University of Music Karlsruhe
Institute for Music Informatics and Musicology

Symbolic Music Typesetting
(The Engine)

Amir TEYMURI

December 7, 2020

Abstract

A new program for generating musical scores called *Symbolic Music Typesetting* has been devised by the author. This program has been evolved primarily from a personal dissatisfaction of the author in his working with different existing computer score-writers, both in terms of results as well as of the program's user-interface design. In this writing some of the principal aspects of SMT's engine, which forms it's graphical rendering kernel, will be depicted accompanied by many examples, in the hope, that a basic understanding of it's potentiality and philosophy can be conveyed.

Contents

CHAPTER 1

Introduction

1.1 Preliminaries

1.1.1 Music Engraving

Until the early 1990s, most of the large music publishing houses were involving practiced and experienced master craftsmen who did all the engraving work merely by hand. Although the developement of computer systems for writing *low-budget* musical scores in the course of the last 40 years has completely outpaced this unique art and craft and made it obsolete by virtue of their comparatively low costs and higher speed of manufacturing printed music, the beauty and the quality of the hand-engraved music scores of old masters still remains unrivaled. Historically, before computers were in widespread use in music typesetting, the main practical methods for the production of musical scores could roughly be divided into the following categories: music engraving, auto-typography, by using transfers, by using note-typewriters and by using stencils. As perhaps the most widespread music print technique and the oldest craftsmanship of it's kind, music engraving met fairly small changes over centuries only in it's fabrication of steel-stamps and precision-instruments for the engraver. Although other experiments in the field of music typesetting could find their place in the industry from time to time[1], music engraving could assert itself as the main procedure in the production of high-quality musical print.

1.1.2 The Workflow of the Master Engraver

As perhaps the most influential of the music printing techniques, music engraving was around for about 200 years until the late 20th century. The production of a single engraving plate for a music of moderate complexity level, would often take a whole working day, and basically took place in two stages: hammering of types (note-heads, stem-flags, accidentals, rests, text etc.) into the plate and subsequently *engraving* all the remaining parts such as stems, ledger lines, legato bow, beams etc.. But before the master engraver could hammer the first symbol into his plate, a comprehensive work should be done, to divide the manuscript into pages and lines in an optimal way, suited best to the content and the character of the music being engraved. In the course of dividing the manuscript, the number of bars on each line[2], and the number of lines to be fitted into each page should be investigated, whereas the convenient page-turning spots and the overall volume of the

1 For instance Johann Gottlob Immanuel Breitkopf's invention of dissasembling note-heads, stems, staff-lines, flags and other parts of music-notation into many different pieces, to mention only a prominent example out of many.

2 Speaking here of rather traditional western music which almost always uses bars as it's *unit* of division.

edition had to be taken into consideration. Hence, it was important for the success of an edition, to settle the content of each page beforehand. The next stage, the division of heights, consisted of a precise calculation of the gaps between the musical systems. Although these spacings oftentimes would be quite varying, it was the artistry of the master engraver, to simulate a visual consistency and to evoke a visual harmonic illusion by means of optical adjustments. Only after this time-consuming preperations, could the five staff-lines be engraved into the plate via the so-called rastrum.

1.1.3 Computer Music Typesetting Systems

The profession of music engraving was learned and handed on to next generations only by doing and practicing it. It's beauty and elegance resulted solely from the judgement of the engraver, his craftsmanship and his experience. Little to no sources does exist which would describe the finest case-related and the overall rules of engraving. These of course make implementing a computer program, which could compete with the work of a master engraver a non-trivial challenge! The computer programs which have been developed for music notation fall into one of the two categories: 1- those with graphical user-interfaces and 2- those who expect input from users in a textual form. These two paradigms exhibit some substantial differences in the way they allow expressing music. Although the first paradigm (also known as *What You See Is What You Get*; WYSIWYG) presents a more intuitive interface, it offers less flexibility in terms of changing the behavior of the system if needed or desirable. Organizing and managing data for bigger projects is basically not left to user's own discretion. Also non-physical storing of the resources (e.g. on the cloud) which is a vital aspect in many fields of musical research, musicology etc. when not happening in another text-based form (for instance in markup langauges and alike) is mostly cumbersome. The other paradigm (which I call *What You Read Is What You Get* or WYRIWYG, to keep up with the style of word composition of the previous paradigm) could give more control (possibly down to the kernel of the system) for adjusting or changing the behavior of the system. With WYRIWYG comes an important aspect with which we are not confronted when only clicking on the mouse- or a midi-instrument-key for inputing music (WYSIWYG). As the WYRIWYG paradigm bears some resemblence to a programming environment (or a markup langauge[1]), it is inevitable for it's design to take the expressiveness of the language into account. As (text-based) computer systems (such as WYRIWYG music notation software) grow in complexity, the importance of the level of *expressiveness* the system offers, becomes more and more relevant. Also when it comes to particular notation fields (e.g. percussion-notation and alike), almost all notation software treat this sector as a special case which involves introducing new syntactic structures for them. While legitimate to treat some *already known* special cases like percussion-notation, ancient mensural-notation, world music, extended-techniques in contemporary music etc., there could potentially be an unlimited range of individual requests and needs, where the users might want to construct and introduce their own notational concepts and systematics.

1 The music notation software LilyPond for instance, having an integrated scheme interpreter as interface to it's parser/compiler toolchain, is a markup language with support for doing some programming tasks in the scheme programming language.

The next lines of this report shall give a preliminary understanding of some of the design aspects and the philosophy of SMT.

1.2 Why Another Score-Writer?

SMT is a Common Lisp system for writing musical scores. Innovative ideas and complexity of music notation in the works of composers of the last and current centuries have posed a great challenge to digital notation systems in both technical as well as philosophical terms. Different composers have conceived and implemented different notation systems for their own work, partly quite distant from the established and conventional concepts we know from the Common Western Music Notation; from graphical and spatial notations of the members of the New York School or artfully rendered scores of George Crumb, attempts for transferring electronic music to paper by Stockhausen, Xenakis and Ligeti to even music not conceived for publication in form of printed material e.g. studies of Conlon Nancarrow for mechanical piano or even improvisational music to only name a few. Although in most cases composers, researchers and publishers snatch at the paradigmes of the Common Western Music Notation, through it's modular and rule-based design, SMT promotes re-thinking of notational concepts as well as conceiving new alternatives which might suit better to certian types of music. The challenge posed to notation systems basically boil down to the question whether different forms and types of music notation (including those unknown to the authors of the systems!) are expressible in the system, hence is the system extensible *enough* and whether it allows for defining and accommodating new notations just as the established conventions of western music notation. Many music typesetting systems provide their users with some possibilities for extending the system's behavior (e.g. the ManuScript language for writing plug-ins in Sibelius or the Guile Scheme for further interaction with the LilyPond's parser-compiler toolchain), oftentimes these extensions however can't go beyond what these systems *already* define and understand as music notation (namely classical western music notation), which leads to the modifications the user has to undertake in his/her notational concepts for them to fit within the scope of the system's definition of music! SMT is also an attempt for compensating some shrotcomings in this respect.

As SMT is still under active development, this report strives for giving an overview of some of it's design aspects accompanied by some examples. Since this report describes a work-in-progess, the author acknowledges also that some of the aspects of the engine addressed in this report are subject to change in the future.

1.3 Current State of Development

The work on the SMT-engine and it's central theme hitherto has been focused on achieving a music notation system which is not only modifiable and extendible, but also definable. Hence a relatively big part of the work until now has been dedicated to the developement of it's rule definition protocol, which should make that goal possible. Although the rule protocol is not thoroughly implemented yet and requires many refactorings, it is on it's way to maturity so that it's basic concepts can be introduced and presented through examples in this writing. SMT is written in Common Lisp and relies heavily on it's Object System (CLOS). Although it might be subject to change in future work, the main way of modifying

SMT's behavior at this moment is through direct communication and changing objects and their slot values. Hence writing effective rules for SMT objects almost *always* involves re-setting values of some of their slots and is thus a destructive act! Until now the following classes have been implemented into the engine: the base abstract class SMTOBJ from which every other class inherits, the abstract class CHASE which represents a frame (or a canvas, or a surface) which embeds a graphical unit (a musical symbol or a COMPOSING-STICK), the abstrace class MTYPE which inherits from the CHASE and represents a musical symbol (a glyph) and some of it's subclasses such as NOTE, NOTEHEAD, ACCIDENTAL and REST, and the abstract class COMPOSING-STICK which plays the role of a container for other objects, including instances of other COMPOSING-STICKS. The three variants of the COMPOSING-STICK abstract class which are in the public interface of the engine are called STACKED-COMPOSING-STICK, HORIZONTAL-COMPOSING-STICK and VERTICAL-COMPOSING-STICK. The terminology and many of the concepts used here in the design of the engine has been *inspired* by the *Letter-press Print* (though in a bit modified form). These objects and their connections to the tools with the same name in the letter-press printing are handled in more depth in the next chapter. SMT's native output format is the ubiquitous Scalable Vector Graphics (SVG). Many other conversions of course (e.g. to PDF) or connecting the output with other XML family members (e.g. embedding into HTML etc.) can easily take place from the generated SVG files.

The following chapters provide a more technical insight into the engine.

CHAPTER 2

SMT Object System

At the top of SMT's object system sits the class SMTOBJ. This base abstract class guarantees that each object used in our score receives a unique id through which the engine can keep track of all it's attributes and actions and also watch the created objects in a hash-table called `*central-registry*`[1]. Every SMTOBJ also has a list of *ancestors* which is updated each time we add the object to the *content* list of other objects[2]. Ancestors affect their *descendant*s (most importantly) by means of movement and scaling; they cause all their descendants to be displaced on the page by the same amount of vertical or horizontal movement, and they cause all their descendants to be re-scaled by the same scaling factor.

2.1 Customizing Rendering of a SMTOBJ Using SVG

Each SMTOBJ possesses a list which contains all of the SVG elements associated with an object and which will be rendered alongside the object. This list can be accessed via the *accessor function* `svglst` of the object. Hence we can add our own SVG elements to this list. As mentioned in the introduction, SMT comes with an integrated SVG system, which we can use for using SVG elements alongside the ones generated by the engine[3]. There is a convenience function for pushing SVG elements to the `svglst` of an object called `packsvg!` which has the following syntax:

packsvg! *object* `&rest` *svg-elements*

and pushes the `svg-elements` to the `object`'s `svglst`. Before I demonstrate a simple example of integrating SVG elements into a resulting image generated by the engine, I would like to show a basic form of generating a single note, which we then decorate with our own SVG elements.
A note object is an instance of the class `note` which is a subclass of the class `stacked-composing-stick` (see the section 2.4 for more on composing sticks). As a subclass of the composing stick class, a note can *contain* other elements (e.g. note-head, stem, flag etc.) which are stored in the `content` list of a note object. Beside the attributes inherited from the composing stick class, a note object has a `:spn` slot which designates the pitch of the note as a dotted list of two elements: `(pitch-name . pitch-octave)` (see the

1 Having unique ids is important in (amongst other things) defining rules. More on this topic later!
2 Only a composing-stick has a `content` list and can hence contain other objects.
3 I will not discuss the SVG layer in this writing. For more on that, please refer to the Github repository of SMT's XML utility under `https://github.com/SymbolicMusicTypesetting/xmlutils.git`

section 3.2.2 for more on SPN). A note has also a duration (specified with the `:dur` slot) with a default value of 1/4, where 1/4 stands for a quarter note, 1/2 for a half note, 1 for a whole note and so forth. For each of the elements of a note which are contained in it's `content` list, there is an additional corresponding slot which allows us to create those objects explicitely (e.g. for creating note-heads or flags or stems). For instance, although the note-head of a note object is created implicitely by the engine based on the duration we specify for the note, it is possible to create a different note-head object by initializing the slot `:head` to an instance of the class `notehead`. In the following example the note and the note-head objects are created by calls to the functions `note` and `notehead` respectively. The first argument to the `note` function is the SPN dotted list, and the first argument to the `notehead` function is a string designating the label of the glyph to be used for the note-head instance (for more on note-heads please see the section 3.2.3):

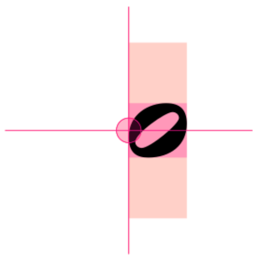
```
CL-USER> (in-package #:smtngn)
#<PACKAGE "SMTNGN">

SMTNGN> (id (head (note '(c . 4) :head (notehead "s0" :id 'my-head))))
MY-HEAD
```

Note that for now other elements like the stems and the flags of the note are not implemented yet, so that they are not going to appear in the following examples.

Let us now go back to experimenting with adding some SVG elements to the image of a note:

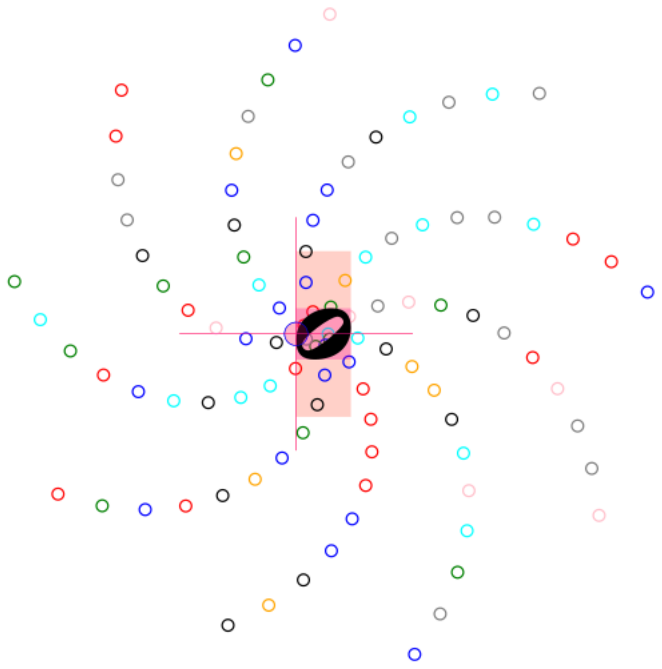
```
(render (list (note '(f . 4) :duration 1/2 :toplevelp t)))
```

The code above will generate the following image:



Let us now add some circles to the image of the note-head:

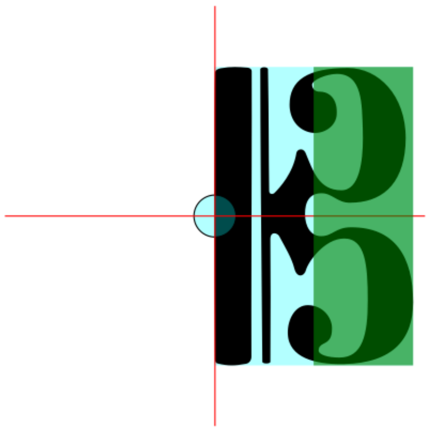
```
(defun spiral (a b step-resolution step-count)
  "Returns a list of coordinates for r=a+b*theta stepping theta by
step-resolution."
  (loop for theta
        from 0 upto (* step-count step-resolution)
        by step-resolution
        for r = (+ a (* b theta))
        for x = (* r (cos theta))
        for y = (* r (sin theta))
        collect (cons x y)))

;; Add SVG circles to the SVGLST of the note:
(let ((f4 (note '(f . 4)
                :duration 1/2
                :toplevelp t))
      (colors '("red" "orange" "pink" "green"
                "blue" "black" "gray" "cyan")))
  (apply #'packsvg! f4
    (loop for pnt in (spiral .8 .8 .7112 100)
          collect (svg:circle
                       ;; Circel X
                       (+ (car pnt) (left f4) (/ (width f4) 2))
                       ;; Circle Y
                       (+ (cdr pnt) (top f4) (/ (height f4) 2))
                       ;; Circle Radius
                       1
                       ;; Circel Color
                       :stroke (nth (random (list-length colors)) colors)
                       ;; Circel Line Thickness
                       :stroke-width 10
                       :fill "none")))
  (render (list f4)))
```



As another example of customizing the output of the engine using it's SVG layer, let us fill the right half of the frame of a mtype object (a clef) with a green rectangle (more information about mtypes can be found in the section 2.3). For this, we again use the `packsvg!` function which we described earlier[1]:

```
(let ((c (clef '(c . 4) :toplevelp t)))
  (packsvg! c (svg:rect
                ;; Rectangle's x
                (+ (left c) (/ (width c) 2))
                ;; Rectangle's y
                (top c)
                ;; Rectangle's Width
                (/ (width c) 2)
                ;; Rectangle's Height
                (height c)
                ;; Rectangle's Color
                :fill "green" :fill-opacity .6))
  (render (list c)))
```

which yields:



While the actions above are not explained in full depth, it should be obvious from the examples that instructing the engine to render more complex SVG drawings is easily viable. For the sake of brevity, I content myself with these small demonstrations on interacting with the SVG-layer of the engine and refer to the online documentations of the project[2] for more information.

Two other important slots of each SMTOBJ which will be handled in more depth in the chapter about the rule protocol are `domain` and `ruleids`. The `domain` of an object tells

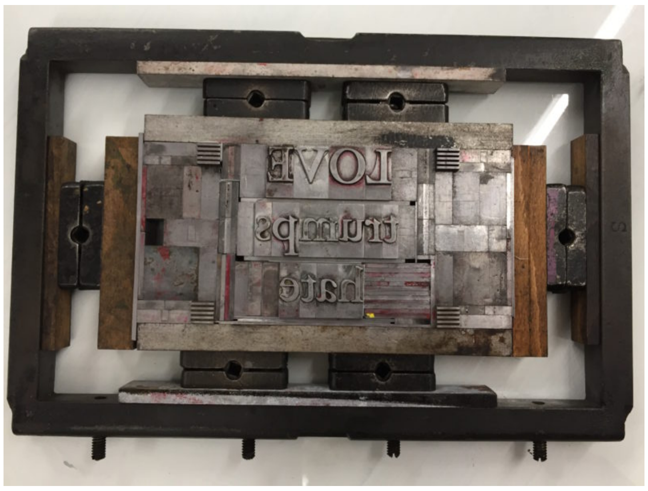
1 Note again that a call to `packsvg!` is destructive, since it changes the state of object's `svglst`.
2 https://symbolicmusictypesetting.github.io/smtdoc/

the engine to look in the defined set of rules and sort out all rules which have been defined for that domain[1]. Before the engine however goes on with applying rules to the object, it iterates over the list of rule-ids which are names of slots of an object, and compares the *types* of values of those slots with the *type specifier* of the sorted out rules. The in this way found rules are then applied to the object (more about rules in the next chapter).

As of this writing not all of the concieved subclasses of the SMTOBJ abstract class are implemented yet, however the three basic classes of objects which I will address hereinafter, namely the chase class, the mtype class and the composing-stick class are to a large extent implemented and will be covered in the following.

2.2 Chase (Subclass of SMTOBJ)

Let us start this section by the definition of the chase in it's letter-press context. *letterpresscommons*[2] gives a good impression of what this tool is:



> ... a metal frame used to hold type in place while printing, usually on a platen press. Type or a base is locked up towards the center of the chase using furniture to position it and quoins to apply pressure against the X and Y axis. Lock up is done on a composing stone to assure that type is level. Before quoins are tightened a planer is tapped gently on the type surface to confirm the feet are flush against the stone, completely level.
>
> Typically made of cast iron, the size of a chase matches a specific press. The measurements, in inches, of the inside of the chase are also used to describe the press size. (So an 8 x 12 CP uses a chase that measures 8 x 12 on the inside of the correct chase.) Some chases, especially large ones, have handles at the top to assist the printer in both transporting and placing the chase in position on the press. A spider chase is made for locking up small forms in a large chase by providing a small frame with "legs" that attach to the larger chase.

1 The domain can be a keyword or one of the symbols `stacked-composing-stick`, `horizontal-composing-stick` or `vertical-composing-stick`. The last three domains are used internally by the engine and can be used for defining rules by one of the macros `define-stacked-rule`, `define-horizontal-rule` or `define-vertical-rule` correspondingly.

2 https://letterpresscommons.com/general-tools-and-supplies/

> Chases can also be useful with flatbed presses for a variety of situations. Composition of complex forms can be created on a composing stone in a large chase, then brought to the press before printing. Small chases are useful to create angled type in a flatbed press, easily locked up with triangular cut furniture and magnets.

As in the letter-press print, SMT's chase is also concieved as a frame for other objects. Children of a chase inherit some of their properties from their parent. Some properties (e.g. coordinates or dimensional properties) are *affected* by those of their parent chase. For example changing the horizontal position of a chase on the page, will cause all the descendants of the chase to be displaced horizontally in the same direction by the same amount (of pixels) as well. This does not happen only if we have set the *absolute* position of a descandant! Every chase object has the following 10 positional attributes: `absx, absy, x-offset, y-offset, x, y, left, right, top` and `bottom`. Each chase in addition possesses the 4 dimensional attributes `width, height, x-scale` and `y-scale` which affect the dimensions of the chase. These methods are *places*, hence can be read from and written to. From the mentioned *posidim*[1] attributes only `absx, absy, x-offset, y-offset, x-scaler` and `y-scaler` can be set at object-creation time. They can as well be modified at any later stages of object's life-time before the rule-application time. The remaining posidim attributes are computed and set after the object initialization, based on the posidims of object's parents (and possibly object's own children in case of composing-sticks).

Chases are printed on the score as filled rectangles by setting the `:chase-visible-p` slot to true. This rectangle, is the *smallest possible* rectangle which includes all the graphics of an object. The definition of this rectangle however is for mtypes and composing-sticks a bit different and is illustrated in the following sections.

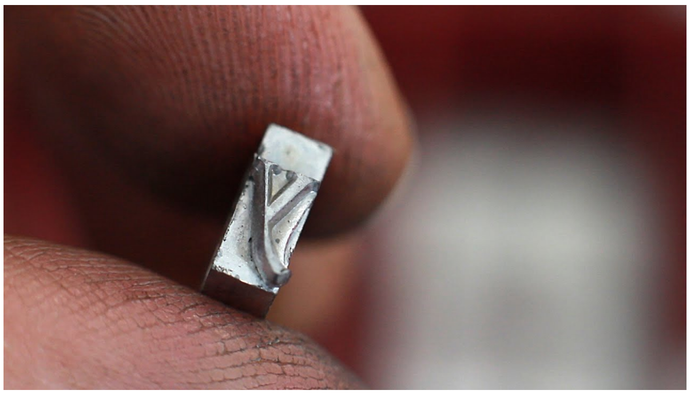
2.3 Mtype (Subclass of Chase)

A *sort* or *type* in the letter-press print is defined by Wikipedia[2] as:

> ... a piece of type representing a particular letter or symbol, cast from a matrix mold and assembled with other sorts bearing additional letters into lines of type to make up a form from which a page is printed.

1 From now on in this writing, I will refer to the 14 abovementioned positional-dimensional attributes of a chase as the *posidims*.

2 `https://en.wikipedia.org/wiki/Sort_(typesetting)`



Correspondingly, a mtype in SMT[1] represents a musical symbol (a glyph) such as a clef, a note-head and so on. Each mtype has a so-called Bounding Client Rectangle (BCR) structure[2], which describes the coordinates of the smallest rectangle framing the entire symbol shape. The BCR structure contains some of the original posidim values, which originate directly from the defined SVG-font *without* any scalings applied to them yet[3]. We can see one such structure and it's slots below for the alto clef:

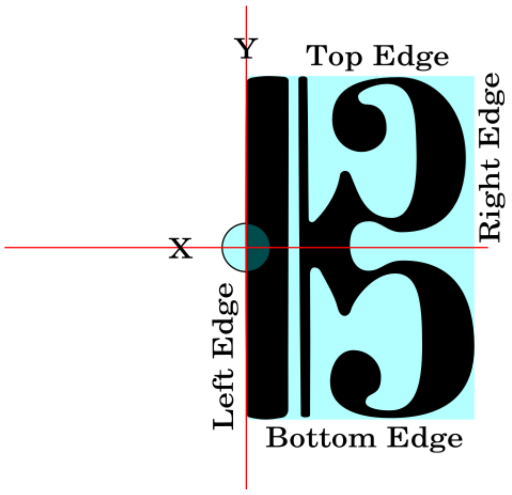
```
SMTNGN> (bcr (clef '(c . 4)))
#S(BCR
   :X -2.6666718
   :Y -503.4
   :WIDTH 671.68335
   :HEIGHT 1007.93335
   :TOP -503.4
   :RIGHT 669.01666
   :BOTTOM 504.53336
   :LEFT -2.6666718)
```

The slot values of the above BCR structure reveal some very important aspects about the design of the glyph in question: the top edge of the glyph (which corresponds to it's y coordinate) is almost the half of it's height off the vertical origin (note that the original position of this glyph is assumed to be at the top-left corner of the page, i.e. at (0 0)). The same story is valid for the left edge (corresponding to the x coordinate of the structure) which is off the horizontal origin to a small extent, namely -2.6666718 pixels. Let's examine some of these properties through the following image:

1 Mtype stands for movable type, a name modification to avoid name clash with the in SBCL locked symbol `type`.

2 A term borrowed from a javascript method with the same name.

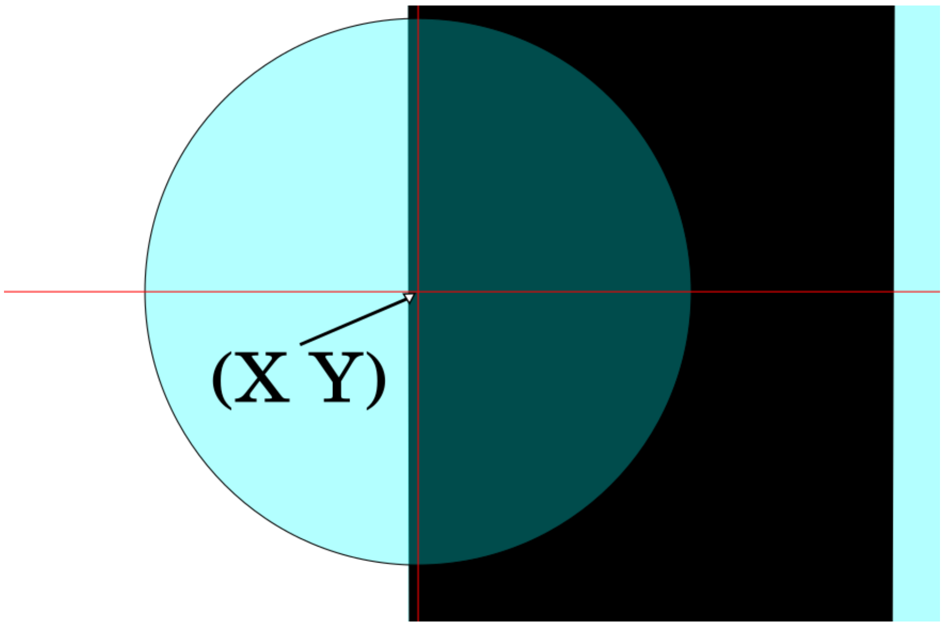
3 Any mtype object is subject to a top-level horizontal as well as vertical scaling by the *system* factor 0.02819834 which is controlled by the *user* factor initially set to be 1. The system-factor for top-level scaling is dynamically obtained by (((4 * current-staff-space) / current-alto-clef-height) * user-factor). This originates from the fact that the height of the stave in a certain rastral scale should be equal to the height of the alto clef in that rastral-scale, which in turn is equal to 4 staff-spaces (H. Chlapik, *Die Praxis des Notengraphikers*, page 30).



On the left side of the picture we see a red cross with a small circle in the middle of it. We call this the *marker* of the object, which shows us the exact position of the object on the page, being the center of the marker-circle where the two lines of the marker's cut across each other. The other four attributes are self-evident: on the vertical axis the left and the right edges mark the left-most and the right-most borders of the symbol. Likewise the top and the bottom edges visualize the clef's top-most and bottom-most boundaries on the horizontal axis. The BCR is then the rectangle in cyan color which encompasses the clef's glyph based on the four computed margins[1]. What might not be obvious at first sight though, is the fact that the x coordinate of our symbol (i.e. it's horizontal positioning on the page) is not the same as it's left edge! And despite the fact that in our example it is already obvious that the y coordinate differs from the top edge thoroughly, it seems appropriate to mention that the y and the top values differ also (albeit possibly marginally) for most glyphs. The xy coordinates of a mtype are computed based on the positions we declare explicitely for the object at object's creation-time (e.g. by initializing `:absx, :absy, :x-offset` or `:y-offset` slots), or implicitely through object's parents. These informations are stored in the `x` and `y` methods and should not be confused with the `:x :y` slots of the BCR structure (which are practically duplicates of the `:left :top` slots respectively)!

To see this difference and the actual xy values let us zoom into the marker's center-circle of our alto clef:

1 In case of a mtype, the BCR and it's chase designate the *same* rectangle. It is this BCR/CHASE rectangle which is colored.



We can now see clearly that the left edge of the clef symbol is to a small extent (the -2.6666718 pixels) far left than our actual `x` value. Although in this case this difference is quite small and imperceptible, for further possibly more complex operations involving mtypes, it is vital to be conscious of this difference.

Note that mtypes build the smallest cell of visible SMT objects. Hence, they may not be a container for any other objects. However since they inherit from the SMTOBJ class, they have a list where every SVG element needed for rendering the mtype on the page is kept in. The content of this list can be inspected or extended by calling the accessor methods `svglst` or `(setf svglst)` respectively on a mtype object. Note that at the time of this writing the reasonable time for investigating the content of `svglst` is *not* at object creation-time, since the engine needs more information about the potential parents of a mtype for creating proper SVG elements. These information are lacking at the object creation-time[1].

2.4 Composing Stick (Subclass of Chase)

In the letter-press print a composing stick is a device used to hold metal type letters while composing lines of type ready for letter-press printing. It is defined by **www.stbrigidpress.net**[2] as:

> ... the tool that holds the pieces of type that are being set (or, "composed"). Usually made of brass or steel, the composing stick is held in the non-dominant hand while the typesetter lines up each desired letter. The stick is adjustable, according to how long the line of type needs to be.

1 By setting the `:toplevelp` of an object to true we inform the engine that it is ready to be rendered in it's current state and hence gets the complete `svglst` immediately.

2 https://www.stbrigidpress.net/blog/a-letterpress-lexicon-part-2

As in letter-press printing, SMT's composing-sticks are conceptually tray-like tools used to assemble pieces of mtype into larger units (of moments, lines and columns). In SMT the concept of composing sticks has been broadened to encompass not only the possibility of lining up symbols horizontally (as in the letter-press printing), but also vertically, as well as piled up on top of eachother (stacked). Thus we have a comparatively small amount of tools at our disposal which give us a wide range of arrangement possibilities for organizing symbols into lines of symbols, columns made up of lines of symbols and so forth! Composing-sticks are thus containers which can accommodate mtype objects, but also other composing-sticks who probably contain other composing-sticks loaded with mtypes, thus in the family-tree of a SMT score a composing-stick can take on the role of an ancestor as well as a descendant of other composing-sticks at the same time! This implies that many of the behaviors of composing-stick objects have to be defined recursively.

A composing-stick object has an initial rectangular frame (see the section on chase). The height of this rectangle has been set to the height of the alto clef from the currently in-use font family. This is because in the Western music notation, the height of the alto clef is supposed to be exactly equal to the height of the staff system[1]. As composing sticks have been concieved to accommodate other objects, many of their attributes (and their visibility) is dependent on their contents (the value of the `:content` slot). A composing stick with an empty `:content` list is an abstract frame with the only dimensional attributes *height, top* and *bottom* initialized. Below we create a composing-stick of type *stacked*, using the function `stick` by providing it the only required argument, namely the type of the composing stick. Note that whenever the engine gets the instruction to render one or many SMT objects, it assumes that the exact starting point position is included in the *top-level* object. If we have not provided this information for a top-level object (which is done explicitely by having initialized both `:absx` and `:absy` slots of the object to the starting point we wish on the page), then the engine goes ahead and sets these two important attributes to some reasonable default values for us (to be more precise, the `:absx` will be set to the value of the constant variable `+left-margin+` being 136.06299 pixels wide, and

1 Or in other words: the height of the staff system should be filled out by the vertical space of the alto clef, which must exactly be equal to four staff-spaces in the used rastral measurement.

the `:absy` will be set to the value of the constant variable `+top-margin+` being 211.65355 pixels high). Every other computation of positions of the top-level object and *all* it's descendants (by inheritance) takes place based on the absolute position of the *root* top-level object. To avoid an error, we also declare our stacked stick to be the top-level element of the score by setting it's `:toplevel` slot to `t` which causes the engine to take care of setting the appropriate starting point for us:

```
SMTNGN> (defparameter *stacked-stick* (stick :stacked :toplevelp t))
*STACKED-STICK*
```

Now when we render our composing-stick:

```
SMTNGN> (render (list *stacked-stick*))
NIL
```

we get no errors, which means that the rendering has been successfully terminated. Let us now examine some of the properties of our object[1]:

```
SMTNGN> (list (absx *stacked-stick*) (absy *stacked-stick*))
(136.06299 211.65355)
SMTNGN> (list (x-offset *stacked-stick*) (y-offset *stacked-stick*))
(0 0)
SMTNGN> (list (x *stacked-stick*) (y *stacked-stick*))
(136.06299 211.65355)
```

`:absx` and `:absy` have indeed been set to the default page margins. Every SMT object has also the two complementary positional attributes: `:x-offset` and `:y-offset` which in contrary to the absolute parameters are always initialized to zero. Most of the time we use the above-mentioned properties at the object's initialization time. Once the objects are created, based on their standpoints in the hierarchy of their family-tree, they need an appropriate computation of their *actual* places on the page. Note that whereas a specified absolute coordinate point (i.e. `:absx, :absy`) will set the xy coordinate of the object to the desired point disregarding of it's ancestors, offsetting (i.e. via `:x-offset, :y-offset`) will merely move the object to the left or right, top or bottom respectively from it's *current* standpoint by the given amount[2]. The `x` and `y` functions above give us always the exact actual position of objects, disregarding their nested level inside of their family's hierarchy and positional values of their ancestors. Both of these reader-functions have a counterpart writer-function `(setf x)` and `(setf y)` which enable us to change the actual position of objects at any stage of their life-time. Since these functions provide us with the actual information about positions we would want to know and take care of the computations

1 All of the numerical values returned by the engine are in pixels.
2 At the time of this writing, the unit of input is interpreted only as pixel.

necessary for it, we most of the time will be working with them.

Next we browse through the vertical dimensions of our composing stick:

```
SMTNGN> (list (height *stacked-stick*) (fixed-height *stacked-stick*))
(28.422043 28.422049)
SMTNGN> (top *stacked-stick*)
197.45851
SMTNGN> (bottom *stacked-stick*)
225.88055
```

Beside the **`height`** attribute who's value is dynamic and can be changed[1], composing sticks also have a constant-height named **`fixed-height`** which is the initial height of each stick and corresponds to the height of the alto clef symbol as described earlier. We can inquire the height of the alto clef and insure that our stick has the exact same initial height:

```
SMTNGN> (height (clef '(c . 4)))
28.422049
```

The top and bottom coordinates of our stick are retrieved also with the two reader functions `top` and `bottom`. These have corresponding writer-functions as well: `(setf top)` and `(setf bottom)`. Before taking a look into the produced graphics, let us examine the vertical coordinates better. We increment the `y` attribute of our stick object by 1 pixel:

```
SMTNGN> (incf (y *stacked-stick*))
212.65355
```

and as we might expect the other vertical coordinates are updated accordingly:

```
SMTNGN> (top *stacked-stick*)
198.45851
SMTNGN> (bottom *stacked-stick*)
226.88055
```

In accordance with the expectation that changing the coordinates shouldn't have an impact on the dimensions of an object, of course the height of our composing stick still holds it's old value:

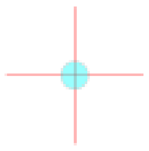
```
SMTNGN> (height *stacked-stick*)
28.422043
```

1 Either explicitely by resetting it to some other values, or implicitely dependent on the positionings of stick's descendants inside of it's borders.

The reverse should logicaly be not the case:

```
SMTNGN> (incf (height *stacked-stick*))
29.422043
SMTNGN> (list (bottom *stacked-stick*) (top *stacked-stick*))
(227.88055 198.45851)
```

Changing the height of the stick moves it's bottom edge. Later in this chapter we will see that these actions will have a desired impact on all ancestors as well as descendants of a composing stick in a recursive manner.
Let us now have a look at the produced SVG graphics and see what has been rendered[1]:

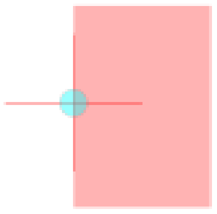


The only thing we can see is the *marker* of our object, which specifies it's xy coordinate. As we saw, although the object even has an abstract frame with corresponding coordinates and dimensions, this frame is not visible[2] until the stick contains some other objects. To be more precise, this is the case because without any content the `width` of our stick is keeping it's initial value of 0. If we modify it's width and re-render:

```
SMTNGN> (setf (width *stacked-stick*) 20)
20
SMTNGN> (render (list *stacked-stick*))
NIL
```

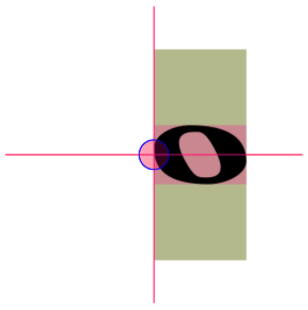
we can see the effect:

1 Note that for saving space, I have trimmed and printed here only *the part* of the SVG graphics which contains our graphics and not the entire page.
2 Note that the visibility of objects's chase is set to true by default *only* in the testing version of the engine.



To see the behavior of composing sticks containing other objects we do the following small experiment: we create a composing-stick (S1) which contains another composing stick (S0) which itself accommodates a note-head object (N) and look at their right edges:

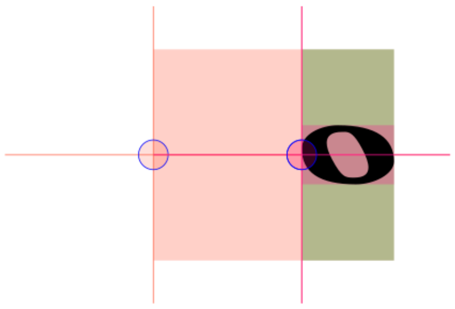
```
(let* ((n (notehead "s0" :id 'n))
       (s0 (stick :stacked :chase-fill "green"
                           :id 's0
                           :content (list n)))
       (s1 (stick :stacked :toplevelp t
                           :id 's1
                           :content (list s0))))
  (render (list s1))
  ;; Print all left edges
  (print (mapcar #'left (list n s0 s1)))
  ;; Print all right edges
  (print (mapcar #'right (list n s0 s1)))
  )

;; => (136.06299 136.06299 136.06299)
;; => (148.49846 148.49846 148.49846)
```



Now we shift the left edge of the S0 by 20 pixels before rendering:

```
(let* ((n (notehead "s0" :id 'n))
       (s0 (stick :stacked :chase-fill "green"
                           :id 's0
                           :content (list n)))
       (s1 (stick :stacked :toplevelp t
                           :id 's1
                           :content (list s0))))
  (incf (left s0) 20)
  (render (list s1))
  ;; Print all left edges
  (print (mapcar #'left (list n s0 s1)))
  ;; Print all right edges
  (print (mapcar #'right (list n s0 s1)))
  )
;; => (156.06299 156.06299 136.06299)
;; => (168.49846 168.49846 168.49846)
```

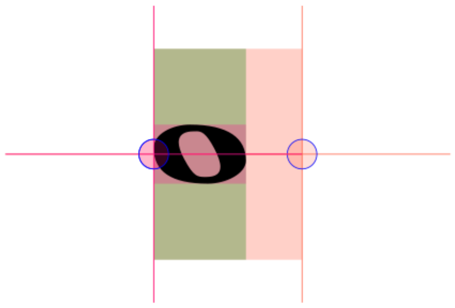
As we can see, this causes the lefts of S0 and it's child N to be incremented, but doesn't touch the left of the root object S1. Meanwhile since S1's right edge is dynamically updated to be *no less* than the *right-most* edge of it's contents, it is pushed alongside the other two right edges by 20 pixels:



Now we shift the left edge of S0 by the same amount, this time to the *left*:

```
(let* ((n (notehead "s0" :id 'n))
       (s0 (stick :stacked :chase-fill "green"
                           :id 's0
                           :content (list n)))
       (s1 (stick :stacked :toplevelp t
                           :id 's1
                           :content (list s0))))
  (decf (left s0) 20)
  (render (list s1))
  ;; Print all left edges
  (print (mapcar #'left (list n s0 s1)))
  ;; Print all right edges
  (print (mapcar #'right (list n s0 s1)))
  )

;; => (116.06299 116.06299 116.06299)
;; => (128.49846 128.49846 136.06299)
```

This time, while all the left edges are moved together with the left edge of S0[1], the right edge of S1 is again set to it's initial *left* which was computed as being either the left-most of it's contents or it's own x coordinate! This is because the right edge of a composing stick will never become *smaller* than it's own x coordinate:

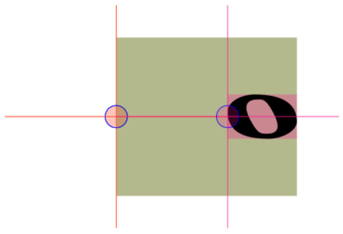


Extending the x of N[2] leaves both lefts of it's ancestors untouched, while it pushes their right edges[3]:

```
(let* ((n (notehead "s0" :id 'n))
       (s0 (stick :stacked :chase-fill "green"
                           :id 's0
                           :content (list n)))
       (s1 (stick :stacked :toplevelp t
                           :id 's1
                           :content (list s0))))
  (incf (x n) 20)
  (render (list s1))
  ;; Print all left edges
  (print (mapcar #'left (list n s0 s1)))
  ;; Print all right edges
  (print (mapcar #'right (list n s0 s1)))
  )
```

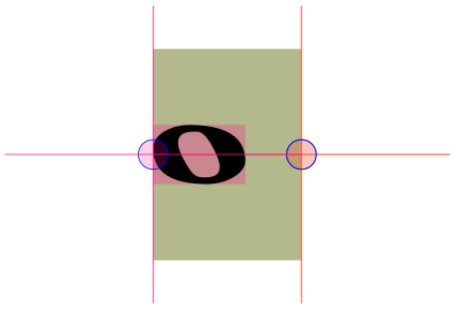
1 The left edge of N moves, because it is contained in S0 and is subject to the movements of it's parent and the left edge of S1 moves, because it is updated to be *no less* than the *left-most* of it's children!
2 Changing left and right edges of a mtype is an error!
3 Their original right edges where placed at 148.49846.

```
;; => (156.06299 136.06299 136.06299)
;; => (168.49846 168.49846 168.49846)
```



Pushing the x-coordinate of N to left shifts all the lefts by the same amount, but can not move the right edges of it's two ancestors beyond their own x coordinates:

```
(let* ((n (notehead "s0" :id 'n))
       (s0 (stick :stacked :chase-fill "green"
                           :id 's0
                           :content (list n)))
       (s1 (stick :stacked :toplevelp t
                           :id 's1
                           :content (list s0))))
  (decf (x n) 20)
  (render (list s1))
  ;; Print all left edges
  (print (mapcar #'left (list n s0 s1)))
  ;; Print all right edges
  (print (mapcar #'right (list n s0 s1)))
  )

;; => (116.06299 116.06299 116.06299)
;; => (128.49846 136.06299 136.06299)
```



Changing the width attribute of objects would accordingly affect their family tree on the horizontal scale. On the vertical scale displacing and modifying the top, bottom, y and height attributes of objects would change the vertical posidim attributes of their descendants and their ancestors correspondingly. To not repeat similiar effects, I will confine myself to the above demonstrations and would refer to the documentation of the engine for further readings.

CHAPTER 3

Definition of Rules

3.1 Introduction

We begin examining some fundamental aspects of SMT's rule protocol by going through the definition a basic and simple rule. But before that, we need to get familiar with some of the terminology used in the protocol.

3.1.1 Rule Tables & Domains

Rules are organized into the so-called rule-tables, which are themselves kept into a central ruletable registry named `*domain-registry*`. This registry itself is a hash-table with it's keys being keywords associated with different rule-tables. In SMT's jargon we call these keys *domains*. Before rendering any objects, the engine will look into a list of already registered domains (the value of the variable `*domain-registry*`) for extracting and applying the for the object defined rules. Hence, before we proceed to the actual definition of our rule, we must make sure that the engine knows about the domain in which we want to define our rule (note that each domain can contain more than one rule). SMT comes with some pre-defined domains named after the common musical clefs, e.g. `:treble, :bass, :alto` etc. Besides there are three pre-defined domains for composing sticks: stacked, horizontal and vertical (more on defining rules for composing sticks later). Although the domain we are going to use for our first rule (`:treble`) is already registered by SMT, we assume that this is not the case and start off by informing the engine about our domain. For this, we use the function `register-ruletable` and pass to it as it's single argument the name of our domain:

```
(register-ruletable :treble)
```

This creates a new ruletable with the domain name `:treble` for us, and registers it into the central domain registry. Trying to define a rule in an unkown domain is an error.

3.1.2 Explaining The DEFRULE Macro

Being certain that the engine now knows about our domain, we can carry on with the actual definition of our rule. We use the keyword `defrule` for defining a new rule, which has the following syntax:

defrule *targets domain* (`&optional` *id* (*typespec* `t`) (*index* `0`)) *lambda-list* `&body` *body*

We already know about the domain parameter, so lets proceed to the next parameter list: `(id typespec index)`[1]. The `id` argument is a keyword designating the name of one of the slots of an object, who's value must be of the same type as designated by `typespec` for the application of our rule to take place. It is important to note that for checking the types of object's slot-values against `typespec`, SMT uses the type system of Common Lisp, so that any valid Common Lisp- or user-defined types could be used as arguments to the `typespec` parameter. To understand this part better, we continue by intoducing a practical case and breaking it apart. When we create a SMT-object, we have to supply it with a unique (under `eq`) id[2]. This id and it's uniqueness will be very important to our rule-definition protocol, so SMT makes sure that each object has a unique id by registering each used id into a hash-table (called `*object-registry*`) upon initializing an object, and will complain when an id is going to be used more than once. It also automatically provides a unique id for our object, should we have forgotten to do so! Below we create a clef object, a subclass of the mtype class, and supply it with the id `'bass-clef`:

```
(defparameter *bass-clef* (clef :f :id 'bass-clef :domain :treble :ruleids
'(:id)))
```

This object has been specified to read from the `:treble` rule-table for the application of rules to him. It also has a rule-ids list with one keyword in it: `:id`. This tells the engine to look at the value of the slot `:id` in this object and to check it's value-type against the `typespec` of rules in the `:treble` rule-table to locate the applicable rule. All SMT objects have a list of applicable rule-ids which can be used in defining rules (as argument to the `id` parameter) and as items in the `ruleids` slot of the object. To know what rule-ids are permitted for each object we can look into it's class's list of applicable rule-ids. For the class SMTOBJ we can read these ids[3] by:

```
(mapcar #'car *smtobj-applicable-rule-ids*)
;; => (:ID :DOMAIN)

*smtobj-applicable-rule-ids*
;; => ((:ID #<STANDARD-GENERIC-FUNCTION SMTNGN::ID (1)>)
;;     (:DOMAIN #<STANDARD-GENERIC-FUNCTION SMTNGN::DOMAIN (1)>))
```

As we can see, `*smtobj-applicable-rule-ids*` is an alist where the key of each entry is the name of a slot and the value of the entry is the slot's reader function.

Let us now see the definition of a simple rule which would be applicable to our bass clef:

1 The first parameter `targets` is a list for storing *types* of objects for which the domain is being defined. I will explain this parameter later.

2 Not to be confused with the `id` parameter of `defrule`!

3 The list shown here is not the complete final version.

```
(defrule (clef) :treble (:id (eql bass-clef) 0) ()
  "Prints a message to the console, once the rule has been applied."
  (print '(my first rule applied!)))
```

Here the `targets` parameter is set to the list `'(clef)`. `targets` is a list which should contain one or more class names of objects we want our rule to be applied to. Also the parameter list `(id typespec index)` is bound to the arguments `(:id (eql bass-clef) 0)` respectively. When the engine is going to render our clef object into graphics, it first reads the values of two slots of the object: the `:ruleids` and the `:domain`. A non-empty `ruleids` list tells the engine that there are some rules to be applied to our object before it is rendered. The engine then proceeds by calling up the value of the rule ids specified in the `ruleids` list. It then retrieves the fitting rule with the correct type-specifier from the `:treble` rule-table, and calls the rule's closure function. In the above example, we are restricting our type-specifier to be exactly the symbol `'bass-clef`. So the above rule is applicable to any objects whose slot `:id` has been initialized to the symbol `'bass-clef`.

The next parameter list is the lambda-list, which offers access to the object in question and it's ancestors. The first parameter of this list, if supplied, will be bound to the object itself. Any subsequent parameters are then bound to the ancestors of the object, so that the second parameter would refer to the object's direct ancestor, the third to the direct ancestor of the object's direct ancestor an so forth. In our example, since we are not going to work with any of the objects in the family tree of the object, we leave out this lambda-list to remain empty. The next forms constitute the body of our rule, which will be evaluated as an implicit `progn` at the rule-application time. Here we are only asking for the list `'(my first rule applied!)` to be printed out to the console, once the rule has been applied:

```
(render (list *bass-clef*) :draw nil)

;; => (MY FIRST RULE APPLIED!)
```

Note that by default the parameter `typespec` is bound to the base type `t`, which means if we hadn't supplied a type-specifier argument for this parameter, our rule would be applicable to *any* objects with a `ruleids` list of `'(:id)`, no matter what type the value of their `:id` slot would be.

Since the value of `ruleids` is a list, we might already have deduced that this list could accommodate more than one rule-id, which is to say applying more than one rule to our object. This is where the defrule's `index` parameter comes in to play. Imagine we wanted to apply a second rule to the object before it gets rendered, based on the value of it's `:domain` slot. In the following we define a new rule which will be applicable to all objects who's `:domains` are `eql` to `:treble`. Notice here the `index` argument which is set to 1, which requires the `:domain` rule id to appear at the `second` place (rule ids count is zero-based) of clef's `:ruleids` list. As long as these are the case, our rule will set the color of the clef symbol to red:

```
(defrule (clef) :treble (:domain (eql :treble) 1) (obj)
  "Sets the color of the symbol to red, if the :DOMAIN of OBJ is EQL to :TREBLE."
  (setf (mtype-fill obj) "red"))
```

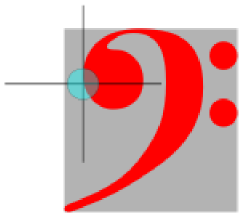
Now upon re-rendering our clef object:

```
(render (list (clef '(f . 4)
                    :id 'bass-clef
                    :toplevelp t
                    :absx 100 :absy 100
                    :domain :treble
                    :ruleids '(:id :domain))))
```

our initial rule prints to the console:

```
;; => (MY FIRST RULE APPLIED!)
```

while the following graphic has been rendered:



Just for the sake of better understanding, let us swap our rule id's in the `ruleids` list of the base-clef and see what happens:

```
(render (list (clef '(f . 4)
                    :id 'bass-clef
                    :toplevelp t
                    :absx 100 :absy 100
                    :domain :treble
                    :ruleids '(:domain :id))))
```

No rules defined for ruleid DOMAIN at ruleid-index 0 in domain TREBLE [Condition of type SIMPLE-ERROR]

SMT will complain about the absence of our desired rule-id at the first position in our `ruleids` list! Although the rendering process has been ceased due to the error, we can be sure that the very same problem also holds true for the `:id` rule-id at the second position. Hence, the order of rule-ids is important to allow for successive applications of rules to affect their previous ones, whenever desirable.

3.2 Composing Sticks

There is basically nothing much special into the definition of rules for composing sticks than pictured in the introductory rules above. The only new aspect we are going to handle will be that we can reach out for the objects themselves and all of their ancestors and every detailed information they carry with them. For this purpose we need to fill out the lambda-list of our `defrule` macro with the corresponding parameter names. In the next sub-section we will play on some of the redundancies found in the common western music notation to exemplify how we can refer to the object's *family tree* and maybe manipulate their attributes via our rules. Here we will also introduce three new special rule-definition macros for composing sticks, namely `define-stacked-rule`, `define-horizontal-rule` and `define-vertical-rule` for defining rules for the stacked, horizontal and vertical sticks respectively. As these macros are merely convenient forms for defining rules for the particular rule-tables, apart from the name of the domain which is ommited[1], they resemble in syntax the `defrule` macro:

define-stacked-rule *targets* (`&optional` *id* (*typespec* `t`) (*index* `0`)) *lambda-list* `&body` *body*
define-horizontal-rule *targets* (`&optional` *id* (*typespec* `t`) (*index* `0`)) *lambda-list* `&body` *body*
define-vertical-rule *targets* (`&optional` *id* (*typespec* `t`) (*index* `0`)) *lambda-list* `&body` *body*

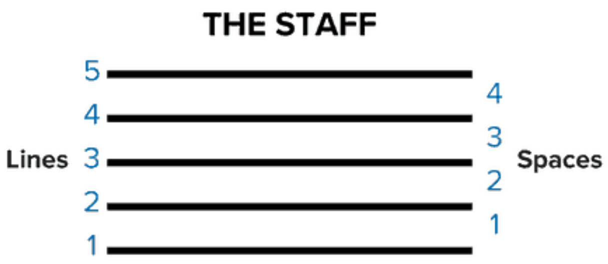
3.2.1 Drawing Staff Lines on Stacked Sticks

Since we will need staff lines for checking the pitches later, we start this section by illustrating how to draw staff lines on a stacked composing stick. As SMT's native output is Scalable Vector Graphics, it is easy to instruct SMT's engine to render our own SVG elements within our score. Before we proceed to drawing the stave, let us recap some of the properties of a stakced stick which would help us in better understanding of the definition of our rule. Composing sticks are abstract containers for housing musical symbols (mtypes) or even other composing sticks. They all initially have the shape of a rectangle with a `fixed-height` which is equal to the height of the alto clef, following the tradition in music notation that the height of the alto clef shall fill out the horizontal space of stave. An important feature of sticks are their coordinates: the dimensional attributes of a stick are subject to changes based on the dimensional attributes of their children. So for instance having a treble clef in the content of the stick will increase it's height to the height of the treble clef. The stick's elasiticy in this sense resembles it to a bounding rectangle which permanently adjusts it's borders for including all it's children. It is important to comprehend that the posidims of a stick can change dynamically. This is definitely not

1 And which is inferred from the macro's name.

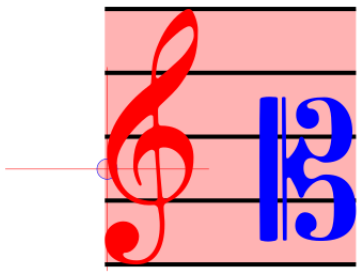
what we want for drawing stave. If we would rely on the height of a horizontal stick, which would say contain a treble clef and an alto clef, and draw the lines of our staff evenly spaced based on the height of the stick, the results will be not what we would expect to be (e.g. the staff spaces between staff lines would follow the height of the stick, which changes based on the height of it's children). The comments in the code snippets below explain my approach to draw the stave. Please note that since the SMT-engine is more of a graphics programming toolkit rather than a firmly pre-defined score-writer, there are in many occasions more than one possible approach to achieve the same results! To show more, I will demonstrate different approaches to show some of the potential pitfalls involved with working with coordinates.

Let us start off with the drawing of stave by abstracting our goal: we want to have the height of our stick being divided into four equal staff-spaces, marked off by five staff-lines:



```
;;; Define the rule for Stacked Sticks
(define-stacked-rule (stacked-composing-sticks) (:id) (me)
  "Generates five lines evenly distributed based on the vertical-space of the
stick ME."
  ;; Divide the height of the stick into four equal staff-spaces
  (let ((space-height (/ (height me) 4)))
    (loop
      ;; Line indices: 0, 1, 2, 3, 4
      :for line-idx :to 4
      ;; To obtain the vertical position of each line (it's Y) re-set the
      ;; Y of the current line to be the amount of vertical-space
      ;; passed from the TOP of the stick hitherto
      :for line-y = (+ (* line-idx space-height) (top me))
      ;; Push a line-element with it's X1 being the LEFT of the stick,
      ;; it's X2 being the LEFT + WIDTH of the stick and it's Y1 & Y2 as
computed above
      :do (push (svg:line (left me) line-y (+ (left me) (width me)) line-y
                          :stroke-width 30
                          :stroke-linecap "butt"
                          :stroke "black")
                (svglst me)))))

;;; Define two clefs (mtypes) as children to a horizontal stick
;;; (which would catch the above rule) and render the
;;; horizontal stick
(let* ((g (clef '(g . 4) :chase-visible-p nil
                         :marker-visible-p nil
                         :mtype-fill "red"))
       (c (clef '(c . 4) :chase-visible-p nil
                         :mtype-fill "blue"
                         :marker-visible-p nil
                         :x-offset 30))
       (s (stick :stacked
                 :toplevelp t
                 :ruleids '(:id)
                 :content (list g c))))
  (render (list s)))
```

The result is five staff lines distributed evenly within the height of the stick as we instructed the engine to do:



This is however not what we had intended, since the stick has modified it's height to fit the height of it's highest of children (being the treble clef) and since we have set the vertical origin of our lines to be the `top` of the stick (the rule line `:for line-y = (+ (* line-index space-height) (top me))` which will just evaluate to `(top me)` for the first line with `line-index` = 0), the stave positions and dimensions come to be invalid[1]!

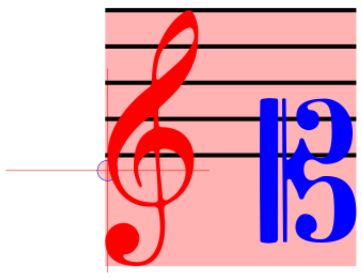
Instead of using the `height` of the stick which is a dynamic variable, we should use the fixed height of the stick which is equal to the height of the alto clef and of which we can be sure, that it never changes, no matter what the actual content of the stick would be.

This is the value of the `fixed-height` method, as mentioned above:

1 If you have noticed the thickness of the outer-most staff lines which exceed the top- and bottom-most edges of the treble clef, this is caused by the SVG line element, which grows in width from it's center off (the xy coordinate of the line) in both directions. For more information about this, consult the SVG specification.

```
;;; Define the rule for stacked sticks
(define-stacked-rule (stacked-composing-sticks) (:id) (me)
  "Generates five lines evenly distributed based on the vertical-space of the
stick."
  ;; Replace the dynamic HEIGHT with the FIXED-HEIGHT
  (let ((space-height (/ (fixed-height me) 4)))
    (loop
      :for line-idx :to 4
      :for line-y = (+ (* line-idx space-height) (top me))
      :do (push (svg:line (left me) line-y (+ (left me) (width me)) line-y
                          :stroke-width 30
                          :stroke-linecap "butt"
                          :stroke "black")
                (svglst me)))))
```

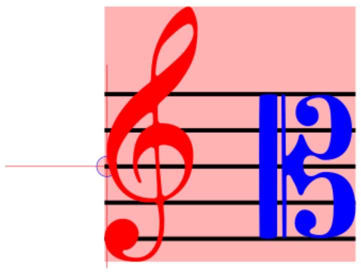
And when we re-render every thing:



We still have a problem with the *onset* of staff lines, although the staff spaces and the overall height of the staff are correct now. To adjust our rule to work, we must realize that the onset of our staff lines can not be set to the top edge of our composing stick, since this is also subject to change (here it is set to the top edge of the treble clef), but the y attribute! The y attribute is defined to be the center of the `fixed-height` of a stick.

```
(define-stacked-rule (stacked-composing-sticks) (:id) (me)
  "Generates five lines evenly distributed based on the vertical-space of the
stick."
  (let ((space-height (/ (fixed-height me) 4)))
    (loop
      ;; Build the 5 line indices around 0
      :for line-idx :from -2 :to 2
      ;; Proceed by adding the accumulated vertical space
      ;; to the Y coordinate of the stick
      :for line-y = (+ (* line-idx space-height) (y me))
      :do (push (svg:line (left me) line-y (+ (left me) (width me)) line-y
                          :stroke-width 30
                          :stroke-linecap "butt"
                          :stroke "black")
                (svglst me)))))
```

Now when we render our stick, we should see the desired behavior:



Note that in the above case instead of computing the staff space[1], we could as well have used the variable `*staff-space*` which has the same value:

```
SMTNGN> (let ((h (stick :stacked)))
          (= *staff-space* (/ (fixed-height h) 4)))
T
```

`*staff-space*` has been computed as described by Chlapik[2] as the rastral number 2. We could retrieve other rastral numbers as described by Chlapik using the function `get-staff-space` which has a single parameter for specifying the rastral number (between 2 and 8) and returns the corresponding staff space in pixels:

```
SMTNGN> (dotimes (i 7)
          (format t "~&Rastral Nr. ~D: ~D Pixels"
                  (+ i 2)
                  (get-staff-space (+ i 2))))

Rastral Nr. 2: 7.105512 Pixels
Rastral Nr. 3: 6.633071 Pixels
Rastral Nr. 4: 6.0472445 Pixels
Rastral Nr. 5: 5.790236 Pixels
Rastral Nr. 6: 5.2913384 Pixels
Rastral Nr. 7: 4.497638 Pixels
```

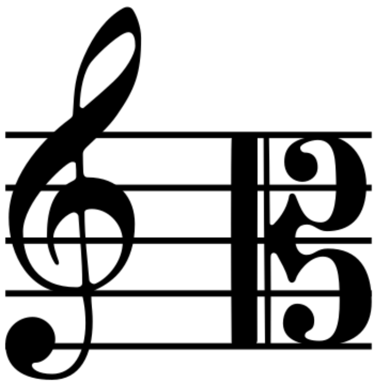
1 I.e. the vertical space between staff's lines.
2 H. Chlapik, *Die Praxis des Notengraphikers; Das Liniensystem (Rastral)*, page 32

```
Rastral Nr. 8: 3.855118 Pixels
NIL
```

Although we can any time use any values as staff-space which suit our needs best, it is advisable for consistency reasons to reset the value of the variable `*staff-space*` whenever possible, for this variable is used system-wide for doing some of the computations which are strictly dependent on staff spaces, such as vertical displacement of noteheads based on their pitches which happens in staff-space units in system-defined rules.

Let us now remove all the annoying extras from the image and look at our stave again:

```
(let* ((g (clef '(g . 4) :chase-visible-p nil
                         :marker-visible-p nil))
       (c (clef '(c . 4) :chase-visible-p nil
                         :marker-visible-p nil
                         :x-offset 30))
       (h (stick :stacked
                 :marker-visible-p nil
                 :chase-visible-p nil
                 :toplevelp t
                 :ruleids '(:id)
                 :content (list g c))))
  (render (list h)))
```

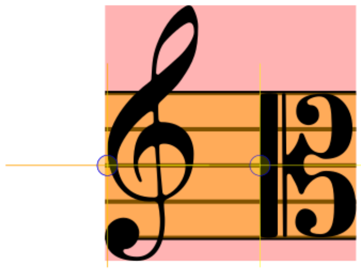


3.2.2 Vertical Adjustment of Symbols

There still is problem with the vertical position of our treble clef, which is placed wrongly on the middle line (the line with the `line-index` 0, which is at the same time the y coordinate of our stick)! This is because the treble clef's glyph has been designed in such a way, to have it's y coordinate at approximately it's bottom third[1]. This is the *point* which is used to move the glyph in the vertical direction.

To solve this problem, let's first have another look at the surface of our composing stick, the position of it's marker and the values of it's fixed vertical properties which we highlight by an orange rectangle. To draw this rectangle we also introduce a new member of the vertical fixed coordinates family of a composing stick, namely the `fixed-top` method:

```
(let* ((g (clef '(g . 4) :chase-visible-p nil
                         :marker-visible-p t))
       (c (clef '(c . 4) :chase-visible-p nil
                         :marker-visible-p t
                         :x-offset 30))
       (h (stick :horizontal
                 :marker-visible-p t
                 :chase-visible-p t
                 :toplevelp t
                 :ruleids '(:id)
                 :content (list g c))))
  (packsvg! h (svg:rect
                ;; X of rectangle
                (left h)
                ;; Y of rectangle
                (fixed-top h)
                ;; Width of rectangle
                (width h)
                ;; Height of rectangle
                (fixed-height h)
                ;; Color of rectangle
               :fill "orange" :fill-opacity .5))
  (render (list h)))
```



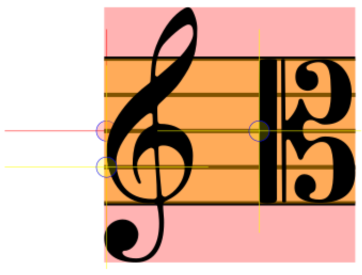
We see that these *musical joint-points* (i.e. the centers of the markers) of both clefs have

1 The left side of the lower half of clef's bowl, or where it's marker is drawn.

been vertically placed in the middle of the fixed-height of our composing stick, which we could express in terms of staff spaces, as 2/4 of it's fixed height. Now the correct position for the treble clef is on the second bottom line, or, again expressed in terms of staff spaces, 3/4 of composing stick's fixed height counted from top, which implies that we only should increment the y attribute of the treble clef by 1/4 of the composing stick's fixed height. Implementing this as a small rule for the treble clef is easy:

```
;;; This rule will apply to any mtype object, who's value of the
;;; :spn slot is of the same type as '(CONS (EQL G) UNSIGNED-BYTE)
(defrule (clef) :treble (:spn (cons (eql g) unsigned-byte)) (me parent)
  (incf (y me) (* 1/4 (fixed-height parent))))

;;; And render every thing
(let* ((g (clef '(g . 4) :chase-visible-p nil
                 ;; Important! Do not forget to tell the engine that
                 ;; we have a rule to be applied to this object.
                 :ruleids '(:spn) :domain :treble
                 :marker-visible-p t))
       (c (clef '(c . 4) :chase-visible-p nil
                         :marker-visible-p t
                         :x-offset 30))
       (h (stick :horizontal
                 :marker-visible-p t
                 :chase-visible-p t
                 :toplevelp t
                 :ruleids '(:id)
                 :content (list g c))))
  (packsvg! h (svg:rect
                ;; X of rectangle
                (left h)
                ;; Y of rectangle
                (fixed-top h)
                ;; Width of rectangle
                (width h)
                ;; Height of rectangle
                (fixed-height h)
                ;; Color of rectangle
                :fill "orange" :fill-opacity .5))
  (render (list h)))
```

And we obtain:



There still exists an ambiguity about the `:spn` rule-id of our defined rule above: what is the `:spn` slot? Each clef object, being an instance of the class `pitched-mtype` has to be initialized with a so-called Scientific Pitch Notation (spn) slot, which designates the pitch of the pitched-mtype object in the form of a dotted list where the first element of the list is a symbol designating the *name* of the pitch, and the second element of the list is a non-negative number designating the *octave* of the pitch[1], e.g. `'(C . 4)` is equal to the C of the fourth octave (the middle C on the piano). The first argument to any `clef` function should be such a dotted-list and can be fetched from the constructed object by calling the `spn` *accessor* on it:

```
SMTNGN> (spn (clef '(g . 4)))
(G . 4)
```

In our rule above we are specifying that the rule should be applicable to all objects with a spn-*name* `eql` to the symbol `g` and some non-negative spn-*octave*. However, we have been ignoring the octave information of the `:spn` slot, since the only information we needed to decide about the vertical positioning of the treble clef was it's *name*, thus a procedure like the following:

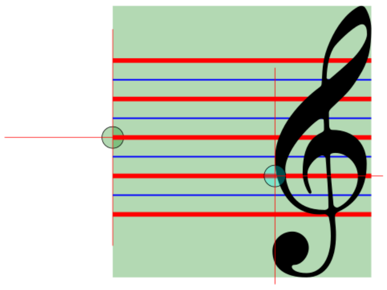
1 For more on spn visit https://en.wikipedia.org/wiki/Scientific_pitch_notation.

```
(let* (
       ;; Increment the octave of clef's spn
       (g (clef '(g . 5) :chase-visible-p nil
                         :ruleids '(:spn) :domain :treble
                         :marker-visible-p t))
       (c (clef '(c . 4) :chase-visible-p nil
                         :marker-visible-p t
                         :x-offset 30))
       (s (stick :stacked
                 :marker-visible-p t
                 :chase-visible-p t
                 :toplevelp t
                 :ruleids '(:id)
                 :content (list g c))))
  (packsvg! s (svg:rect (left s) (fixed-top s)
                        (width s) (fixed-height s)
                        :fill "orange" :fill-opacity .5))
  (render (list s)))
```

does result in exactly the same outcome as before! Although it might not make very much sense in musical terms, for the sake of demonstration we can modify our rule to take the octave of clef's SPN into play, so that different spn octaves would result in correct vertical displacement of our treble clef:

```
(defrule (clef) :treble (:spn (cons (eql g) unsigned-byte)) (me parent)
  (let ((octave (cdr (spn me))))
    ;; First step: move Y by 1/4 of the FIXED-HEIGHT towards
    ;; bottom of the page
    (incf (y me) (* 1/4 (fixed-height parent)))
    ;; Second step: displace Y by 7/8 of the FIXED-HEIGHT
    ;; to the direction of the OCTAVE
    (incf (y me) (* (- 4 octave) 7/8 (fixed-height parent)))))
```

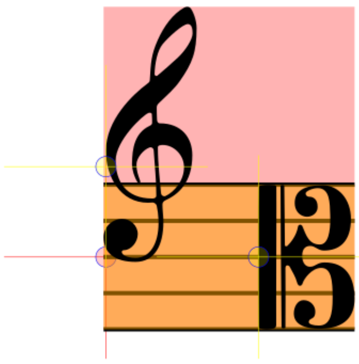
Since our starting point here is the fourth octave, we are transforming the *octave* information of clef's spn into a vactor of vertical displacement by subtracting 4 (our original octave) from the octave number of the spn. This vector is then multiplied by the vertical space equal to one octave, namely 7/8 of staff's height (read *stick*'s `fixed-height`). Below we can assure ourselves of the amount of vertical staff spaces equal to one octave:



Now changing the octave of the treble clef:

```
(let* (
       ;; Increment the octave of clef's spn
       (g (clef '(g . 5) :chase-visible-p nil
                         :ruleids '(:spn) :domain :treble
                         :marker-visible-p t))
       (c (clef '(c . 4) :chase-visible-p nil
                         :marker-visible-p t
                         :x-offset 30))
       (s (stick :stacked
                 :marker-visible-p t
                 :chase-visible-p t
                 :toplevelp t
                 :ruleids '(:id)
                 :content (list g c))))
  (packsvg! s (svg:rect (left s) (fixed-top s)
                        (width s) (fixed-height s)
                        :fill "orange" :fill-opacity .5))
  (render (list s)))
```

results in:



Thus we have systemized the octavation for our treble clef too. As said before, we might not need automating octavations for working with clefs in common Western music notation, but the concept of such automation will serve us well when we shortly define rules for vertical positioning of noteheads and accidentals.

Although the last version of our rule does the job, just for the sake of showcase, we could likewise have the y of the treble clef `setf`ed to be the bottom 1/4 of the stick's fixed height, and then displaced by the octave vector:

```
;;; This will do exactly the same thing as the previous version of the rule:
(defrule (clef) :treble (:spn (cons (eql g) unsigned-byte)) (me parent)
  (let ((octave (cdr (spn me))))
    (setf (y me)
          (+
           ;; Move the Y from the BOTTOM edge one staff space upwards
           (- (fixed-bottom parent) *staff-space*)
           ;; Displace Y in the direction of the OCTAVE vector
           (* (- 4 octave) 7/8 (fixed-height parent))))))
```

Please note that having a `:spn` slot for a clef object has been a design decision, to ease positioning of clefs on any desired place on a staff[1].

3.2.3 Automating Vertical Placement of Symbols Based on their Pitch

In the realm of western classical music, the natural use of *scientific pitch notation* is of course for notes and accidentals, which I will address in this section briefly. Note that at the time of this writing *note*s haven't been implemented thoroughly yet[2], so that for this section, I will focus on working with their pitch and note-head components, which are in any case adequate for our intention for demonstrating the automation of vertical positioning of note-heads on the staff, based on their scientific pitch notaion.

Let us start with the very basics! Each time we create a note object, we are implicitely creating a note-head object too. Usually the decision of choosing the appropriate note-head glyph is left to the engine and happens based on the value of the note's duration slot. We can however easily instruct the engine to go with the note-head of our choice and thus neglecting the duration for this matter[3], but before that, since we are going to deal with note-heads a quick excursion on them shall be helpful.

Note-heads are object's on their own right, descendants of the mtype class which we can create using the function `notehead`, which has a single positional parameter `label` and (like other functions for creating musical symbols) any number of *appropriate* keyword parameters for initializing an mtype object:

1 And also to be able to let the octave-displacements to be systemized via rules, as depicted above.
2 I.e. composed from all their different components note-heads, stems, flags and beams.
3 Notice that the value of the `:duration` slot is still used e.g. for computing the amount of horizontal space after the note's note-head.

```
SMTNGN> (notehead "s0")
#<NOTEHEAD {1001F81883}>
```

The argument to the `label` parameter should be a string which designates the name of a glyph[1].

We can throw a glance at all[2] valid labels for note-heads by calling the function `glyph-labels` which expects the *class* of a glyph-name as it's positional argument:

```
SMTNGN> (glyph-labels "noteheads")
("uM2" "dM2" "sM1" "sM1double" "s0" "s1" "s2" "s0diamond" "s1diamond"
"s2diamond" "s0triangle" "d1triangle" "u1triangle" "u2triangle" "d2triangle"
"s0slash" "s1slash" "s2slash" "s0cross" "s1cross" "s2cross" "s2xcircle" "s0do"
"d1do" "u1do" "d2do" "u2do" "s0doThin" "d1doThin" "u1doThin" "d2doThin"
"u2doThin" "s0re" "u1re" "d1re" "u2re" "d2re" "s0reThin" "u1reThin"  "d1reThin"
"u2reThin" "d2reThin" "s0mi" "s1mi" "s2mi" "s0miMirror" "s1miMirror"
"s2miMirror" "s0miThin" "s1miThin" "s2miThin" "u0fa" "d0fa" "u1fa" "d1fa" "u2fa"
"d2fa" "u0faThin" "d0faThin" "u1faThin" "d1faThin" "u2faThin" "d2faThin" "s0sol"
"s1sol" "s2sol" "s0la" "s1la" "s2la" "s0laThin" "s1laThin" "s2laThin" "s0ti"
"u1ti" "d1ti" "u2ti" "d2ti" "s0tiThin" "u1tiThin" "d1tiThin" "u2tiThin"
"d2tiThin" "u0doFunk" "d0doFunk" "u1doFunk" "d1doFunk" "u2doFunk" "d2doFunk"
"u0reFunk"  "d0reFunk" "u1reFunk" "d1reFunk" "u2reFunk" "d2reFunk" "u0miFunk"
"d0miFunk" "u1miFunk" "d1miFunk" "s2miFunk" "u0faFunk" "d0faFunk" "u1faFunk"
"d1faFunk" "u2faFunk" "d2faFunk" "s0solFunk" "s1solFunk" "s2solFunk" "s0laFunk"
"s1laFunk" "s2laFunk" "u0tiFunk" "d0tiFunk" "u1tiFunk" "d1tiFunk" "u2tiFunk"
"d2tiFunk" "s0doWalker" "u1doWalker" "d1doWalker" "u2doWalker" "d2doWalker"
"s0reWalker" "u1reWalker" "d1reWalker" "u2reWalker" "d2reWalker" "s0miWalker"
"s1miWalker" "s2miWalker" "s0faWalker" "u1faWalker"  "d1faWalker" "u2faWalker"
"d2faWalker" "s0laWalker" "s1laWalker" "s2laWalker" "s0tiWalker" "u1tiWalker"
"d1tiWalker" "u2tiWalker" "d2tiWalker")
```

The second optional argument to `glyph-labels` is `family`, which specifies the name of the font-family we want to retrieve the labels from, and which by default is bound to the variable `%glyph-family%`:

```
SMTNGN> %glyph-family%
:HAYDN-11
```

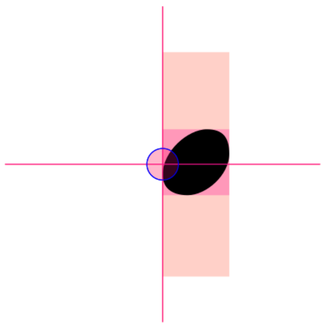
When we create a note by calling the function `note`, the musical symbol to be used as the note-head component is decided on based on the duration of the note (the keyword argument `:duration` to the `note` function, the first positional argument is a SPN designator as described earlier in the section 3.2.2):

```
SMTNGN> (render (list (note '(c . 4) :duration 1/4 :toplevelp t)))
```

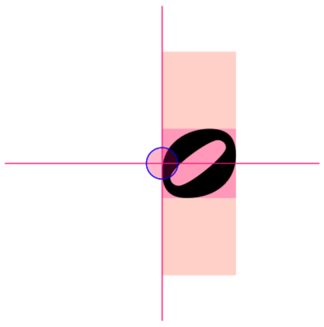
1 A glyph name is string of the form `"class.label"`, where the class specifies the category of the glyph e.g. noteheads, clefs, accidentals etc., and the label designates one of the defined lables in that category.

2 Making usage of user-defined musical symbols alongside the system-defined ones possible, is not implemented yet, but is planned for the future.

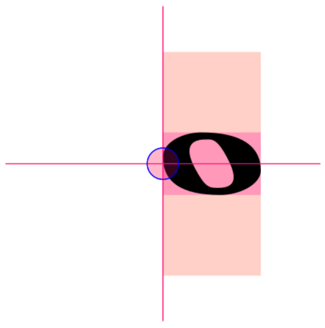
```
NIL
```



```
SMTNGN> (render (list (note '(c . 4) :duration 1/2 :toplevelp t)))
NIL
```

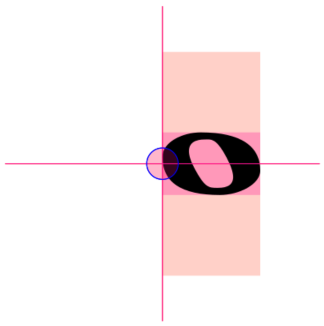


```
SMTNGN> (render (list (note '(c . 4) :duration 1 :toplevelp t)))
NIL
```



We could also give the `note` function our desired note-head explicitly, which would then take priority over it's `:duration`, if provided. Below, although our note object has a duration of a quarter note, it's note-head will be the object we explicitly create:

```
SMTNGN> (render (list (note '(c . 4)
                            :toplevelp t
                            :head (notehead "s0")
                            :duration 1/4)))
NIL
```



In the picture above, we can see two rectangles: the small pink rectangle surrounding the note-head, and the bigger rectangle which includes the note-head and it's bounding rectangle. If we have noticed (perhaps with suspicion!) that the bigger rectangle has approximately the height of the alto clef, it is because a note object is a *recognizable subtype* and an example of an internal usage of the stacked-composing-stick. A note is thus in principle a container for it's various components:

```
SMTNGN> (content (note '(c . 4) :duration 1))
(#<NOTEHEAD {1003A62583}>)

SMTNGN> (subtypep (type-of (note '(c . 4) :duration 1)) 'stacked-composing-stick)
T
T
```

Let us now return to defining our rule to make the pitch of a note impact it's vertical position on the staff. We can approach this in two different ways; either by defining a rule for the note-head part of the note to find it's position on the surface of it's parent (being the note object), or by instructing the note object itself to place it's child (the note-head) on the correct position. Either way, we need access to the vertical dimension of the stacked stick for classifying and associating pitches with it. In the following, I will demonstrate both of the mentioned approaches.

The computations for recognizing pitch can happen basically the same way as we already did for the treble clef in the previous section. There is though one enhancement here, namely we are specifying that the name part (the `car`) of the spn dotted list could be any symbol instead of a specific one, and based on that name we decide the correct amount of `*staff-space*` for shifting the `y` of our note-head. Thus we can cover the diatonic scale of not only the fourth octave, but for any other possible octave as well. We will see the results of this rule in the next pages.

```
(defrule (notehead) :treble (:spn (cons symbol unsigned-byte)) (me parent)
  "Assigns correct vertical positions to note-heads,
  based on their pitch-name and their octave."
  (let ((pitch-name (car (spn me)))
        (octave (cdr (spn me))))
    (setf (y me)
          (+ (- (fixed-bottom parent)
                (case pitch-name
                  (c (- *staff-space*))
                  (d (- (* .5 *staff-space*)))
                  (e 0)
                  (f (* .5 *staff-space*))
                  (g *staff-space*)
                  (a (* 1.5 *staff-space*))
                  (b (* 2 *staff-space*))))
             (* (- 4 octave) 7/8 (fixed-height parent))))))
```

Notice again, that here we are adding the symbol `notehead` to the list of target types of this rule.

Before we go on to the rendering part, I would like to (re)define the rule for drawing staff lines to the *background surface* of each note[1]:

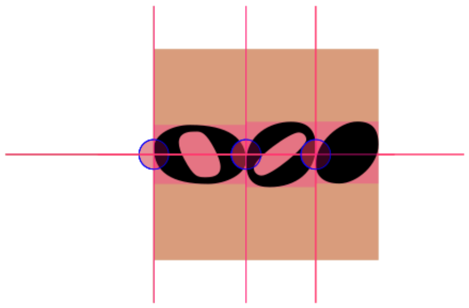
```
;;; Add five staff lines to the SVGLST of a stacked stick
;;; to be drawn by the engine:
(define-stacked-rule (stacked-composing-stick) (:id) (me)
  (loop
    :for line-idx :from -2 :to 2
    :for line-y = (+ (* line-idx *staff-space*) (y me))
    :do (push (svg:line (left me) line-y (+ (left me) (width me)) line-y
                        :stroke-width *staff-line-thickness*
                        :stroke-linecap "butt"
                        :stroke "black")
              (svglst me))))
```

Now we have defined all rules we need for pitches to be placed on the correct vertical positions. Before we render some notes with the rules applied to them, let us first consider the notes without the application of the above rules:

1 Notice also that deciding about *where* the staff lines should be drawn is to some extent arbitrary. I could as well be drawing lines directly on the note objects (which are stacked sticks themselves), but I prefer to create extra sticks for this purpose, so that the *backgroung surface* and the note object itself remain independent of each other!

```
(render
 (list
  ;; This horizontal stick will line up all it's children
  ;; horizontally based on their width's.
  (stick :horizontal
         :toplevelp t
         :content
         (list
           ;; First note surface, where the staff lines will be drawn
           (stick :stacked
                  :content
                  (list (note '(g . 5)
                              :duration 1)))
           ;; Second note surface
           (stick :stacked
                  :content
                  (list (note '(g . 4)
                              :duration 0.5)))
          ;; Third note surface
          (stick :stacked
                 :content
                 (list (note '(g . 3)
                             :duration 0.25))))))
```

In the resulting picture:



we learn about a central feature of horizontal composing sticks: they lock up their *direct* descendants horizontally by placing the *left* edge of every next child to the *right* edge of it's previous sibling[1]:

```
(define-horizontal-rule (horizontal-composing-stick) (:id) (me)
  (dolist (child (content me))
    (format t "~&~A Left: ~D, Right: ~D"
       (id child)
```

1 Similiar line-up process is applied to vertical sticks as well, which we are not going to discuss in this writing.

```
       (left child)
       (right child))))
```

Now by setting the slot `:ruleids` to `'(:id)` for our horizontal stick and re-rendering we can observe in the console that the line-up indeed happens:

```
SMTOBJ1255 Left: 141.06299, Right: 153.49846
SMTOBJ1258 Left: 153.49846, Right: 162.8885
SMTOBJ1261 Left: 162.8885, Right: 171.348
```

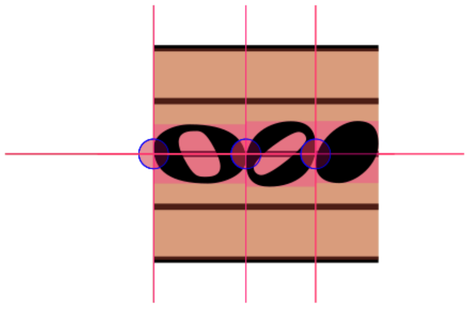
Apparently the line-up depends solely on the order in which we have listed the content of the horizontal stick. Although it is possible to scramble this horizontal line via rules[1], we should mostly rely on the order of objects computed by the horizontal and vertical sticks[2].

Now, first thing we want to see, are the staff lines drawn on each note background (the *stacked* sticks). This is easily done by specifying the correct rule-id for sticks we want our staff-line rule to be applied to:

```
(render
 (list
  (stick :horizontal
         :toplevelp t
         :content
         (list
           (stick :stacked
                  ;; Rule-id tells the engine to apply the staff-line rule
                  ;; to this surface
                  :ruleids '(:id)
                  :content
                  (list (note '(g . 5)
                              :duration 1)))
           (stick :stacked
                  :ruleids '(:id)
                  :content
                  (list (note '(g . 4)
                              :duration 0.5)))
           (stick :stacked
                  :ruleids '(:id)
                  :content
                  (list (note '(g . 3)
                              :duration 0.25)))))))
```

1 Rules are applied to objects *after* all necessary line-ups have taken place. This keeps the rule protocol at the top of the modification facilities for controling the behavior of SMT.

2 In future versions of SMT, these computations will happen based on spacing considerations.



Adding the rule-id for vertical positioning of notes needs some more explanations. Note that in the above sequence of notes, note-heads are being created implicitely via the supplied durations, which makes accessing them cumbersome (e.g. to supply them with the correct rule-id and the correct domain)[1]. There is a workaround for this: each composing stick can specify a *pre-processor* which will be evaluated before any other steps in the life-time of an object takes place. The pre-processors are evaluated recursively on *all* descendants of a stick and are boiled down to functions of one argument (the descendant). We supply such a pre-processor by setting the slot `:preproc` to a call to the `preproc` macro with the following syntax:

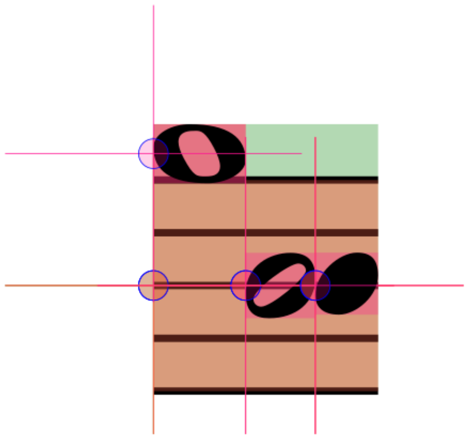
preproc *var* `&body` *cases*

with `var` parameter being the variable name we want to give a descendant object, and which we can refer to in the `cases` of the pre-processor. The `cases` are lists in the shape of `(test form-1 form-2 ... form-N)`. `form-1 ... form-N` are evaluated only when the `test` part evaluates to true. We have thus the possiblity of specifying different pre-processors for the descendants of a composing stick. In the pre-processor below which we supply for the first note, we check whether a child object (here bound to the variable `x`) is of type note-head, and if so we set it's rule-ids and domain so that the object is *responsive* to our rule:

```
(render
 (list
  (stick :horizontal
         :toplevelp t
         :content
         (list
           (stick :stacked
```

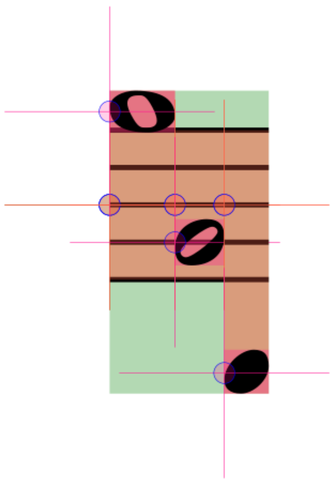
1 We could of course have defined each note-head ourselves (by specifying the `:head` slot for each note object) to circumvent this problem.

```
                  :ruleids '(:id)
                  :content
                  (list (note '(g . 5)
                              :duration 1
                              ;; Specifying a pre-processor to change the
                              ;; rule-ids and the domain of a NOTEHEAD for
                              ;; catching the rule
                              :preproc
                              (preproc x
                                ((typep x 'notehead)
                                 (setf (ruleids x) '(:spn)
                                       (domain x) :treble))))))
           (stick :stacked
                  :ruleids '(:id)
                  :content
                  (list (note '(g . 4)
                              :duration 0.5)))
           (stick :stacked
                  :ruleids '(:id)
                  :content
                  (list (note '(g . 3)
                              :duration 0.25)))))))
```



Since however we want the same pre-processing to be applied to all (implicitely created) note-heads, and since pre-processors are called recursively on all descendants of a stick irrespective of their nesting level, we can supply the same pre-processor to the top most horizontal stick, which saves us supplying the same code for every note object. Another repeated action above, which we can factor out with the aid of a pre-processor is setting the `ruleids` slots of the stacked sticks:

```
(render
(list
 (stick :horizontal
        :preproc (preproc x
                   ;; Recognising pitches in note-heads
                   ((typep x 'notehead)
                    (setf (ruleids x) '(:spn)
                          (domain x) :treble))
                   ;; Drawing staff lines on stacked sticks
                   ;; (will be applied to ANY of the descendants
                   ;; which are STACKED-COMPOSING-STICKs)
                   ((eq (class-name (class-of x)) 'stacked-composing-stick)
                    (setf (ruleids x) '(:id))))
        :toplevelp t
        :content
        (list
          (stick :stacked
                 :content
                 (list (note '(g . 5)
                             :duration 1)))
          (stick :stacked
                 :content
                 (list (note '(g . 4)
                             :duration 0.5)))
          (stick :stacked
                 :content (list (note '(g . 3) :duration 0.25))))))
```



3.2.4 Adding Stems to Noteheads

As a closing example for this chapter, I would like to add some stems to the notes (when they need one).

To accomplish this, and since I want the same stem-drawing rule to be applied to all note objects (for brevity) I will define an id-less rule[1]. In the following definition I set the vertical onsetting point of the stem (the `stem-onset` binding below) to be *approximately* at the center of the note-head[2], although with small modifications (the value of `dy` below) to prevent the note-head ends of the stem being visible beyond the note-head. I also use the *standard* length of a stem which is stored in the variable `*octave-space*`[3]:

```
(defparameter *octave-space* (* 3.5 *staff-space*))
```

The decision about the direction of stem is made by a call to the following function[4]:

```
(defun down-stem-p (spn)
  "Decides about the direction of a stem,
 based on the pitch-name and the octave of spn."
  (let ((pitch-name (car spn))
        (octave (cdr spn)))
    (or (>= octave 5)
        ;; The stem direction of the '(b . 4) should
        ;; be determined based on the stem directions before
        ;; and after in a real solution, hence the whole rule
        ;; for giving stems their directions should be moved to
        ;; a superior container of notes (e.g. a horizontal stick)
        ;; which  has an overview to whole sequence of notes. The following
        ;; exceptional clause for handling the case of center staff-line
        ;; however shall suffice for this fast demonstration!
        (and (eq pitch-name 'b) (= octave 4)))))
```

Eventually the id-less rule itself is defined as follows:

```
(defrule (note) :treble () (n)
  "Draws stem lines on the <correct> side of the note N."
  ;; Give the note object N a stem only when it's duration < whole-note
  (when (< (duration n) 1)
    (let* ((dx .44)
           (dy 1.3)
           (head-top (top (head n)))
           (head-center (* (height (head n)) .5))
           (stem-onset (+ head-top head-center)))
      (packsvg! n
```

1 Id-less rules are defined when we don't specify any rule-id, which makes the rule applicable to all the *target* objects within the rule's domain, independent of any slot values of the object.

2 E. Gould: *Behind Bars*, page 14

3 E. Gould defines the standard stem length to be eqaul to 3.5 staff spaces. (*Behind Bars*, page 14)

4 Note that this example is meant to serve us in better understanding of the rule-protocol, and is not a real-case solution for generating stems. Stems shouldn't be added as svg-lines to the svg-list of a note, rather they will be objects in their own right which will be created implicitely by notes, and who's direction is calculated also based on their *context* rather than only their octave as the `down-stem-p` function above determines!

```
                (svg:line (if (down-stem-p (spn n))
                              (+ dx (x (head n)))
                              (- (right (head n)) dx)
                              )
                          (if (down-stem-p (spn n))
                              (+ stem-onset dy)
                              stem-onset
                              )
                          (if (down-stem-p (spn n))
                              (+ dx (x (head n)))
                              (- (right (head n)) dx)
                              )
                          (if (down-stem-p (spn n))
                              (+ stem-onset *octave-space*)
                              (- stem-onset *octave-space*)
                              )
                          :stroke-width (- *staff-line-thickness* 20)
                          :stroke-linecap "round"
                          :stroke "black")))))
```

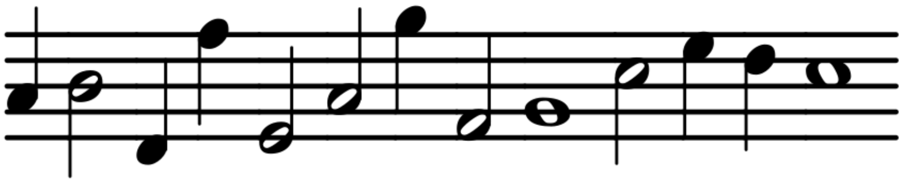
In the rendering below, I have doubled the width of note spaces just to make the overall picture easier to recognize. I also have disabled the drawing of all markers and backgrounds[1]. Note that the second note B4 should get a stem-up with regard to the stem directions of it's two neighbore notes, which we haven not considered in our example.

```
(render (list (stick :horizontal
                     :ruleids '(:content)
                     :chase-visible-p nil
                     :marker-visible-p nil
                     :toplevelp t
                     :content
                     (loop
                       for dur in '(.25 .5 .25 .25 .5 .5 .25 .5 1 .5 .25 .25 1)
                       for pitch in '(a b d f e a g f g c e d c)
                       for oct in '(4 4 4 5 4 4 5 4 4 5 5 5 5)
                       collect (stick :stacked
                                      :content (list (note (cons pitch oct)
                                                           :duration dur))))
                     :preproc (preproc x
                                ((typep x 'notehead)
                                 (setf (ruleids x) '(:spn)
                                       (domain x) :treble
                                       (chase-visible-p x) nil
                                       (marker-visible-p x) nil))
                                ((typep x 'note)
                                 (setf
```

1 In a release these *helper* graphics of course are by default set to NIL.

```
                                   ;; Doubling the width temporarily to ease
                                   ;; reading
                                   (width x) (* (width x) 2)
                                   (domain x) :treble
                                   (chase-visible-p x) nil
                                   (marker-visible-p x) nil))
                                ((typep x 'stacked-composing-stick)
                                 (setf (chase-visible-p x) nil
                                       (marker-visible-p x) nil))
                                ))))
```

The code snippet above renders the final example of this writing:



Closure

In this report I outlined the hitherto progress in the development of the SMT engine. The work on SMT has happened in compliance with a desire for a more flexible and extendible, yet fully functional music typesetting system compared to extant softwares. Hence, SMT will come with a full set of basic functionality which will satisfy the music typesetting tasks as musicians would excpect and know from other music notation systems. What makes it different from other systems however, is it's *rule definition protocol* which at the same time forms an important part of it's user-interface. Under the hood, most basic functionalities for doing typesetting tasks has also been defined and implemented in the very same protocol which is at user's disposal. Typesetting rules could then be modified, removed or new rules can easily be introduced to the system in order to enhance or to change the behavior of the system possibly down to it's kernel. Through SMT's modular and extendible design, even conceiving new music notation *semantics* should become a realistic venture!
The behavior and the syntax of the engine is still in rapid development and change. Many new features are conceived to be integrated into the system which were not handled in this report. One of the most important of such aspects for instance is a new syntactic layer which will serve as the actual surface for SMT. While momentarily the only way to work with SMT is by loading it's *engine* package into the Common Lisp session (and hence in Common Lisp's *syntax*), this new enhanced (particularly in musical terms) syntactic layer should make the system usable for the non-Common-Lisper as well. This syntactic layer will be implemented mainly using Common Lisp's read-macros, which should smoothen the transition between the engine and the language layers significantly and thus making a natural integration of Common Lisp code into the SMT language[1] viable. Also since the generated output of SMT is SVG, and there where no Common Lisp packages for handling SVG flexible enough for the work I needed[2], I started SMT's own XML utility system which already includes SVG, and can be expanded as well in the future to encompass other desirable XML-related output formats. Adding support for outputting other formats e.g. MEI, MusicXML etc. is part of the planned future work.

As for the rule protocol, some major design questions are not decided on yet. For instance the decision about the *order* in which rules should be applied to objects. Also the definite way of *modifying* pre-defined rules is not yet concluded.

I would preferably leave the further documentation of the project to take place over it's

1 E.g. for doing arbitrary computations in Common Lisp and alike, when needed.

2 The only reasonable choice would be the CL-SVG package which turned out to have many restrictions for the tasks of the engine.

documentation page which will be shortly started and updated. The URL of the documentation page is placed at https://symbolicmusictypesetting.github.io/smtdoc/. Until then, I would be glad to answer to questions and comments concerning the current state and further development of the engine at any time via e-mail under the address: *ateymuri63[at]gmail[dot]com.*

Bibliography

[1] Mark McGrain, *Music Notation, Theory and Thechnique for Music Notation*, Berklee Press, 1966

[2] Herbert Chlapik, *Die Praxis des Notengraphikers*, Doblinger, 1987

[3] Elaine Gould, *Behind Bars*, Faber Music, 2011

[4] Kurt Stone, *Music Notation in the Twentieth Century, A Practical Guidebook*, W. W. Norton & Company, Inc., 1980

[5] Jonathan Feist, *Contemporary Music Notation*, Berklee Press, 2017

[6] Karl Hader, *Aus der Werkstatt eines Notenstechers*, Waldheim-Eberle-Verlag, 1948

[7] The LilyPond Development Team, *Essay on Automated Music Engraving*, http://lilypond.org/

[8] Johannes Wolf, *Handbuch der Notationskunde*, Breitkopf & Härtel, 1913

[9] Leland Smith, *SCORE, A Musician's Approach to Computer Music*, 40th Convention of the Audio Engineering Society, Los Angeles, 1971

[10] Perry Roland, Ichiro Fujinaga, Andrew Hankinson: *The Music Encoding Initiative As A Document-Encoding Framework*, 12th International Society for Music Information Retrieval Conference (ISMIR 2011)