatio_lineaVacuaUtSpatium=false   ..bassusUtSpatium=false
```

Figure 3.2: Example of the experimental LaTeX rendering

Thanks to the stratified architecture of funCode, all other syntax and semantics stay the same, once the track information has been extracted and controls the parsing process, esp. the interval inheritance from a sound label to its successor.

3.7 An Experimental LaTeX Back-End

Listing 3.4 to Listing 3.7 show experimental code for translating some funCode data into a TeX/LaTeX rendering, called *current implementation* in the following. Procedure generate_latex_files /5 generates a text file with the given name which can be processed into a PDF result. This presents the sequence of functional labels as a sequence of two-dimensional graphical symbols under a place-holding musical staff and bullet symbols • representing the score time points. The track layout algorithm from section 3.5 is called first, supplied with its parameter Gap. The line breaks must be supplied explicitly as a list of score positions (=start of the lines) in the parameter Linebreaks.

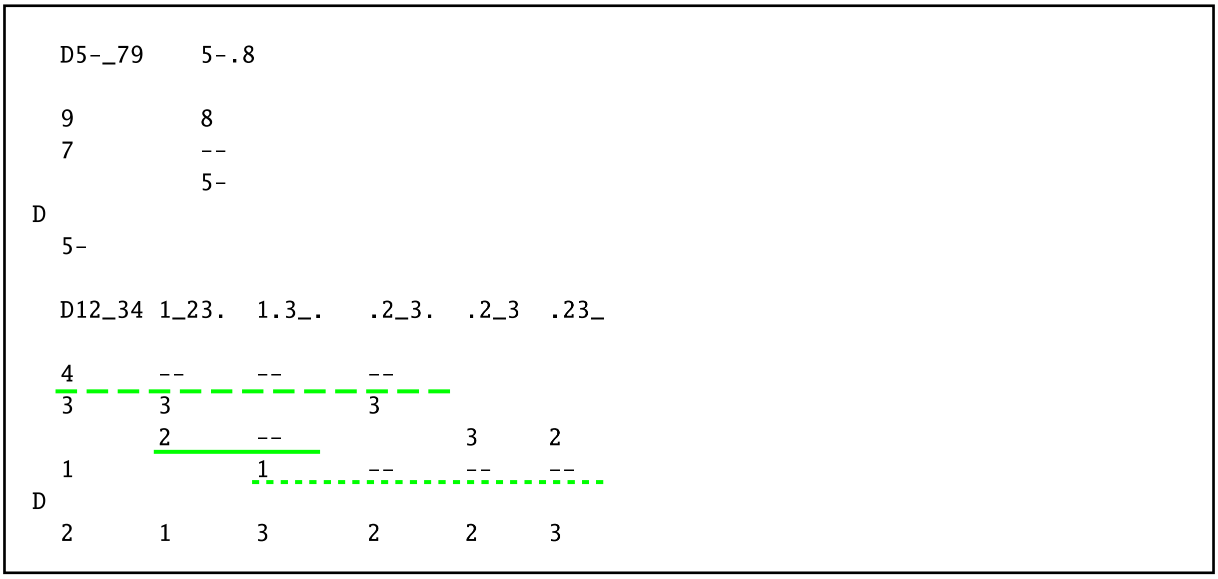
An example for the generated graphics is presented Figure 3.2.

The current implementation is far from complete, but not more than a first experimental start. Not yet supported are . . .

- printing of track headers, i.e. track name and tonic centre
- leaving out inherited root symbols, as is “**`D4 3`**”. (Even the necessary information in the sound/7 data structure is still missing.)
- generating ligatures like 𝔻
- support of “**`[..]`**”
- changing the font for root symbols, e.g. to “sans serif”
- the heureka! operator “**`!`**”
- the space and the idem labels “**`~`**” and “**`-`**”

3.7.1 The Problem of a “Jumping Bass”

A fundamental problem has not yet been resolved by the current implementation: In the funCode source text all interval numbers form a single horizontal sequence and bass and melody pitches are indicated by added decorations.



```
   D5-_79    5-.8

   9         8
   7         --
             5-
 D
   5-

   D12_34 1_23.  1.3_.   .2_3.  .2_3  .23_

   4      --     --      --
   3      3              3
          2      --             3     2
   1             1       --     --    --
 D
   2      1      3       2      2     3
```

Figure 3.3: Examples of the Jumping Bass Problem

Contrarily, in C2DA the bass pitch appears as subscript[2] other pitches in a vertical stack in superscript. When applying current implementation a labelling like

“**D5-_79- 5-.8**”

then the “inherit interval” symbol “–” in the rendering (corresponding to the “**.**” operator in the source) is wrongly aligned with the “**9-**” instead of the “**7**”, see the very last two number stacks in Figure 3.2.

A possible solution is to repeat the interval number of the bass pitch class in the vertical stack, printing it in ”phantom” mode, i.e. ”white on white”, see the top example in Figure 3.3. In the current implementation, this “phantom mode” is enabled by the global style parameter fun_praesentatio_bassusUtSpatium/0. It can be switched on globally by default, but its results are sometimes ugly and confusing. Switching it on automatically only for sections in which it is needed, is complicated because non-local analysis is required. See the lower lines in Figure 3.3: phantom mode is necessary as soon as in the sequence of an interval number and its subsequent inherit symbols “**.**” there are both positions with and without a bass note preceding in the source text (= lower in the stack, see the solid line).

Once activated, phantom mode must be extended to all overlapping inheritance sequences completely, even if they do not fall under this condition themselves. This extends not only into the future but even to the past (see dashed line). But this extension only applies to intervals *later* in the source text sequence (=*higher* in the stack). The others are not necessarily affected (see dotted line).

Different solutions to the jumping bass problem (like a complete re-ordering of the interval numbers in the rendering) are possible and should be explored.

3.8 Translation of Root Symbols to LPR Transformations

Listing 3.8 shows the code to convert sequences of funCode root symbols into transformation codes of LPR style, as defined by Hyer (1989, pg. 175).

First step is to make the changes of mode (minor vs. major) explicit, which are implicit in the GM-style notation of funCode. Additionally to Table 2.4 and line 811 in Listing 2.26, this must also be applied to sequences like “**SpD**”, because “**D**” has different translations when applied to major or minor chords (namely “**LR**” and “**RLP**” or “**PLR**”). For “**S**” and “**D**” the rules are naturally complementary: An implicit change of mode is indicated by subsequent characters of *different* case.

When translating we assume that the code shall be *applied to the major triad of the tonic centre*. Therefore “**T**” is ignored and “**t**” translated to “**R**”.

---

[2] This way of writing has been proposed by Capellen and adopted by many German theorists.(Imig, 1970, pg. 143)

```
% assume Linebreaks are 1-based and contain only the starts of
% all lines (in score positions) but the very first , sorted ascending.
% Redundant tail will be cut off .
generate_latex(D,Gap,Linebreaks,Result) :-
     calculate_layout (D,Gap),
     last_score_pos(D,LastScorePos),
%    track_end(D,1,_,LastScorePos),
     plus(LastScorePos,1,Limit),
     include(>(Limit), Linebreaks, Valids ),
     append(Valids,[Limit],[ Next|Lbs]),
     genLT_line(D,1,Next,Lbs ,[], Result_as_list ),
     flatten ( Result_as_list ,Result).


generate_latex_files (D,Source,Gap,Linebreaks,FilenameStub) :-
     parse_string(td,Source),
     generate_latex(D,Gap,Linebreaks,Result),
     concat(FilenameStub, ".tex",FilenameShort),
     concat(FilenameStub, "-testbed.tex",FilenameLong),
     open(FilenameShort,write,StreamShort),
     open(FilenameLong,write,StreamLong),
     write(StreamShort,Result),
     write(StreamLong,"\\documentclass{article}\n"),
     write(StreamLong,"\\pagestyle{empty}\n"),
%% ??   write(StreamLong,"\\def\\funcodeStartline\\relax"),
     write(StreamLong,"\\newcommand{\\funcodeStartline}{\n"),
     write(StreamLong,"␣␣\\begin{tabular}{p{\\textwidth}}\n"),
     write(StreamLong,"␣␣␣␣\\hline\\\\␣\\hline\\\\␣\\hline\\\\␣\\hline\\\\␣\\hline\\\\\n"),
     write(StreamLong,"␣␣\\end{tabular}\\\\}\n"),
     write(StreamLong,"\n"),
     write(StreamLong,"\\renewcommand{\\familydefault}{cmss}\n"),
     write(StreamLong,"\n"),
     write(StreamLong,"\\begin{document}\n"),
     get_style (fun_praesentatio_lineaVacuaUtSpatium, FPLVUS),
     get_style (fun_praesentatio_bassusUtSpatium, BSPAT),
     format(string (LineParams),
          "\\noindent␣source␣=␣\\verb$~w$\\\\\n␣gap=~w␣\\quad␣line␣breaks=~w\\\\",
          [Source,Gap,Linebreaks]),
     write(StreamLong,LineParams),
     format(string (LineParams2),
          "fun\\_praesentatio\\_lineaVacuaUtSpatium=~w\\quad..bassusUtSpatium=~w\\\\\n",
          [FPLVUS,BSPAT]),
     write(StreamLong,LineParams2),
     write(StreamLong,Result),
     write(StreamLong,"\\end{document}\n"),
     close(StreamShort),
     close(StreamLong).
```

Listing 3.4: Generate Experimental LaTeX Rendering

```
% generate source for one array=one score line
%@param +Start=Scorepos
%@param +Limit=first Scorepos on next line
%@param +Rest=List of further line start score poss
%@param -Result
genLT_line(D,Start, Limit ,Rest,In,Result) :-
    plus(End,1,Limit),
     findall (Track,  (fun_node(D,Track,_,TStart,_),  track_end(D,Track,_,TEnd),
                        common_interval(Start,End,TStart,TEnd)),
              Tracks),
    plus(Start,Count,Limit),
    plus(Count,1,CountP1),
     string_multi ("\\bullet␣&",Count,Bullets),
    format(string (Line0),"%~n%line␣of␣timepoints␣~d␣to␣~d␣(excl.)~n", [Start, Limit]),
    format(string (Line1),
          "\\funcodeStartline{}~n$\\begin{array}{*{~d}l}~n␣~w␣␣\\\\~n",
          [CountP1, Bullets]),

    maplist(track_row(D),Tracks,Rows),
    sort(Rows,RowsSorted),
    ((RowsSorted=[R|_],!,genLT_track(D,Start,Limit,R,RowsSorted,[], Sub0Result));
     (Sub0Result=[])),
    genLT2_line(D,Limit,Rest, [In,Line0,Line1,Sub0Result,"\\end{array}$\\\\\n"], Result).

common_interval(S,E,S2,E2) :- E>=S2, E2>=S.

 string_multi (_S,0,"") :- !.
 string_multi (S,M,Res) :- plus(P,1,M), string_multi(S,P,Pref),  string_concat(S,Pref,Res).

 flatten  ([], "") :-!.
 flatten (S,S) :- string (S),  !.
 flatten (S,SS) :- atom(S), !,  atom_string(S,SS).
 flatten (S,SS) :- integer(S),  !,  number_string(S,SS).
 flatten ([A|B],Result)  :-  flatten (A,As),  flatten (B,Bs),  string_concat(As,Bs,Result).

genLT2_line(D,Limit,[Next|Rest],In,Result)  :-
     !,  genLT_line(D,Limit,Next,Rest,In,Result).
genLT2_line(_D,_Limit ,[], Result,Result).

genLT_track(D,Start,Limit,R,[R|Rest],In,  [Line0,  SubResult, "\\\\\n" | RestResult]) :- !,
    format(string (Line0),  "%----␣Row␣~d~n",R),
    genLT_cell(D,Start, Limit, R ,[], SubResult),
    plus(R,1,R1),
    genLT_track(D,Start,Limit,R1,Rest,In,  RestResult).
genLT_track(_D,_Start, _Limit, _R ,[], In,  In)  :-  !.

% special case: track  is  empty in  this  score  line :
genLT_track(D,Start,Limit,R,OtherRest,In, [MyResult|RestResult]) :-
    genEmtpyLine(D,MyResult),
    plus(R,1,R1),
    genLT_track(D,Start,Limit,R1,OtherRest,In, RestResult).
genEmtpyLine(_D,MyResult) :-
    fun_praesentatio_lineaVacuaUtSpatium, !, MyResult= "\\phantom{D}\\\\\n".
genEmtpyLine(_D,MyResult) :-
    MyResult="".
```

Listing 3.5: Generate Experimental LaTeX Rendering – II

```
genLT_cell(_D,Start,Start,_R,In,In)  :-!.
genLT_cell(D,Start,Limit,R,In,  Result) :-
    format(string(Line0), "%--------␣Score␣Time␣Point␣~d~n",Start),
    track_row(D,T,R), genLT_cell2(D,T,Start,SubResult),
    plus(Start,1,Start1),
    genLT_cell(D,Start1,Limit,R, [In,Line0,SubResult], Result).

genLT_cell2(D,T,Score, Result) :-
    fun_node(D,_,T,Score,sum([OneSound])), !,
    genLT_sounds([OneSound],[],SubResult),
    Result=["␣\\begin{array}[b]{@{}l}␣",SubResult,"␣\\end{array}␣&\n"].

genLT_cell2(D,T,Score, Result) :-
    fun_node(D,_,T,Score,sum(Sounds)), !,
    genLT_sounds(Sounds,[],SubResult),
    Result=["␣\\begin{array}[b]{|@{}l}␣",SubResult,"␣\\end{array}␣&\n"].

genLT_cell2(_D,_,_, "␣&\n").


genLT_sounds([],In,In).
genLT_sounds([sound(_Root,_Pitches,_Base,_Mel,RamSource,IntSource,Suppress)|Rest],
             In,[RamSource,Supp,IResult,"\\\\",RestResult]) :-
    genLT_suppress(Suppress,Supp),
    genLT_intervalsB(IntSource,IntSource,IResult),
    genLT_sounds(Rest,In,RestResult).

genLT_suppress(0,"") :-!.
genLT_suppress(1,"\\kern-1.5ex/") :-!.
genLT_suppress(2,"\\kern-1.7ex/\\kern-0.5ex/") :-!.
```

Listing 3.6: Generate Experimental LaTeX Rendering – III

```
% search intervals for bass note:
genLT_intervalsB ([], IntSource,
                ["_{\\phantom9}␣{\\begin{array}{@{}l}",Result,"\\end{array}}"]) :-
    !, genLT_intervals(IntSource,"[-1.0ex]",Result).
genLT_intervalsB([ [_,_,false|_] | R],IntSource,Result) :-
    !, genLT_intervalsB(R,IntSource,Result).
genLT_intervalsB([ [Value,Modif,true,_] | _], IntSource,
                ["_{",Value,ModifText,"}␣{\\begin{array}{@{}l}",Result,"\\end{array}}"]) :-
    !, genLT_modifs(Modif,ModifText), genLT_intervals(IntSource,"[-0.7ex]",Result).
genLT_intervalsB([ [_Value,_Modif,true,_, inherit ] | _], IntSource,
                ["_{-\\phantom9}␣{\\begin{array}{@{}l}",Result,"\\end{array}}"]) :-
    !, genLT_intervals(IntSource,"[-1.0ex]",Result).

genLT_intervals ([], _Dist ,[]).

genLT_intervals ([ [Value,Modif,false,false] | R] ,Dist,
               [Result,"\\scriptstyle",Value,ModifText,"\\\\",Dist]) :-
    !, genLT_modifs(Modif,ModifText), genLT_intervals(R,"[-1.5ex]",Result).
genLT_intervals ([ [_Value,_Modif,false,_, inherit ] | R],Dist,
               [Result,"\\scriptstyle-\\phantom9\\\\",Dist])
 :- !, genLT_intervals(R,"[-1.3ex]",Result).
genLT_intervals ([ [Value,Modif,false,true] | R] ,Dist,
               [Result,"\\hat{\\scriptstyle",Value,ModifText,"}\\\\",Dist]) :-
    !, genLT_modifs(Modif,ModifText), genLT_intervals(R,"[-1.5ex]",Result).

genLT_intervals ([ [Value,_,true|_] | R],Dist,
               [Result,"\\scriptstyle\\phantom{",Value,"}\\\\",Dist]) :-
   fun_praesentatio_bassusUtSpatium, !, genLT_intervals(R,"[-1.5ex]",Result).
genLT_intervals ([ [_Value,_,true|_] | R],Dist, Result) :-
   genLT_intervals(R,Dist,Result).

genLT_modifs ([],[]) :- !.
genLT_modifs(Modifiers,["{\\scriptscriptstyle", Modifiers,"}"]) :- !.
```

Listing 3.7: Generate Experimental LaTeX Rendering – IV

```
%! normalize_LPR(+In:ListOfRootFunctionCodes,−Out:dto)
normalize_LPR([X], [X])  :−  !.

normalize_LPR([A, B | R], [A, C | S]) :−
    normalize_LPR(A, B, C, D), !,
    normalize_LPR([D | R], S).

normalize_LPR([A | R], [A | S]) :−
    normalize_LPR(R, S).

normalize_LPR(X, 'G', 'g', '↑') :− fun_upper(X).
normalize_LPR(X, 'P', 'p', '↑') :− fun_upper(X).
normalize_LPR(X, 'g', 'G', '↓') :− fun_lower(X).
normalize_LPR(X, 'p', 'P', '↓') :− fun_lower(X).
normalize_LPR(X, 'd', 'D', '↓') :− fun_upper(X).
normalize_LPR(X, 's', 'S', '↓') :− fun_upper(X).
normalize_LPR(X, 'D', 'd', '↑') :− fun_lower(X).
normalize_LPR(X, 'S', 's', '↑') :− fun_lower(X).

translate_LPR(A,B) :−
normalize_LPR(['T'|A],AN), tr_LPR(AN,B).

tr_LPR ([],    []).
tr_LPR(['T'|A], B) :− tr_LPR(A,B).
tr_LPR(['t'|A], ['P'|B]) :− tr_LPR(A,B).

tr_LPR(['G'|A], ['L'|B]) :− tr_LPR(A,B).
tr_LPR(['g'|A], ['L'|B]) :− tr_LPR(A,B).
tr_LPR(['P'|A], ['R'|B]) :− tr_LPR(A,B).
tr_LPR(['p'|A], ['R'|B]) :− tr_LPR(A,B).
tr_LPR(['↑'|A], ['P'|B]) :− tr_LPR(A,B).
tr_LPR(['↓'|A], ['P'|B]) :− tr_LPR(A,B).
tr_LPR(['D'|A], ['L','R'|B]) :− tr_LPR(A,B).
tr_LPR(['d'|A], ['R','L'|B]) :− tr_LPR(A,B).
tr_LPR(['S'|A], ['R','L'|B]) :− tr_LPR(A,B).
tr_LPR(['s'|A], ['L','R'|B]) :− tr_LPR(A,B).
```

Listing 3.8: Converting Functional Roots to LPR Modifiers

Bibliography

Broy, M. (2011). Informatik als wissenschaftliche Methode: Zur Rolle der Informatik in Forschung und Anwendung. In *41. Jahrestagung der Gesellschaft für Informatik*, volume P192 of *LNI*, pages 43–44, Berlin.

Cohn, R. (2011). Tonal pitch space and the (neo-)Riemannian Tonnetz. In Gollin and Rehding (2011).

Cohn, R. (2012). *Audacious Euphony — Chromaticism and the Triad's Second Nature*. Oxford Press.

Euler, L. (1774). De harmoniae veris principiis per speculum musicum repraesentatis. *Novi Commentarii academiae scientiarum Petropolitanae*, 18.

Gollin, E. (2011). From matrix to map: Tonbestimmung, the Tonnetz, and Riemann's combinatorial conception of interval. In Gollin and Rehding (2011).

Gollin, E. and Rehding, A., editors (2011). *The Oxford Manual of Neo-Riemannian Music Theories*. Oxford Press, New York, NY.

Grabner, H. (1923). *Die Funktionentheorie Hugo Riemanns und ihre Bedeutung für die praktische Analyse*. Leuckart, München.

Hauptmann, M. (1873). *Die Natur der Harmonik und Metrik*. Breitkopf und Härtel, Leipzig.

Heetderks, D. (2015). From uncanny to marverlous: Poulenc's hexatonic pole. *Theory and Practice*, 40:177–204. `https://www.jstor.org/stable/10.2307/26477736`.

Hewitt, M. (2000). *The Tonal Phoenix*. Orpheus Verlag, Bonn.

Hussong, H. (2005). *Untersuchungen zu praktischen Harmonielehren seit 1945*. Verlag im Internet GmbH, Berlin.

Hyer, B. (1989). *Tonal Intuitions in Tristan und Isolde*. University Microfilms International, Ann Arbor, MI.

Imig, R. (1970). *Systeme der Funktionsbezeichnung in den Harmonielehren seit Hugo Riemann*. Gesellschaft zur Förderung der systematischen Musikwissenschaft e.V., Düsseldorf.

Iso (1996). Iso/iec 13211: Programming languages — Prolog.

Kopp, D. (2011). Chromaticism and the question of tonality. In Gollin and Rehding (2011).

Lepper, M. (2021). *de Linguis Musicam Notare — Beiträge zur Bestimmung von Semantik und Stilistik moderner Musiknotation durch mathematische Remodellierung*. epOs, Osnabrück.

Lepper, M. and Trancón y Widemann, B. (2011). d2d — a robust front-end for prototyping, authoring and maintaining xml encoded documents by domain experts. In *Proceedings 3rd International Joint Conference on Knowledge Engineering and Ontology Development (KEOD 2011)*, pages 449–456. SciTePress Digital Library.

Lewin, D. ([1984]2006). Amfortas prayer to titurel and the role of d in parsifal. In *Studies in Music with Text*. Oxford University Press.

Lindholm, T., Yellin, F., Bracha, G., Buckley, A., and Smith, D. (2018). *The Java Virtual Machine Specification*, volume 384 of *JSR*. Oracle, Java SE 11 edition.

Maler, W. (1931). *Beiträge zur durmolltonalen Harmonielehre*. Leuckart, München.

Nápoles López, N. and Fujinaga, I. (2020). Harmalysis: A Language for the Annotation of Roman Numerals in Symbolic Music Representations. In De Luca, E. and Flanders, J., editors, *Music Encoding Conference Proceedings 2020*, pages 83–85. Humanities Commons.

Neuwirth, M., Harasim, D., Moss, F. C., and Rohrmeier, M. (2018). The Annotated Beethoven Corpus (abc): A dataset of harmonic analyses of all Beethoven string quartets. *Frontiers In Digital Humanities*.

Ploeger, R. (1990). *Studien zur systematischen Musiktheorie*. Eres, Lilienthal, Bremen.

Riemann, H. (1918). *Handbuch der Harmonielehre*. Leipzig.

Schenker, H. (1906). *Harmonielehre*. Univeral Edition, Wien.

Vogel, M. (1975). *Die Lehre von den Tonbeziehungen*. Verlag für systematische Musikwissenschaft, Bonn-Bad Godesberg.

Weber, G. (1817). *Versuch einer geordneten Theorie der Tonsetzkunst*. Mainz.

Appendix A

All current test codes

```

all_tests  :- run_tests(consistent),
              run_tests(store_reference),
              run_tests(backtabs),
              run_tests( track_toplevel ),
              run_tests(tonic_center),
              run_tests( tonic_locale ),
              run_tests( illegal_relRegs ),
              run_tests(items_linear),
              run_tests( items_relative ),
              run_tests(track_name),
              run_tests(fun_sounds),
              run_tests(funSound_intervals_only),
              run_tests(fun_sound_1),
              run_tests(fun_sound_2),
              run_tests( check_inherit_intervals ),
              run_tests(  inherit_intervals  ),
              run_tests(root_and_mode),
              run_tests(root_and_mode_fancy),
              run_tests(function_macros),
              run_tests( intervals_greater_10 ),
              run_tests( intervals ),
              run_tests(extract_base_and_mel),
              run_tests( interval_pitch ),
              run_tests( default_intervals ),
              run_tests(  default_intervals_in_context  ),
              run_tests(normalize_functions),
              run_tests(mdom),
              run_tests( retrieval ),
              run_tests(retrieval_sum),
              run_tests(layout),
              run_tests(latex),
              run_tests( translate_lpr ).
```

```

:- begin_tests(consistent).
test(track_Ok1) :- consistent(track   ([],[],[])).
test(track_Ok2) :- consistent(track("trackName",[],[])).
test(track_Ok3) :- consistent(track  ([],[ 'c','#','\''],euler(3,3))).

test( track_fail1 ) :- \+ consistent(track("trackName",['C'],euler([],2))).
test( track_fail2 ) :- \+ consistent(track("trackName",[],euler(1,2))).

```

```
test(sound_Ok1) :- consistent(sound(euler(1,2),[],undef,undef ,[],[],[],2)).
test(sound_fail1) :- \+ consistent(sound(euler (1,2),[], undef ,[],[],[],[],2)).

test(sum_Ok1) :- consistent(sum([sound(euler(1,2),[],undef,undef ,[],[],[],2)])).
test(sum_fail1) :- \+ consistent(sum ([])).

test(itr_Ok1) :- consistent( interrupted_track (1,2,[3,4,5])).
test( itr_fail1 ) :- \+ consistent( interrupted_track (1,2,[3, inherit ,5])).
:- end_tests(consistent).
```

```

:- begin_tests(store_reference).
test(simple) :-
    fun_node_retract(td),
    store_reference(td ,3,5,9),
     relative_root (td ,3,9), relative_root (td ,4,9), relative_root (td ,5,9),
     findall (td, relative_root (td,_,_), Stored),
    length(Stored,3).
%%% \+ relative_root (td,2,_), \+ relative_root (td,_,6), \+ relative_root (td,_,10).
:- end_tests(store_reference).
```

```

 :- begin_tests(backtabs).
test(simple) :- atom_chars("<<",X),phrase(backtabs(2),X).

test(pop0) :- poptabs(td, [3,4,5],0,3).
test(pop1) :- poptabs(td, [3,4,5],1,4).
test(pop2) :- poptabs(td, [3,4,5],2,5).

test(pop3) :- set_error_pos(td, 10,11,12), poptabs(td, [3,4,5],3, _),
              fun_error(td, 10,11,12,"undefined␣tab␣stop,␣too␣many␣<␣signs",[]).
test(tab_track) :-
    parse_string(td, "D␣<{S}"),
    fun_node(td,1,0,1,track(_,_,_)),
    fun_node(td,3,1,1,track(_,_,_)).
test(tab_back1) :-
    parse_string(td, "D␣>␣T␣>␣S␣<{S}"),
    fun_node(td,5,1,3,track(_,_,_)),
    fun_node(td,6,5,3,sum(_)).
test(tab_back2) :-
    parse_string(td, "D␣>␣T␣>␣S␣<<{S}"),
    fun_node(td,5,1,2,track(_,_,_)),
    fun_node(td,6,5,2,sum(_)).
test(tab_back_error) :-
    \+ parse_string(td, "D␣>␣T␣<<S"),
     fun_error(td ,4,1,3, "undefined␣tab␣stop,␣too␣many␣<␣signs", []).
test(tab_warning) :-
    parse_string(td, "D␣>␣>␣T"),
    fun_warning(td,3,1,2, "multiple␣set␣of␣same␣tab␣stop", []).
:- end_tests(backtabs).
```

```

:- begin_tests( track_toplevel ).
test( title_only ) :-
    fun_node_retract(td),
    atom_chars("\"main\"",X), phrase(parse_track(td, 1, 0, 30, []), X),
    fun_node(td, 1,0,30,track("main" ,[],[])).
```

```
test(title_and_one_func) :-
    fun_node_retract(td),
    set_interval_defaults_none,
    atom_chars("\"main\"␣␣D",X), phrase(parse_track(td, 1, 0, 30, []), X),
    fun_node(td, 1,0,30,track("main",[],[])),
    fun_node(td, 2,1,30,sum([sound(euler(1,0),[], undef, undef, ['D'],[],0)])).
test(key_and_one_func) :-
    fun_node_retract(td),
    set_interval_defaults_none,
    set_style(fun_initiaSyntonica, true),
    atom_chars("F#,:␣␣D/",X), phrase(parse_track(td, 1, 0, 30, []), X),
    fun_node(td, 1,0,30,track([],['F','#',','],euler(2,1))),
    fun_node(td, 2,1,30,sum([sound(euler(1,0),[], undef, undef, ['D'],[],1)])).

test(key_name_and_one_func) :-
    fun_node_retract(td),
    set_interval_defaults_none,
    atom_chars("\"main\"␣F#:␣␣D//",X), phrase(parse_track(td, 1, 0, 30, []), X),
    fun_node(td, 1,0,30,track("main",['F','#'],euler(6,0))),
    fun_node(td, 2,1,30,sum([sound(euler(1,0),[], undef, undef, ['D'],[],2)])).


test(heureka_ok) :-
    fun_node_retract(td),
    atom_chars("D␣!T", X), phrase(items(td, 11, 1, 31, [], []), X),
    \+ fun_error(td, _,_,_,_,_),
    fun_heureka(td,1,12,32).

test(heureka_fail) :-
    fun_node_retract(td),
    atom_chars("D␣!T␣!s", X), phrase(items(td, 11, 1, 31, [], []), X),
    fun_heureka(td,1,12,32),
    findall( [A,C,D] ,fun_error(td,A,_B,C,D,_E),Errors),
    Errors=[ [13,33,"More␣than␣one␣heureka/!␣operators"] ].

:- end_tests( track_toplevel ).
```

```

:- begin_tests(tonic_center).
test(c) :- phrase(whiteKey(['C'],['C']), ['C']).
test(lower_c) :- phrase(whiteKey(['c'],['C']), ['c']).
test(x) :- \+ phrase(whiteKey(_,_), ['X']).

test(dsharp) :- string_chars("d#:", X), phrase(tonicCenterSpec(['d','#'],['D'],['#'],[]), X).
test(h) :- \+ phrase(tonicCenterSpec(_,_,_,_), ['H',':']).
test(eflat_commasdown) :-
    string_chars("eb,,:", X), phrase(tonicCenterSpec(['e','b'],['E'],['b'],[',',',']), X).

test(eflat_commaswrong) :- string_chars("Eb,\':", X), \+ phrase(tonicCenterSpec(_,_,_,_), X).
test(a_bothaccidentals) :- string_chars("A#b:", X), \+ phrase(tonicCenterSpec(_,_,_,_), X).

test(multiacc_allowed) :-
    set_style(fun_accidensRepetendum,true),
    parse_string(td,"c##:D").
test(multiacc_forbidden) :-
    set_style(fun_accidensRepetendum,false),
    \+ parse_string(td,"c##:D"),
    fun_error(td,1,0,1, "more␣than␣one␣accidens␣requires␣fun_accidensRepetendum",[]).
```

```

test (syntonic_allowed) :-
     set_style ( fun_initiaSyntonica ,true),
     parse_string(td,"c'':D").
test (syntonic_forbidden) :-
     set_style ( fun_initiaSyntonica ,false),
     \+ parse_string(td,"c'':D"),
     fun_error (td ,1,0,1, "comma␣not␣allowed␣with␣tonic␣center␣by␣style␣parameter␣fun_initiaSyntonica", []).

test ( eval_fis ) :- tonic_center_to_euler (['F','#'],euler(6,0)).
test (eval_es_synt_allowed) :-
     set_style ( fun_initiaSyntonica , true),
     fun_node_retract(td ),
     tonic_center_to_euler (['E','b',','],euler(-7,1)).
:- end_tests(tonic_center ).
```

```

:- begin_tests( tonic_locale ).
test (es) :-
     select_locale_for_key ('DE'),
     parse_string(td, "Es:D"),
     fun_node(td,1,0,1,track ([], ['E','s'], euler(-3,0))).

:- end_tests( tonic_locale ).
```

```

:- begin_tests( illegal_relRegs ).
test (case_a) :-
     fun_node_retract(td ),
     atom_chars("D␣(:D␣D:)␣D",X), phrase(items(td, 2, 1, 30, [], []), X),
     fun_error (td ,5,1,33, "relative␣region␣cannot␣look␣to␣both␣sides",[]).
test (case_b) :-
     fun_node_retract(td ),
     atom_chars("D␣(D)␣(:D)",X), phrase(items(td, 2, 1, 30, [], []), X),
     fun_error (td ,4,1,32, "adjacent␣right-␣and␣left-looking␣parentheses",[]).
test (case_c) :-
     fun_node_retract(td ),
     atom_chars("D␣(D␣(D))␣D",X), phrase(items(td, 2, 1, 30, [], []), X),
     fun_error (td ,5,1,33, "adjacent␣right-looking␣parentheses",[]).
test (case_d) :-
     fun_node_retract(td ),
     atom_chars("D␣(:(:D)␣D)",X), phrase(items(td, 2, 1, 30, [], []), X),
     fun_error (td ,3,1,31, "adjacent␣left-looking␣parentheses",[]).
test (case_e) :-
     fun_node_retract(td ),
     atom_chars("D␣((:D)␣D)␣D",X), phrase(items(td, 2, 1, 30, [], []), X),
     fun_error (td ,3,1,31, "adjacent␣right-␣and␣left-looking␣open␣parentheses",[]).
test (case_f) :-
     fun_node_retract(td ),
     atom_chars("D␣(:D␣(D))␣D",X), phrase(items(td, 2, 1, 30, [], []), X),
     fun_error (td ,5,1,33, "adjacent␣right-␣and␣left-looking␣close␣parentheses",[]).
:- end_tests( illegal_relRegs ).
```

```

% Node Track Score
:- begin_tests( items_linear ).
% ATTENTION track must be = 1 for the finishing parser rule
test ( finished ) :- phrase(items(td, 2, 1, 30, [], []), []).
```

```
test(one_space) :-
    fun_node_retract(td),
    phrase(items(td, 2, 1, 30, [], []), ['~']),
    fun_node(td, 2,1,30,space).
test(space_idem) :-
    fun_node_retract(td),
    \+ phrase(items(td, 2, 1, 30, [], []), ['~', '-']).
test(d_space_idem) :-
    fun_node_retract(td),
    phrase(items(td, 2, 1, 30, [], []), ['D','/','~','-']),
    fun_node(td, 2,1,30,sum([sound(euler(1,0),_,_,_,['D'],[],1)])),
    fun_node(td, 3,1,31,space),
    fun_node(td, 4,1,32,idem).
test(cut_on_t) :-
    fun_node_retract(td),
    % attention: default intervals are defined globally and may change.
    % therefore the are matched with '_' = 'don't care' in the following:
    phrase(items(td, 2, 1, 30, [], []), ['D', 's', 't']),
    fun_node(td, 2,1,30,sum([sound(euler(0,0),_, undef, undef, ['D','s'], [], 0)])),
    fun_node(td, 3,1,31,sum([sound(euler(0,0),_, undef, undef, ['t'], [], 0)])).
test(two_times_two) :-
    fun_node_retract(td),
    atom_chars("D&s␣S&t",X), phrase(items(td, 2, 1, 30, [], []), X),
    fun_node(td, 2,1,30,sum([sound(euler(1,0),_, undef, undef, ['D'], [], 0),
                             sound(euler(-1,0),_, undef, undef, ['s'], [], 0)])),
    fun_node(td, 3,1,31,sum([sound(euler(-1,0),_, undef, undef, ['S'], [], 0),
                             sound(euler(0,0),_, undef, undef, ['t'], [], 0)])).
test(inherit_root) :-
    fun_node_retract(td),
    set_interval_defaults_none,
    atom_chars("D4␣3",X), phrase(items(td, 2, 1, 30, [], []), X),
    fun_node(td, 2,1,30,sum([sound(euler(1,0),[euler(-1,0)], undef, undef, ['D'],
                                  [ [4,[], false,false] ], 0)])),
    fun_node(td, 3,1,31,sum([sound(euler(1,0),[euler(0,1)], undef, undef, ['D'],
                                  [ [3,[], false,false] ], 0)])).
test(inherit_simple) :-
    fun_node_retract(td),
    set_interval_defaults_none,
    atom_chars("D4␣.",X), phrase(items(td, 2, 1, 30, [], []), X),
    fun_node(td, 2,1,30,sum([sound(euler(1,0),[euler(-1,0)], undef, undef, ['D'],
                                  [ [4,[], false,false] ], 0)])),
    fun_node(td, 3,1,31,sum([sound(euler(1,0),[euler(-1,0)], undef, undef, ['D'],
                                  [ [4,[], false,false, inherit] ], 0)])).
test(inherit_simple_with_defaults) :-
    fun_node_retract(td),
    set_interval_defaults_conventional,
    atom_chars("D4␣.",X), phrase(items(td, 2, 1, 30, [], []), X),
    fun_node(td, 2,1,30,sum([sound(euler(1,0),Ints, undef, undef, ['D'],
                                  [ [4,[], false,false] ], 0)])),
    fun_node(td, 3,1,31,sum([sound(euler(1,0),Ints, undef, undef, ['D'],
                                  [ [4,[], false,false, inherit] ], 0)])),
    sort(Ints,Sorted), Sorted=[euler(-1,0),euler(0,0),euler(1,0)].

test(interspersed_space) :-
    fun_node_retract(td),
    set_interval_defaults_none,
    atom_chars("D4&t3␣-~␣␣3&.", X), phrase(items(td, 2, 1, 30, [], []), X),
    fun_node(td, 2,1,30,sum([sound(euler(1,0),[euler(-1,0)], undef, undef, ['D'],
```

```
                                        [  [4,[], false,false]  ],  0),
                                sound(euler(0,0),[euler(1,-1)],  undef, undef, ['t'],
                                        [  [3,[], false,false]  ],  0)])),
    fun_node(td, 3,1,31,idem),
    fun_node(td, 4,1,32,space),
    fun_node(td, 5,1,33,sum([sound(euler(1,0),[euler (0,1)],  undef, undef, ['D'],
                                        [  [3,[], false,false]  ],  0),
                                sound(euler(0,0),[euler(1,-1)],  undef, undef, ['t'],
                                        [  [3,[], false,false, inherit ]  ],  0)])).

test(two_times_two_inherit) :-
    fun_node_retract(td),
     set_interval_defaults_none ,
    atom_chars("T6-&D7+␣T.&.", X), phrase(items(td, 2, 1, 30, [], []), X),
    fun_node(td, 2,1,30,sum([sound(euler(0,0),[euler(0,-1)],  undef, undef, ['T'],
                                        [  [6,['-'],false,false]  ],  0),
                                sound(euler(1,0),[euler (1,1)],  undef, undef, ['D'],
                                        [  [7,['+'],false,false]  ],  0)])),
    fun_node(td, 3,1,31,sum([sound(euler(0,0),[euler(0,-1)],  undef, undef, ['T'],
                                        [  [6,['-'],false,false, inherit ]  ],  0),
                                sound(euler(1,0),[euler (1,1)],  undef, undef, ['D'],
                                        [  [7,['+'],false,false, inherit ]  ],  0)])).

test(decime_inherit) :-
    fun_node_retract(td),
     set_interval_defaults_none ,
    atom_chars("DD/5-7-10-␣..9-", X), phrase(items(td, 2, 1, 30, [], []), X),
    fun_node(td, 2,1,30,sum([sound(euler(2,0),[euler(-2,-1),euler(-2,0),euler(1,-1)],
                                        undef, undef, ['D','D'],
                                        [  [5,['-'],false,false ],[7,[ '-'],false,false ],[10,[ '-'],false,false]  ],
                                         1)])),
    fun_node(td, 3,1,31,sum([sound(euler(2,0),[euler(-2,-1),euler(-2,0),euler(-1,-1)],
                                        undef, undef, ['D','D'],
                                        [  [5,['-'],false,false, inherit  ],[7,[ '-'],false,false, inherit ],
                                           [9,['-'],false,false]  ],
                                         1)])).

test(separate_13) :-
    fun_node_retract(td),
     set_interval_defaults_none ,
    atom_chars("T2+4␣1,3",X), phrase(items(td, 2, 1, 30, [],  []),  X),
    fun_node(td, 2,1,30,sum([sound(euler(0,0),[euler (2,0), euler (-1,0)],
                                        undef, undef, ['T'],
                                        [  [2,['+'],false,false  ],[4,[],  false,false ]],
                                         0)])),
    fun_node(td, 3,1,31,sum([sound(euler(0,0),[euler (0,0), euler (0,1)],
                                        undef, undef, ['T'],
                                        [  [1,[], false,false],   [3,[], false,false]  ],
                                         0)])).


test(two_dots) :-
    fun_node_retract(td),
     set_interval_defaults_none ,
    atom_chars("D3␣.␣.", X), phrase(items(td, 2, 1, 30, [],   []),  X),
    fun_node(td, 2,1,30,sum([sound(euler(1,0),[euler (0,1)],  undef, undef, ['D'],
                                      [  [3,  [],  false,false]  ],  0)])),
    fun_node(td, 3,1,31,sum([sound(euler(1,0),[euler (0,1)],  undef, undef, ['D'],
```

```
                                    [ [3, [], false,false, inherit ] ], 0)])).


test(two_dots_in_sum) :-
    fun_node_retract(td),
     set_interval_defaults_conventional ,
    atom_chars("D//1&t␣.&s␣.&D7", X), phrase(items(td, 2, 1, 30, [], []), X),
    fun_node(td, 2,1,30,sum([sound(euler(1,0),[euler (0,0)],
                                    undef, undef, ['D'], [ [1,[], false,false] ], 2),
                              sound(euler(0,0),Int1 ,
                                    undef, undef, ['t'], [], 0)
                              ])),
    fun_node(td, 3,1,31,sum([sound(euler(1,0),[euler (0,0)],
                                    undef, undef, ['D'], [ [1,[], false,false, inherit ] ], 2),
                              sound(euler(-1,0),Int2,
                                    undef, undef, ['s'], [], 0)
                              ])),
    sort(Int1 ,Sort1), Sort1=[euler(0,0),euler(1,-1),euler (1,0)],
    sort(Int2 ,Sort2), Sort2=[euler(0,0),euler(1,-1),euler (1,0)].



:- end_tests(items_linear ).
```

```

:- begin_tests( items_relative ).

test( right ) :-
    fun_node_retract(td ),
    atom_chars("T(t␣S)s", X), phrase(items(td, 10, 1, 30, [], []), X),
     relative_root (td, 11,13), relative_root (td, 12,13).

test( right_colon ) :-
    fun_node_retract(td ),
    atom_chars("T(t␣S:)s", X), phrase(items(td, 10, 1, 30, [], []), X),
     relative_root (td, 11,13), relative_root (td, 12,13).

test( left ) :-
    fun_node_retract(td ),
    atom_chars("T(:t␣S)s", X), phrase(items(td, 10, 1, 30, [], []), X),
     relative_root (td, 11,10), relative_root (td, 12,10).

test(double_colon) :-
    fun_node_retract(td ),
    atom_chars("T(:t␣S:)s", X), phrase(items(td, 10, 1, 30, [], []), X),
     fun_error (td,13,1,33,"relative␣region␣cannot␣look␣to␣both␣sides",[]).

test(nested) :-
    fun_node_retract(td ),
    atom_chars("T((t)␣S)s", X), phrase(items(td, 10, 1, 30, [], []), X),
     relative_root (td, 11,12), relative_root (td, 12,13).

test(nested_changing) :-
    fun_node_retract(td ),
    atom_chars("T(t␣(:␣S))s", X), phrase(items(td, 10, 1, 30, [], []), X),
     relative_root (td, 11,13), relative_root (td, 12,11).

:- end_tests( items_relative ).
```

```

:- begin_tests(track_name).
test(simple) :- phrase(trackNameSpec("main"), ['"','m','a','i','n','"']).
test(empty) :- phrase(trackNameSpec(""), ['"','"']).
test(none) :- phrase(trackNameSpec([]), []).
:- end_tests(track_name).
```

```

:- begin_tests(fun_sounds).
test(simple):-  set_interval_defaults_none ,
                phrase(funSound(td,A), ['D','7','+']),
                phrase(funSounds(td,[A],[sound(euler(1,0),[euler (1,1)],  undef, undef, ['D'],
                                         [  [7,  ['+'], false,false, inherit ]  ],  0),
                                     sound(euler(-1,0),[],  undef, undef, ['s'],  [],   0)]),
                    ['.','&','s']).
test(simple2):-  set_interval_defaults_none ,
                 phrase(funSound(td,A), ['D','7','+']),
                 phrase(funSounds(td,[A,A],[sound(euler(1,0),[euler (1,1)],  undef,undef,['D'],
                                            [  [7,  ['+'], false,  false,  inherit ]  ],  0),
                                        sound(euler(-1,0),[],  undef, undef, ['s'],  [],   0)]),
                     ['.','&','s']).
test( virtual_ok1 ):-  set_style (fun_emotioFugax,false),
                    set_error_pos(td ,1,2,3),
                    phrase(funRootN(td, ['T','g'], euler(0,1)),['T','g']),
                    \+ fun_error(td,_,_,_,_,_).
test( virtual_ok2 ):-  set_style (fun_emotioFugax,true),
                    set_error_pos(td ,1,2,3),
                    phrase(funRootN(td, ['T','G'], euler(0,1)),['T','G']),
                    \+ fun_error(td,_,_,_,_,_).
test( virtual_error1 ):-  set_style (fun_emotioFugax,false),
                       fun_node_retract(td ),
                       set_error_pos(td ,1,2,3),
                       phrase(funRootN(td, ['T','G'], euler(0,1)),['T','G']),
                        findall ([A,B,C,D,E], fun_error(td,A,B,C,D,E), Errors),
                       Errors = [  [1,2,3,
                                "Superfluous␣mode␣change,␣e.g.␣use␣[Tg],␣not␣[TG]",
                                [['T','G']] ]].
test(tooLateToInherit_1):-
    phrase(funSound(td, A), ['D','7','+']),
    fun_node_retract(td ),
    set_error_pos(td ,1,2,3),
    phrase(funSounds(td, [A],_),   ['s','&','.']),
    fun_error(td ,1,2,3, "attempt␣to␣inherit␣interval␣with␣no␣chord␣preceding", []).
test(tooLateToInherit_2):-
    phrase(funSound(td, A), ['D','7','+']),
    fun_node_retract(td ),
    set_error_pos(td ,1,2,3),
    phrase(funSounds(td, [A],_),   ['s','&','4']),
    fun_error(td ,1,2,3, "attempt␣to␣inherit␣interval␣with␣no␣chord␣preceding", []).

:- end_tests(fun_sounds).
```

```

:- begin_tests(funSound_intervals_only).
test(inherit_whole_chord) :-
    set_interval_defaults_none ,
    phrase(funSounds(td,A), ['T', '&','d']),
```

```
    phrase(funSounds(td,A,[sound(euler(-1,0),[],undef,undef,['s'],[],0),
                           sound(euler (1,0),[], undef,undef,['d' ],[],0)]),  ['s','&','.']).

test( inherit_one_int ) :-
    set_interval_defaults_none ,
    phrase(funSound(td,A), ['D','7','+']),
    phrase(funSound(td,A,sound(euler(1,0),[euler(1,1)],undef,undef,['D'],
                                [ [7, ['+'], false,false, inherit ]  ],0)),
        ['.']).

test(inherit_too_much) :-
    set_error_pos(td ,1,2,3),
    fun_node_retract(td ),
    phrase(funSound(td,A), ['D','7','+']),
    phrase(funSound(td,A,_),  ['.', '.']),
    fun_error(td ,1,2,3, "attempt␣to␣inherit␣interval␣from␣undefined␣stack␣position",_).

test(inherit_too_much_2):-
    set_error_pos(td ,1,2,3),
    fun_node_retract(td ),
    phrase(funSound(td,_), ['D','7','+','.']),
    fun_error(td ,1,2,3, "attempt␣to␣inherit␣interval␣with␣no␣chord␣preceding", _).

:- end_tests(funSound_intervals_only).
```

```

:- begin_tests(fun_sound_1).
test( no_interval ) :- set_interval_defaults_none ,
                      phrase(funSound(td,sound(euler(1,0),[],undef,undef,['D' ],[],0)),  ['D']).
test( interval ) :- set_interval_defaults_none ,
                   phrase(funSound(td,sound(euler(1,0),[euler(1,1)],undef,undef,['D' ],[[7,[+], false,false ]],0)),
                         ['D', '7','+']).
test(bass) :- set_interval_defaults_none ,
             phrase(funSound(td,sound(euler(1,0),[euler(1,1)],
                                      [1,  euler (1,1)], undef,['D' ],[[7,[+], true,false ]],0)),
                   ['D', '7','+','_']).
test(more_tones) :- set_interval_defaults_none,
                   phrase(funSound(td,sound(euler(0,0),[euler(1,-1),euler (1,1)],
                                            [2,  euler (1,1)],  [1,euler(1,-1)],
                                            ['D','s'],  [[3,[], false,true],  [7,[+], true,false ]],0)),
                         ['D', 's','3','^','7','+','_']).
:- end_tests(fun_sound_1).
```

```

:- begin_tests(fun_sound_2).
% interaktiv :   phrase(funSound(A),['D ','7',+]),   phrase(funSound(A,R),['D ','.']).
test( inherit_one_int ) :-
    set_interval_defaults_none ,
    phrase(funSound(td,A), ['D','7','+']),
    phrase(funSound(td,A,sound(euler(1,0),[euler(1,1)],undef,undef,['D'],
                                [ [7,['+'],false,false, inherit ]  ],0)),  ['D', '.']).
test( inherit_two_int ) :-
    set_interval_defaults_none ,
    phrase(funSound(td,A), ['D','7','+', '3', '5']),
    phrase(funSound(td,A, sound(euler(1,0),[euler(1,1),  euler(-1,0),  euler (1,0)], undef,undef,['D'],
                                 [ [7,  ['+'], false,  false,  inherit ],
                                   [4,[], false,false ],
                                   [5,[], false,false, inherit ]  ],0)),
```

```
            ['D', '.', '4', '.']).
:- end_tests(fun_sound_2).
```

```

:- begin_tests( check_inherit_intervals ).
test(ident) :- check_inherit_intervals (Pitches,Pitches,['D','s','S'], ['D','s','S']).
test(ident_arrow) :- check_inherit_intervals (Pitches,Pitches,['D','s','↑'], ['D','s','↑']).
test(ident_mod_case) :- check_inherit_intervals (Pitches,Pitches,['D','s','S'], ['D','s','s']).
test(ident_mod_case_arrow) :- check_inherit_intervals (Pitches,Pitches,['D','s','↑'], ['D','s','↓']).
test( diff ) :- check_inherit_intervals (_,noIntervalInheritance,['D','s','S'], ['D','S','S']).
test( diff_length ) :- check_inherit_intervals (_,noIntervalInheritance,['D','s'], ['D','s','S']).
:- end_tests( check_inherit_intervals ).

:- begin_tests( inherit_intervals ).
test(empty) :- inherit_intervals (td ,[], [c, d], []).
test(no_inherit_from_none) :- inherit_intervals (td,[a, b], [], [a, b]).
test( no_inherit ) :- inherit_intervals (td,[a, b], [c, d], [a, b]).
test( inherit_one ) :- inherit_intervals (td,[ inherit ], [ [c1,c2,c3,c4] ], [ [c1,c2,c3,c4, inherit ] ]).
test( inherit_one_of_two ) :- inherit_intervals (td,[ inherit , b], [ [c1,c2,c3,c4], d], [ [c1,c2,c3,c4, inherit ], b]).
test( inherit_two ) :- inherit_intervals (td,[ inherit , inherit ],
                                         [ [c2,c3,c3,c4], [d1,d2,d3,d4] ],
                                         [ [c2,c3,c3,c4, inherit ], [d1,d2,d3,d4, inherit ] ] ).
test(much_input) :- inherit_intervals (td,[a], [c, d, e], [a]).
:- end_tests( inherit_intervals ).
```

```

:- begin_tests(root_and_mode).
% vorbild sind interaktive aufrufe wie phrase(root_and_mode(R),['D', p, 'D'])
test(simple) :- phrase(root_and_mode(['D','p','G']), ['D','p','G']).
test(not_root_1) :- \+ phrase(root_and_mode(_), ['p','G']).
test(not_root_2) :- \+ phrase(root_and_mode(_), ['D','G','e']).
:- end_tests(root_and_mode).
```

```

:- begin_tests( intervals_greater_10 ).

test(one_eleven) :-
    no_intervals_larger_10 (td),
    allow_interval_11 (td),
    phrase(intervals(td,[ [1,[], false,false], [11,['+'],false,false] ]), ['1',',','1','1','+']).

test(two_eleven) :-
    no_intervals_larger_10 (td),
    allow_interval_11 (td),
    phrase(intervals(td,[ [2,[], false,false], [11,['+'],false,false] ]), ['2','1','1','+']).

test(eleven_one) :-
    no_intervals_larger_10 (td),
    allow_interval_11 (td),
    phrase(intervals(td,[ [11,[], false,false ],[1,[ '+','+'],false,false] ]),[ '1','1','1','+','+']).

test(simple_no_11) :-
    no_intervals_larger_10 (td),
    \+ phrase(interval(td ,[11,[], false,false ]), ['1','1']).

test(simple_no_11_double) :-
    allow_interval_11 (td),
    no_intervals_larger_10 (td),
```

```
    \+ phrase(interval(td ,[11,[], false,false ]), ['1','1']).

test(simple_11) :-
    no_intervals_larger_10 (td),
    allow_interval_11 (td),
    phrase(interval(td ,[11,[], false,false ]), ['1','1']).

:- end_tests(intervals_greater_10 ).
```

```

:- begin_tests( intervals ).
test(simple) :- phrase(intervals(td ,[[3, suppress,false,false ],[6,[ '+','+'],false,false]]),
                       ['3','/','6','+','+']).
test( fails ) :- \+ phrase(intervals(td,_), ['3','/','6','+','-']).
test(comma) :- phrase(intervals(td,[ [1,[], false,false], [2,[], false,false] ]), ['1',',','2']).
test(one_one) :- phrase(intervals(td ,[ [1,[], false,false], [1,[ '+'],false,false] ]),
                        ['1',',','1','+']).
test(one_nine) :- phrase(intervals(td ,[ [1,[], false,false], [9,[], false,false] ]), ['1','9']).
test( fail_modifier ) :- \+ phrase(intervals(td,_), ['1','x']).
test( inherit ) :- phrase(intervals(td ,[ [1,[], false,false], inherit , [2,[], false,false] ]), ['1','.','2']).
test(no_2_comma) :- \+ phrase(intervals(td,_), ['1', ',', ',', '2']).
test(no_3_mel) :- \+ phrase(intervals(td,_), ['2', '_', '~', '^', '3']).
test(bass_mel) :- phrase(intervals(td ,[[3,[ '-'],true,true], [4,[], false,true] ]),
                         ['3', '-', '_', '~','4', '^']).
:- end_tests( intervals ).
```

```

:- begin_tests(extract_base_and_mel).
test(bass_1) :-
    extract_base_and_mel(td, [[ _,_,false,false], [_,_,false,false], [_,_,true,false ],[ _,_,false,false ]],
    0, undef, undef, 2, undef).
test(bass_2) :-
    extract_base_and_mel(td, [[ _,_,false,false], [_,_,false,false], [_,_,true,false ],[ _,_,true,false ]],
    0, undef, undef, _, _).
test(mel_and_bass) :-
    extract_base_and_mel(td, [[ _,_,false,false], [_,_,false,true], [_,_,true,false ],[ _,_,false,false ]],
    0, undef, undef, 2,1).
test(bass_and_mel) :-
    extract_base_and_mel(td, [[ _,_,true,false], [_,_,false,false], [_,_,false,true ],[ _,_,false,false ]],
    0, undef, undef, 0,2).
test(bass_and_mel_error1) :-
    \+ parse_string(td,"D3~79^"),
    fun_error(td,1, _, _, "double␣melody␣pitch␣selection", [1,3]).
:- end_tests(extract_base_and_mel).
```

```

:- begin_tests( interval_pitch ).
test(minor) :- interval_pitch ( [3,[], _], euler(1,-1), minor, _).
test(major) :- interval_pitch ( [3,[], _], euler(0, 1), major, _).
test(d7) :- interval_pitch ( [7,[], _], euler(-2, 0), major, is_dominant).
test(multi) :- interval_pitches ( [ [3,[], _], [6, ['+'], _] ], [euler(0, 1), euler(-1, 1)], major, _).
:- end_tests( interval_pitch ).
```

```

:- begin_tests( default_intervals ).
%          result                     akku        defaults           explicit     suppRules
test(simple) :-
```

```
    add_defaults([ [5,[]| _], [3,[]| _], [1,[]] ], [ [1,[]] ], [ [3,[]], [5, []] ], [ [1,[]] ], [] ).
test(suppress) :-
    add_defaults([ [5,[]| _] ], [], [ [5,[]], [3,[]] ], [ [2,['+'], gr, foo] ], [ [2, ['+'], 3, []] ]).
test(suppress_by_overriding) :-
    add_defaults( [], [], [ [5,[]], [3,[]] ], [ [5,[], gr, foo], [3, ['+'], fr, groo] ], []).

% result, akku, ondeDefault, explicit, suppress_rules, suppress_rules-backup
test(one_suppress_empty) :-
    add_one_default ([],[], [3,[]], [ [2, ['+'], gr, foo] ], [ [2, ['+'], 3, []] ], grmpf).
test(one_suppress_not) :-
    add_one_default([ [3,['-'],_,_] ], [], [3,['-']], [ [2, ['+'], gr, foo] ], [ [2, ['+'], 3, []] ], grmpf).
test(one_suppress_modif) :-
    add_one_default ([], [], [3,['-']], [ [2, ['+'], gr, foo] ], [ [2, ['+'], 3, ['-']] ], grmpf).
test(one_suppress_any) :-
     add_one_default ([], [], [3,['-']], [ [2, ['+'], gr, foo] ], [ [2, any, 3, ['-']] ], grmpf).
test( one_suppress_with_prefix_explicit ) :-
    Rules = [ [2, any, 3, ['-']] ],
    add_one_default ([], [], [3,['-']], [ [5, [], gr, foo], [2, ['+'], gr, foo] ], Rules, Rules).
test(one_suppress_with_prefix_rules) :-
    add_one_default ([], [], [3,['-']], [ [2, ['+'], gr, foo] ], [ [7, [], 8, []], [2, any, 3, ['-']] ], grmpf).
test(one_suppress_with_prefices) :-
    Rules = [ [7, [], 8, []], [2, any, 3, ['-']] ],
    add_one_default ([], [], [3,['-']], [ [5, [], gr, foo], [2, ['+'], gr, foo] ], Rules, Rules).

:- end_tests( default_intervals ).

:- begin_tests( default_intervals_in_context ).
test(suppress_all) :- set_interval_defaults_conventional ,
                        phrase(funSound(td,sound(euler(1,0), [], undef, undef, ['D'], [], 2)),
                              ['D', '/','/']).

test(suppress_3) :- set_interval_defaults_conventional ,
                       phrase(funSound(td,sound(euler(1,0), Ints, undef, undef, ['D'],
                                     [ [3, suppress, false, false] ], 0)), ['D', '3', '/']),
                       sort(Ints, [ euler (0,0), euler(1,0) ]).

test(suppress_1) :- set_interval_defaults_conventional ,
                       phrase(funSound(td,sound(euler(1,0), Ints, undef, undef, ['D'], [], 1)), ['D', '/']),
                       sort(Ints, [euler (0,1), euler (1,0)]).
%    member(euler(1,0), Ints),    member(euler(0,1), Ints),      length(Ints, 2).

test(suppress_by_4) :- set_interval_defaults_conventional ,
                         phrase(funSound(td,sound(euler(-1,0), Ints, undef, undef, ['s'], [ [4,[], false,false] ],
                                            0)), ['s', '4']),
                         sort(Ints,Sorted), Sorted = [euler(-1,0), euler (0,0), euler (1,0)].

test(suppress_by_4_not) :- set_interval_defaults_conventional ,
                             phrase(funSound(td,sound(euler(-1,0), Ints, undef, undef, ['s'],
                                              [ [4,['+'],false,false] ],
                                              0)), ['s', '4', '+']),
                             sort(Ints,Sorted), Sorted = [euler (0,0), euler(1,-1), euler (1,0), euler (2,1)].
:- end_tests( default_intervals_in_context ).
```

```

:- begin_tests(normalize_functions).
test(idem) :- normalize(td,['T','p'],['T','p']).
test(up_expl) :- normalize(td,['T','P'],['T','p','↑']).
test(up_down_expl) :- normalize(td,['T','P','g','g'],['T','p','↑','g','G', '↓' ]).
```

```

test(fugax_pass) :-
    fun_node_retract(td),
    set_style(fun_emotioFugax,true),
    set_error_pos(td,11,21,31),
    normalize(td, ['T','P','D'],['T','p','↑','D']),
    \+ fun_error(td, _,_,_,_,_).

test( fugax_fail ) :-
    fun_node_retract(td),
    set_style(fun_emotioFugax,false),
    set_error_pos(td,11,21,31),
    normalize(td, ['T','P','D'],['T','p','↑','D']),
    fun_error(td, 11,21,31, "Superfluous␣mode␣change,␣e.g.␣use␣TgD,␣not␣TGD", _).

%% NOETIG to fail ?? should be excluded by grammar!
test(no_funcode) :- \+ normalize(td,[e], _).
test(no_funcodes) :- \+ normalize(td,[e, 'P'],_).
test(no_funcodes2) :- \+ normalize(td,['P', e],_).
:- end_tests(normalize_functions).
```

```

:- begin_tests(mdom).
test(major) :- extract_mdom(['D','t','D'], major, is_dominant).
test(minor) :- extract_mdom(['D','t'], minor,[]).
test(min_arrow) :- extract_mdom(['X','Y', '↓'], minor,[]).
:- end_tests(mdom).
```

```

:- begin_tests( retrieval ).
test(simple) :-
    set_interval_defaults_none ,
    parse_string(td,"c:D␣DD"),
% ASSUME node 1 is top-level track node
    all_results (td, 2, euler (1,0),[], undef,undef),
    all_results (td, 3, euler (2,0),[], undef,undef).

test( simple_relative ) :-
    set_interval_defaults_none ,
    parse_string(td,"g:(D)␣DD"),
    all_results (td, 2, euler (4,0),[], undef,undef),
    all_results (td, 3, euler (3,0),[], undef,undef).

test( multi_relative ) :-
    set_interval_defaults_none ,
    parse_string(td,"eb:D␣(:(D)␣DD)"),
    all_results (td, 2, euler (-2,0),[], undef,undef),
    all_results (td, 3, euler (1,0),[], undef,undef),
    all_results (td, 4, euler (0,0),[], undef,undef).

test(simple_subtrack) :-
    set_interval_defaults_none ,
    parse_string(td,"c:D␣{f#:D}␣T"),
    all_results (td, 2, euler (1,0),[], undef,undef),
% 3 is sub-track
    all_results (td, 4, euler (7,0),[], undef,undef),
    all_results (td, 5, euler (0,0),[], undef,undef).

```

```
test(multi_subtrack) :-
      set_interval_defaults_none ,
     parse_string(td,"f#:D␣{␣D␣{f:T}␣ss}␣"),
      all_results (td, 2, euler (7,0),[], undef,undef),
% 3 is sub-track
      all_results (td, 4, euler (7,0),[], undef,undef),
% 5 is sub-track
      all_results (td, 6, euler (-1,0),[], undef,undef),
      all_results (td, 7, euler (4,0),[], undef,undef).

test( virtual_relative ) :-
      set_interval_defaults_none ,
     parse_string(td,"g:␣(D)␣[Sp]"),
      all_results (td, 2, euler (0,1),[], undef,undef).

test( virtual_relative_back ) :-
      set_interval_defaults_none ,
     parse_string(td,"g:␣[Sp]␣(:D)␣"),
      all_results (td, 3, euler (0,1),[], undef,undef).

test(error_reference_sum) :-
      set_interval_defaults_none ,
     parse_string(td,"g:␣(D)␣Sp&s"),
     \+ all_results (td, 2, _,_,_,_),
     fun_error(td, 2,1,1, "cannot␣refer␣to␣a␣sum␣of␣more␣than␣one␣functions",[3]).

test(error_reference_idem,blocked(doesntPARSE)) :- %% FIXME CHECK ***
      set_interval_defaults_none ,
     parse_string(td,"g:␣(D)␣-␣"),
     \+ all_results (td, 2, euler (0,1),[], undef,undef),
     fun_error(td, 2,1,1, "cannot␣resolve␣this␣node␣as␣a␣reference␣point",[2]).

test(error_tonic_undefined) :-
     parse_string(td,"␣D␣{T}␣␣"),
     \+ all_results (td, 4, _,_,_,_),
     fun_error(td, 4,3,2, "top␣track␣tonic␣centre␣is␣undefined", []).

:- end_tests( retrieval ).
```

```

:- begin_tests(retrieval_sum).
test(error_more_melody) :-
      set_interval_defaults_none ,
      set_style (fun_extremaldempotentes, false),
     parse_string(td,"g:␣D1^&T5^"),
     \+ all_results (td, 2, _,_,_,_),
     fun_error(td, 2,1,1, "more␣than␣one␣melody␣indication␣in␣sum␣expression", _).

test(no_error_more_melody) :-
      set_interval_defaults_none ,
      set_style (fun_extremaldempotentes, true),
     parse_string(td,"g:␣D1^&T5^"),
      all_results (td, 2, euler (2,0),[ euler (2,0)], undef,euler (2,0)).

test(error_more_bass) :-
      set_interval_defaults_none ,
      set_style (fun_extremaldempotentes, true),
     parse_string(td,"g:␣D1_&T1_"),
```

```
    \+ all_results (td, 2, _,_,_,_),
    fun_error(td, 2,1,1, "more␣than␣one␣bass␣indication␣in␣sum␣expression", _).

:- end_tests(retrieval_sum ).
```

```

:- begin_tests(root_and_mode_fancy).
test (fancy1) :-  select_locale_for_function ('FX'),
                  parse_string(td, "C:udl␣OdR"),
                  fun_node(td,2,1,1,sum([sound(euler(-1,-1), [], undef, undef, ['s','g'], [],  0)])),
                  fun_node(td,3,1,2,sum([sound(euler( 0,1),  [],  undef, undef, ['D','P'], [],  0)])),
                  % is \xt{sg}.and \xt{OdR} is \xt{DP}.
                   select_locale_for_function ('DE').
:- end_tests(root_and_mode_fancy).
```

```

:- begin_tests(function_macros).
test (macro_dv) :-
     select_locale_for_function ('DE2'),
     set_interval_defaults_conventional ,
    parse_string(td,"c:DDV7_"),
     all_results (td,2,  euler (2,0),  Ps, euler (0,0), undef),
    sort(Ps, [euler (0,0), euler(1,-1),euler (2,1), euler  (3,0)]),
     select_locale_for_function ('DE').

test (macro_sn) :-
     select_locale_for_function ('DE2'),
     set_interval_defaults_conventional ,
    parse_string(td,"c:TgsN2-␣.1"),
     all_results  (td,2,  euler(-1,0),  Ps,  euler(-1,1),undef),
    sort(Ps,  [euler(-2,-1),euler(-1,1),euler (0,0)]),
     all_results  (td,3,  euler(-1,0),  Pt,  euler(-1,1),undef),
    sort(Pt,  [euler(-1,0),euler(-1,1),euler (0,0)]),
     select_locale_for_function ('DE').

test (macro_dhv) :-
     select_locale_for_function ('DE2'),
     set_interval_defaults_conventional ,
    parse_string(td,"c:Dhv9-"),
     all_results  (td,2,  euler (1,0),  Ps, undef,undef),
    sort(Ps, [euler(-1,-1),euler(-1,0),euler(0,-1),euler (1,0), euler  (1,1)]),
     select_locale_for_function ('DE').

:- end_tests(function_macros).
```

```

:- begin_tests(layout ).
test (sort_simple)  :-
    parse_string(td,"c:D␣{f#:T}{g#:T}␣S"),
    sorted_sub_tracks(td  ,1,[5,3]).

test (sort_crossed) :-
    parse_string(td,"c:D␣{f#:T}␣<{g#:T}␣S"),
    sorted_sub_tracks(td  ,1,[5,3]).

test (layout_simple)   :-
    % score           1   1 2 2   2   1 1 2
    % node            1   2 3 4   5   6 7 8
```

```
    parse_string(td,"c:␣D␣{␣d}␣D␣<{␣T␣T}␣"),
    %                    D D
    %                      d
    %                    T T
    calculate_layout(td,2),
    track_row(td,1,1),  track_row(td,3,2),  track_row(td,6,3).

test(layout_TMP) :-
    parse_string(td,"c:D␣{f#:T}␣{g#:T}␣S"),
    calculate_layout(td,1),
    track_row(td,1,1),  track_row(td,3,2),  track_row(td,5,3).

test(layout_crossed) :-
    parse_string(td,"c:D␣{f#:T}␣<{g#:T}␣S"),
    calculate_layout(td,1),
    track_row(td,1,1),  track_row(td,3,2),  track_row(td,5,3).

test(layout_crossed_long) :-
    % score            1  2 3 4 5 6   2 3  4  5  6     7  8  9  7  8      7  7
    % node           1 2  3 4 5 6 7  89 10 11 12 13 14 15 16 17 18 19 20 21 22
    parse_string(td,"c:D␣>D␣D␣D␣D␣D␣<{T␣T␣␣T␣␣T␣␣T␣␣{␣␣S␣␣S␣␣S}␣T␣␣T␣}␣{␣␣d}␣D"),
    %                  D  D D D D D  D
    %                                d
    %                     T T T T T  T T
    %                                S S S
    calculate_layout(td,1),
    track_row(td,1,1),  track_row(td,8,3),  track_row(td,14,4),  track_row(td,20,2).

test(layout_crossed_long2) :-
    % score          1 1  2 3 4 2  2 3  4  5 6  6   6
    % node           1 2  3 4 5 6  7 8  9 10 11 12  13
    parse_string(td,"c:D␣>D␣D␣D␣<{␣T␣T␣␣T}␣D␣{␣␣d}␣␣D␣"),
    %                  D  D D D D D
    %                           d
    %                     T T T
    calculate_layout(td,2),
    track_row(td,1,1),  track_row(td,6,3),  track_row(td,11,2).


test(layout_crossed_stabile) :-
    % score          1  1 2 2   2  2 2
    % node           1  2 3 4   5  6 7
    parse_string(td,"c:␣D␣{␣d}␣>D␣<{␣T}␣"),
    %                    D   D
    %                        d
    %                        T
    calculate_layout(td,2),
    track_row(td,1,1),  track_row(td,3,2),  track_row(td,6,3).
:- end_tests(layout).
```

```

:- begin_tests(latex).
test(common_interval_1):-
    common_interval(1,4,4,10).
test(common_interval_2):-
    common_interval(10,11,4,10).
test(common_interval_3):-
    \+ common_interval(1,2,4,10).
```

```
test(common_interval_4):-
    \+ common_interval(11,12,4,10).

test(sounds) :-
    genLT_sounds([sound(_,_,["D"],_,_,_,_), sound(_,_,["T"],_,_,_,_ )],[], _R).

test(simple) :-
    parse_string(td,"c:D␣{f#:T}{g#:T}␣S"),
    generate_latex(td ,1,[3,6,9], _Result).


test(file_complex) :-
     generate_latex_files (td,"c:D35-_␣S␣{t&s␣d}{d␣t}␣T//␣D␣",
                               2,[3,6,9], "LTXTEST-complex").

test( file_collision ) :-
     set_style (fun_praesentatio_bassusUtSpatium,false),
    %% fixed GEHT NICHT " S S" wird NICHT gedruckt ???
    %%     generate_latex_files (td,"c:D T {t d}{d t S S} T D ",
    %% OK
                 generate_latex_files (td,"c:D35-_␣S␣{t7-_35&s6+_2+4␣d}{D5-_79-␣5-.8}␣T//␣D␣",
    %% OK   generate_latex_files (td,"c:D35-_ S {t7-_35&s6+_2+4_d}{D5-_79- D5-.8} T// D ",
    %% OK        generate_latex_files (td,"c:D35-_ D.",
                               2,[3,7,11], "LTXTEST-collision").


:- end_tests(latex).
```

```

:- begin_tests( translate_lpr ).
test(xDD) :-
    atom_chars("DD",X), atom_chars("LRLR",Y), translate_LPR(X, Y).
test(s) :-
    atom_chars("s",X), atom_chars("RLP",Y), translate_LPR(X, Y).
test(xTG) :-
    atom_chars("TG",X), atom_chars("LP",Y), translate_LPR(X, Y).
test(xTGP) :-
    atom_chars("TGP",X), atom_chars("LPRP",Y), translate_LPR(X, Y).
test(tg) :-
    atom_chars("tg",X), atom_chars("PLP",Y), translate_LPR(X, Y).
test(tgp) :-
    atom_chars("tgp",X), atom_chars("PLPRP",Y), translate_LPR(X, Y).

:- end_tests( translate_lpr ).
```