L2/16-210R

Title: A system of control characters for Ancient Egyptian hieroglyphic text (updated version)
From: Mark-Jan Nederhof & Vinodh Rajan & Johannes Lang (University of St Andrews, UK), Stéphane Polis & Serge Rosmorduc (Ramses Project, Université de Liege, Belgium & CNAM, Paris), Tonio Sebastian Richter & Ingelore Hafemann & Simon Schweitzer (Thesaurus Linguae Aegyptiae, Berlin-Brandenburgische Akademie der Wissenschaften, Berlin)
To: UTC
Date: 2017-01-25

1 Introduction

This is a revision of L2/16-210, incorporating syntax as anticipated in Section 9 of L2/16-210 and specified formally in L2/16-233, with slight differences. The first version of this document was L2/16-177.

Some arguments from past versions will not be repeated. Those unfamiliar with the background and motivations of this proposal are encouraged to read Sections 2 and 3 of L2/16-177. In particular, our observation in Section 2 thereof remains highly relevant:

> **An encoding scheme for a script only makes the least bit of sense if there is reasonable hope one can encode a text not seen before.**

Apart from the new syntax, used in all examples and described formally in Section A, the following has changed relative to L2/16-210:

- Extended discussion of the motivations for stacking (Section 7.1).
- Introduced case studies (Section 13).
- Description of a new OpenType implementation (Section C).

In final stages of putting this document together, we were informed that thanks to continued efforts by Andrew Glass and the subcommittee, there is now a wish to return to bracketed notation. In Section A.1, we formalize this notation as we understand it and relate it to the notation used elsewhere in this document.

2 The encoding

In this document we present a proposal on the basis of proven concepts taken from various frameworks for encoding hieroglyphic text, most notably PLOTTEXT [12, 13] and RES [6, 7], which shares some primitives with JSesh [10]. The first was used, among other things, to prepare a grammar [4], the second was used, among other things, for the St Andrews corpus [8], and the third in the Ramses Project [9].

Our starting points were:

- The encoding should rely on a small set of primitives, each of which can be precisely defined in a self-contained manner, in terms of relatively simple geometric principles, without reference to external databases of any kind, nor to any heuristics that might lead to unpredictable behaviour.
- It should be possible to implement the primitives, preferably using off-the-shelf rendering engines, to give satisfactory visual realizations for typical encodings.
- The primitives should be expressive enough to be able to (approximately) reflect relative positions of signs in a wide range of original hieroglyphic texts. Of secondary importance are reproductions of existing type-set editions, as these suffer from limitations of partly outdated printing technology.

Table 1: The most important control characters.

default glyph	short name	character name
	**HOR**	EGYPTIAN HIEROGLYPH HORIZONTAL
	**VERT**	EGYPTIAN HIEROGLYPH VERTICAL
	**KERN**	EGYPTIAN HIEROGLYPH KERN
	**INSERT_T_L**	EGYPTIAN HIEROGLYPH INSERT TOP LEFT
	**INSERT_B_L**	EGYPTIAN HIEROGLYPH INSERT BOTTOM LEFT
	**INSERT_T_R**	EGYPTIAN HIEROGLYPH INSERT TOP RIGHT
	**INSERT_B_R**	EGYPTIAN HIEROGLYPH INSERT BOTTOM RIGHT
	**INSERT_CENTER**	EGYPTIAN HIEROGLYPH INSERT CENTER
	**STACK**	EGYPTIAN HIEROGLYPH STACK
	**EMPTY**	EGYPTIAN HIEROGLYPH EMPTY

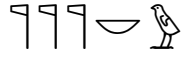
- The primitives do *not* specify exact distances between signs nor exact scaling factors.
- The functionality of the encoding should be extensible by formats outside the realm of Unicode, to allow more precise specification of positioning and scaling. However, neither by Unicode nor by the extended formats do we intend to achieve quasi-facsimiles of original texts.

A powerful encoding scheme not only relieves font developers of the permanent and unreasonable burden of having to update fonts ad nauseum with new spatial arrangements of signs, it will also avoid proliferation of the sign list by unnecessary composite signs, which would place a permanent and unreasonable burden on Unicode itself to provide frequent updates, as well as a collective burden on the Egyptological community to provide suggestions for such updates.

The main control characters introduced in this proposal are listed in Table 1, and will be motivated step by step in the following sections, where we will use abbreviated names for these characters. Code points will be assigned in a follow-up document. Implicit in the table is that there are in fact three copies of the **HOR** and **VERT** primitives, for different levels of operator precedence, and similarly two copies of each of the insertion primitives. This will become clearer shortly.

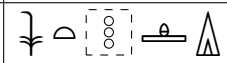
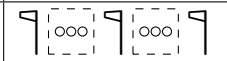
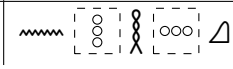
3 Linear text

In the simplest case, hieroglyphic text can consist of a series of signs one after the other, in a horizontal row. For this, no control characters are needed. The signs are separated by a default, font-defined, inter-sign distance. An example is:

Appearance	Unicode	PLOTTEXT	RES
 [a]		R8 R8 R8 V30 G43	R8-R8-R8-V30-G43

[a] BM EA 584 [2, p. 122]

Table 2: Groups.

Appearance	Unicode	PLOTTEXT	RES
 [a]		M23/X1,R4/X8	M23-X1:R4-X8
 [b]		"R8 R8 R8"	R8*R8*R8
 [c]		V30,U23 D58/ N26,O49	V30:U23*D58-N26:O49
 [d]		N35,V28 "N29,D21"	N35:V28*(N29:D21)

[a] BM EA 571 [2, p. 77]
[b] BM EA 585 [2, p. 48]
[c] BM EA 587 [2, p. 46]
[d] BM EA 1783 [2, p. 74]

Note that the fourth sign from the left,  , is less high than the height of a line, and is therefore vertically centered by default.

As we will see later, a sign may be scaled down when it is combined with other signs in a group. The size of a sign before it is scaled down will be called its *natural size*. The natural size is measured in terms of the height of the unscaled 'sitting man' sign  , which is called the *unit size*. In the above example, the natural height of  is 1.0, while that of  may be closer to 0.4 (in our font). Often, but not always, the height of a line is the same as the unit size.

In the example above, the text is written from left to right. Original hieroglyphic texts, however, are often written from right to left. The signs then appear mirrored, with the hieroglyphs representing living entities facing right. Text may also be written in vertical columns, either left-to-right or right-to-left. We will assume most rendering engines can only realize (hieroglyphic) text horizontally from left to right, but see Section 10.2 for further discussion.

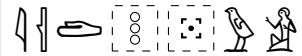
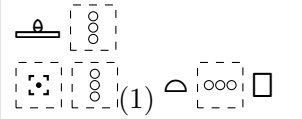
4 Groups

For encoding complex groups of signs, we need to be able to compose signs horizontally and vertically, at the very least. This may require repeated composition. An example is the group  , which is a vertical arrangement of two groups, of which the bottom one,  , is a horizontal arrangement of two groups, of which the second,  , is a vertical arrangement of two signs.

We will use control character **HOR** with default glyph  between signs or groups to be arranged horizontally, and **VERT** with default glyph  between signs or groups to be arranged vertically. The former binds more tightly. To be able to represent deeper structures, we add subscripts (1) or (2) to these control characters for nested horizontal or vertical groupings; see further Section 9. Examples are listed in Table 2.

The simplest case of grouping is if we have a horizontal arrangement of signs in horizontal text, as in

Table 3: Empty signs.

Appearance	Unicode	PLOTTEXT	RES
 [a]		M17/Aa28/"D46,"/G43/A1	M17-Aa28-D46:.-G43-A1
 [b]		R4,",X1" Q3	R4:(.:X1)*Q3

[a] BM EA 584 [2, p. 122]
[b] BM EA 581 [2, p. 59]

 . The intended rendering of this is very similar to what we would get without grouping, except that line breaks are disallowed within a group.

5 Empty

It is convenient to have an **EMPTY** sign  with zero width and height. Placing this sign above or below another sign effectively pushes the latter sign down or up, respectively. Examples are listed in Table 3.

We leave open the question whether **EMPTY** can be an existing empty character from Unicode, or whether a dedicated character for hieroglyphic encoding is appropriate. We suspect that if we do not explicitly introduce the possibility of using **EMPTY** in hieroglyphic encoding here, then the use will sneak in some time in the future, using this or using another existing empty character, as encoders will frequently want to flush the position of a sign to the top or to the bottom of a line, to describe the appearance on original manuscripts.

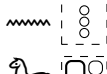
6 Insertion

A fair number of signs have empty space in one of the corners of their bounding box. Often this empty space is used for placement of a smaller sign, especially, but not exclusively, if the two signs are in a special relationship, for example, if the two signs together are the writing of (a part of) a morpheme or a direct genitive. The empty space may also be occupied by several signs. Our encoding includes a number of primitives for such a composition of signs.

In our current syntax, the main sign, in which smaller signs are inserted, comes first. This is followed by one of four 'corner insertion' control characters and then the sign or group to be inserted. In fact, the main sign may be followed by more than one such insertion, in the extreme case by all four, which must appear in a canonical order.

There is also evidence suggesting use of a 'center insertion' of one sign inside another. Here the main sign is followed by  and then the sign or group to be inserted. Such insertion is particularly common in the writing of *wꜥbt* ('priestess', 'pure thing', etc.) as  ; the inserted sign is the feminine ending, and an analysis of the group as an atomic sign would be highly unsatisfactory. In groups such as  moreover, an encoding using vertical arrangement and kerning (Section 8) is inappropriate, as this would not express

Table 4: Insertions.

Appearance	Unicode	PLOTTEXT	RES
 [a]		D60;/X1/;	insert(D60,X1)
		F4;X1//;	insert[ts](F4,X1)
 [b]		F20;Z1/;	insert[b](F20,Z1)
 [c]		I10;S29/;	insert[b](I10,S29)
 [d]		I10;D46/;	insert[s](I10,D46)
 [e]	 (1)	I10;X1,N17;	insert[bs](I10,X1:N17)
 [f]		D17;/X1;	insert[t](D17,X1)
 [g]		G39;/N5;	insert[te](G39,N5)
		G17;/%B4+D36;	insert[te](G17*.,D36)
 [h]		A17;//X1;	insert[be](A17,X1)
		G39;X1/;;/N21;	insert[te](insert[s](G39,X1),N21)
 [i]	 	N35,X1, "V28 E6;/X1;"	N35:X1:V28*insert[te](E6,X1)

[a] BM EA 143 [2, p. 110], Meir I, pl. 9 [2, p. 45]
[b] BM EA 581 [2, p. 59]
[c] BM EA 1783 [2, p. 74]
[d] BM EA 581 [2, p. 59]
[e] BM EA 101 [2, p. 58]
[f] URK IV,373,12
[g] BM EA 117 [2, p. 31]
[h] P. Turin Cat. 2070
[i] Karnak (KRI II,226,6)

that the inserted group is to be entirely inside the bigger sign.

Examples are listed in Table 4. In some cases, the inserted sign or group fits entirely within the bounding box of the main sign, possibly after scaling it down. There are cases however where the inserted sign or group may spill over to outside the bounding box, as in the case of  . It is for the font to decide whether the inserted group is small enough to fit within the bounding box, possibly after some scaling down, or whether it should extend to beyond the bounding box. In PLOTTEXT and RES, additional control characters would be required to artificially extend the bounding box, whereas in our Unicode encoding, we sacrifice precision for ease of use.

The inserted groups may be arbitrarily complex. For example in  we have a vertical group within a

Table 5: Stacking.

Appearance	Unicode	PLOTTEXT	RES
 [a]		P6=D36	stack(P6,D36)
 [b]		U34=I9	stack(U34,I9)
 [c]		P6=V12	stack(P6,V12)

[a] BM EA 581 [2, p. 59]
[b] BM EA 584 [2, p. 122]
[c] [11, p. 758]

horizontal group within a vertical group, which is inserted into another sign.[1] Note that the inserted group here spills over to below the bounding box.

Relative to PLOTTEXT, our notation is simplified in that 'insert just above the feet of a bird' and 'insert into the lower-left corner' have been merged. This sacrifice of precision for simplicity seems justifiable. Relative to RES, our notation is simplified in that we have only five insertion primitives, rather than nine. The functionality of the extra four in RES, i.e. 'insert into the right/left/bottom/top side', can partly be taken over by the **KERN**, to be discussed in Section 8. Further, the appearances of for example  and  , encoded using 'insert into the left side' and 'insert into the bottom-left corner' in RES, could be confused without causing significant problems for typical users.

JSesh has two primitives for insertion, each corresponding to one of up to two rectangular *zones* per sign. These primitives are used as G^^^S and S&&&G, respectively, where S is a sign and G is a group. In the first case, G is inserted in zone 1 of S and in the second case, G is inserted in zone 2 of S. The zones can be defined in the font or can be computed automatically through heuristics. Typically, zone 1 is at the bottom or in front of a sign and zone 2 is at the top of or behind a sign. The zones may extend beyond the bounding box. Moreover, a zone of a sign is associated with a *gravity*, which indicates towards which of the four sides of the rectangle the inserted group is to be flushed. If a sign has two zones, the two insertions may be combined in the form of G1^^^S&&&G2.

7 Stacking

Sometimes two signs or groups are superimposed. Table 5 presents examples. The first two happen to also exist as individual Unicode characters, while the third is not part of any established sign list as far as we know. There are also many examples of whole groups being stacked, such as  , which is the stacking of horizontal group  and vertical group  . In JSesh, stacking of two signs is expressed using binary operator ##.

It cannot be emphasized enough that stacking is part of how the Ancient Egyptian writing system works. Much like horizontal and vertical grouping and insertion, it was one of the mechanisms the ancient scribes had to their disposal to position signs relative to one another. In other words, stacking is to a large extent productive. The fact that some frameworks in the past (most notably the Manuel de Codage [1]) resorted to introducing separate code points for stacked sign combinations, even for those that are hapax, may be

[1] Stela Cairo, JE 60539, l. 8

Table 6: Stacking as compositional operation.

Stacking	Attested alternatives	Transliteration
 [a]		$\underline{h}^{c}$
 [b]		$\underline{h}^{cc}$
 [c]		$\underline{h}^{c}$
 [d]	 [e]	${}^{c}bb$
 [f]	 [g]	$b\,{}^{c}b^{c}$
 [h]		$b\check{s}$
 [i]	 [j]	$b\underline{t}$

[a] WB III p. 40
[b] Dendera VII, 148.4
[c] Stacked form and non-stacked alternative: WB III p. 40
[d] ASAE 43, p. 254
[e] WB I p. 178
[f] BIFAO 43, p. 118 and WB I p. 447
[g] WB I p. 446
[h] Stacked form and non-stacked alternative: WB I p. 477
[i] MIFAO 16, p. 49
[j] Both non-stacked alternatives: WB I p. 485

blamed on shortcomings of the used technology more than anything else.

Another selection from the many thousands of known stacked signs is given in Figure 6. Some have attested non-stacked alternatives, while for others we cannot immediately verify whether non-stacked alternatives might have existed. In the overwhelming majority of cases, the meaning of the stacked signs is completely compositional. For example, a stacked sign may represent a sequence of phonemes, each of which corresponds to one of the constituent signs.

One may naively object that stacked signs are not entirely compositional, because they not only represent the constituent signs themselves, but also the order in whether these are to be read. This objection is weakened, if not invalidated altogether, by the many known cases where in fact all conceivable orders are valid, as long as an existing word is written. For example,  may be used both in words starting with ${}^{c}b$, where the non-stacked alternative  may be used, and in words starting with $b\,{}^{c}$, where the non-stacked alternative may be used.[2]

7.1 The arguments against stacking refuted

Various objections have been raised against a stacking primitive, and none have held up to scrutiny. We discuss these arguments one by one.

[2] See WB I p. 173-178 and p. 446-450, respectively.

“It would create multiple encodings for the same appearance”

The argument was that there are already single code points in Unicode that represent stacked sign combinations, such as  , which could then be alternatively encoded as the combination of two signs with the stacking primitive.

This was the first objection raised by members of the UTC, and was presented to us as the foremost and definitive argument again a stacking primitive. Surprisingly, this objection was thereafter silently dropped. Perhaps this is because there are many composite signs in Unicode, formed not only by stacking, but also by horizontal and vertical grouping and by insertion. Examples are  , and many dozens more. The implication is that if no control characters may be introduced that create alternative encodings for signs that already exist, then no control characters whatsoever may be introduced, by which encoding of Ancient Egyptian in Unicode is at an impasse.

“We might as well continue adding more code points for stacked signs”

A reminder is in order of the motto stated in the introduction:

> An encoding scheme for a script only makes the least bit of sense if there is reasonable hope one can encode a text not seen before.

An encoding scheme that requires regular additions is not fit for purpose, and needing to use a new code point for each newly discovered stacked sign combination would require regular additions.

“A stacking primitive cannot be implemented in OpenType”

This is patently false. We have been able to implement stacking using OpenType substitution rules. In fact, compared to the difficulties of implementing several levels of horizontal grouping, vertical grouping and insertions, implementing stacking is trivial.

“Stacking achieved using substitution rules would not look perfect”

This is a curious argument, given that we keep being reminded that we should not expect paleographic accuracy from Unicode.

Suppose an encoder of large amounts of hieroglyphic texts has the choice between two scenarios:

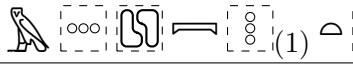
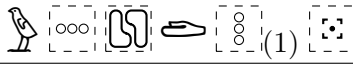
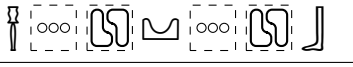
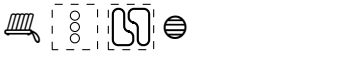
- The next time they find a new stacked sign combination, they need to apply to the Unicode Consortium to have it added, and after a considerable delay, the stacked combination can be encoded and printed in its final form.
- They can encode and print it on the spot, but relative to the ‘ideal’ appearance, the rendering is off by a tiny fraction, which can be corrected if desired by a font designer adding one extra rule to a font.

It requires no further evidence the second scenario is much preferable.

“It is interesting to make lists of stacked sign combinations, which we can put in a big database”

It is far more interesting and useful to encode stacked sign combination as stacked sign combinations, rather than as code points like any other. Corpora encoded in this way contain strictly more information. Concretely, if anyone is interested in a database of stacked sign combinations, it can be extracted automatically from a corpus.

Table 7: Kerning.

Appearance	Unicode	RES
 *a*		G17-[fit]N1:X1:Z1
 *b*		G17-[fit]D21:.
 *c*		G43-[fit]D46:.*O49
 *d*		U23*N26*[fit]D58
 *e*		F39:[fit]Aa1

[a]BM EA 581 [2, p. 59]
[b]BM EA 584 [2, p. 122]
[c]BM EA 585 [2, p. 48]
[d]BM EA 143 [2, p. 110]
[e]BM EA 587 [2, p. 46]

8 Kerning

A natural consequence of the tendency to make efficient and esthetically pleasing use of available space was to squeeze groups together. It would be highly undesirable to have a rendering engine do this indiscriminately for all groups. Therefore, we introduce **KERN**, with default glyph  , which can be put behind the  and operators to indicate that neighbouring signs or groups may, but need not, move towards each other, to have their bounding boxes overlap. The signs should preferably not touch each other however.

Table 7 presents examples. The second example is similar, but not quite identical, to an insertion in the top-right corner. (As a rule, insertions try to scale down inserted groups before they spill over to outside the bounding box, which is not what happens here. Admittedly, there are no absolute criteria when to use **KERN** and when to use insertion.)

9 Operator precedence

As there was opposition to the use of bracketed notation in the earliest versions of this proposal, we made the transition to operators with multiple levels of operator precedence. The multiple levels are needed because there is true recursion in the structure of groups of signs, and Ancient Egyptian may be one of the very few writing systems with this property. Concretely, we can have a vertical group containing a horizontal group containing a vertical group, etc. The deeper the nesting, the more rare such groups become, but we see no obvious reason why a nesting of say 4 or 5 levels deep could not be found if one looked hard enough.

It seem reasonable that some implementations (fonts) can only handle up to a bounded depth, and as Unicode wants to restrict itself to encodings that can be realized in OpenType today, it seems acceptable to choose a bound of say 3 levels, for now. But it is important that the design of the syntax is extendable to more levels, for the sake of future font technology.

To illustrate the levels of operator precedence, consider a group such as  , which suggests that an

Table 8: Enclosures.

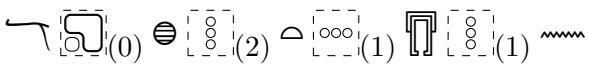
Appearance	Unicode	PLOTTEXT	RES
 [a]		%Z1 N5 L1 D28 %Z2	cartouche(N5-L1-D28)

[a] BM EA 586 [2, p. 25]

operator for vertical grouping should have a lower binding value than that for insertion.[3] An appropriate encoding is therefore:

where the basic level is explicitly indicated by a subscript (0).

But now consider again  . Here we need two more operators for vertical grouping with different binding values, both higher than that of the insertion operator. The encoding is now:



So at the very least, we need vertical grouping at three different levels.

For a precise definition of the syntax involving operator precedence, see Section A.

10 Secondary issues

The following are on our wish list, but should not distract from the main issues.

10.1 Cartouches and other enclosures

It may be noted that the existing hieroglyphs in Unicode include individual symbols for starting and ending parts of several kinds of 'enclosures', but [3] writes "Plain text and general purpose software should likewise treat these signs as characters and not render the fully enclosed form." As this is the only authoritative source we have been able to find that tells us anything at all about intended purpose and use of hieroglyphs in Unicode, we infer we cannot reuse these characters for representing actual, full-form cartouches, serekhs, etc.

Furthermore, cartouches in hieratic are written as two isolated starting and ending parts. It seems natural to reserve  and  for representing these, and to introduce additional primitives to denote full-form cartouches. These primitives may differ between one protocol and another, so it is a pair `<` and `>` in JSesh, a pair `\%Z1` and `\%Z2` in PLOTTEXT, and `cartouche( )` in RES,

For Unicode one could consider an open cartouche sign  (or open serekh or open 'castle walls'), followed by a number of groups and closed by some fixed closing sign  . An example is given in Table 8. If such a bracketed notation is acceptable, one may wonder why the bracketed notation we proposed originally for horizontal and vertical groups was not.

One could also consider encoding an enclosure by a special use of the **INSERT_CENTER**  , consisting of a sign for a cartouche (there is such a sign in the existing Unicode repertoire already, but not for

[3] BM EA 581.

serekh or 'castle walls'), followed by **INSERT_CENTER**, followed by a horizontal group (in horizontal text) or a vertical group (in vertical text). The downside is that automatic conversion from vertical text to horizontal text or vice versa becomes more difficult, as in vertical text a cartouche can be unboundedly high, whereas in horizontal text a cartouche can be unboundedly wide. A restructuring between vertical and horizontal grouping would therefore be required for the conversion to give satisfactory results. The question of how to best encode cartouches is thereby connected to Section 10.2.

10.2 Changes in text direction

In the simplest case, hieroglyphic text consists of a sequence of signs that are reasonably wide and high. The signs can then be placed next to one another for horizontal text directions and underneath one another for vertical text directions. However, two tall narrow signs would typically be put next to one another and two wide thin signs would typically be put above one another. Note however that top-level horizontal groups are strictly speaking redundant in our encoding of horizontal text, and so are top-level vertical groups in vertical text, and as a result we may miss appropriate groupings if we convert between text directions.

These problems are partially avoided in PLOTTEXT by allowing the user to omit groupings of signs, leaving it to the application to find suitable groupings based on the dimensions of the individual signs. In this proposal, we have assumed that we do need to specify groupings in the encoding, relieving the font and rendering engines from this difficult task. However, we wish to keep open the possibility that adequate rearrangements of groups are made automatically by the application in case it imposes a change of text direction relative to the original manuscript. This however requires that the original text direction is or can be encoded.

Some examples are given in Table 9. In the first, the vertical text is best changed to horizontal text by simply stringing signs together horizontally. In the second example however, a more pleasing appearance is achieved by introducing some vertical groups. In the third example, there is kerning between the two signs that was meant to apply to the vertical direction only, but there is no reason to believe kerning would be appropriate for horizontal direction as well so there it should be ignored. In the fourth example, the group is exceptionally high for horizontal text, and becomes quite small if scaled down to fit within a row of height 1, and a thorough restructuring would be desirable.

It would be advantageous therefore to be able to encode text direction. This could be done by, optionally, putting one from the set of four newly introduced characters  ,  ,  or  at the beginning of a fragment of hieroglyphic. This does *not* dictate the direction of rendering, but it informs the application for which direction the encoding is meant.

11 Rendering

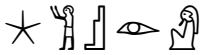
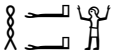
Here we discuss the ideal scaling and positioning of signs within groups. Practical implementations may deviate from this ideal due to technical limitations; see Section C.

11.1 Horizontal and vertical groups

What is described here is consistent with both JSesh and RES. Formatting of groups is done in two steps. First, we determine how much signs need to be scaled down (signs are never scaled up) to fit two main constraints. Second, we insert additional whitespace to center and align signs and groups.

For the first step, that of scaling down, we consider inner-most groups before considering enclosing groups. A first constraint is that a vertical or horizontal subgroup within an enclosing group, together with the default inter-sign distance, should not be higher or wider, respectively, than 1 (in terms of the unit size). We illustrate this using Figure 1. Here, the natural size of the signs B and C plus the default inter-sign

Table 9: Change of text direction from vertical to horizontal imposed by the application (all examples from BM EA 101 [2, p. 58])

Original text	Allowable rendering	Better rendering
		
		
		
		

distance add up to a height smaller than 1. Therefore B and C by themselves need not be scaled down. However, they form a horizontal group with A (which is enclosed in another vertical group), of which the natural width exceeds 1. Therefore, A, B and C are all scaled down uniformly, to make that width exactly 1. Similarly, D, E and F need to be scaled down to make their added width exactly 1. A second constraint is that a group within a line of horizontal text does not exceed the height of that line, which is normally 1. This may require further uniform scaling down of all signs and their inter-sign distances.

If we have a group with a similar structure but with signs of different sizes, the following would happen, with w for the natural width, h for the natural height, and *sep* for the default inter-sign distance.

- If $h(B) + sep + h(C) > 1$, then determine scaling f_1 such that $f_1 \cdot (h(B) + sep + h(C)) = 1$; otherwise let $f_1 = 1$.
- If $w(A) + sep + \max(f_1 \cdot w(B), f_1 \cdot w(C)) > 1$, then determine f_2 such that $f_2 \cdot (w(A) + sep + \max(f_1 \cdot w(B), f_1 \cdot w(C))) = 1$; otherwise let $f_2 = 1$.
- If $w(D) + sep + w(E) + sep + w(F) > 1$, then determine scaling f_3 such that $f_3 \cdot (w(D) + sep + w(E) + sep + w(F)) = 1$; otherwise let $f_3 = 1$.
- If the text is written horizontally in rows, with line height 1, then in the same vein we compute f_4 to make the whole group fit within the line.

For the second step, we distribute 'excess whitespace' equally over subgroups. We need to distinguish between two cases, namely subgroups consisting of a single sign, and subgroups consisting of several recursive subgroups. In the first case the single sign is centered within the available space, and in the second case, the excess whitespace is divided equally between the subgroups. This is illustrated in Figure 2.

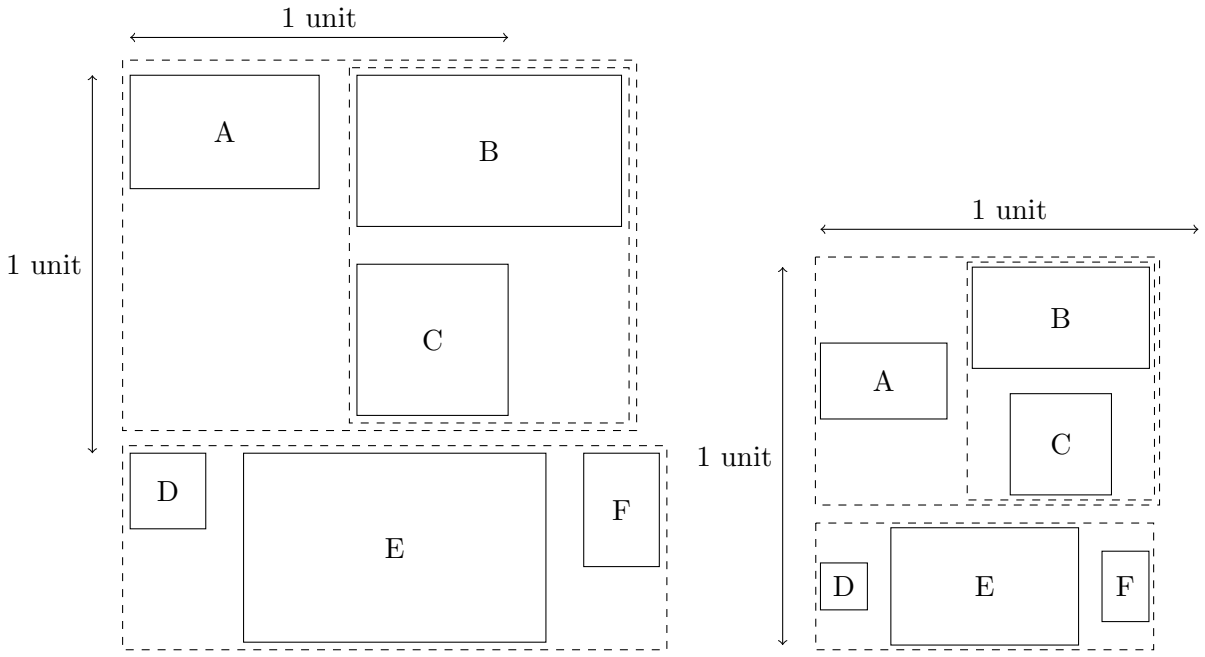


Figure 1: A nested group, before scaling and positioning (left) and after (right).

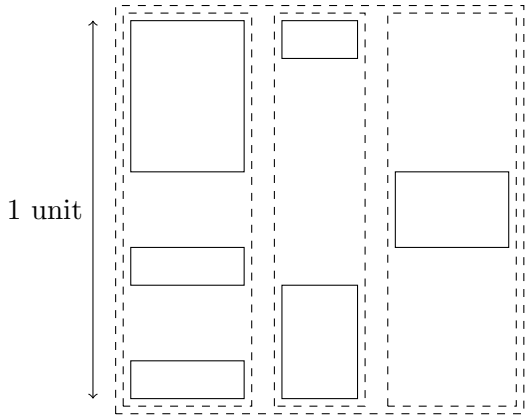


Figure 2: In this horizontal group, there is excess whitespace in all three vertical subgroups. In the rightmost subgroup, there is only a single sign, which is centered. In the leftmost two subgroups, the excess whitespace is divided equally between the (recursive) subgroups (which here happen to be three and two single signs, respectively; they could have been nested horizontal groups as well).

11.2 Insertion

For the implementation of  , the sun sign is placed in the upper right corner of the bounding box of the duck, with the top-most and right-most points of the sun flushed against the bounding box. The sun is ideally as large as possible (but not bigger than the natural size), while keeping some distance (ideally the default inter-sign distance) away from the duck.

A less ideal, but still acceptable, rendering results if we precompile tables indicating for each sign where inserted groups are to be placed. For example, the table might indicate the rectangle as in Figure 3, to

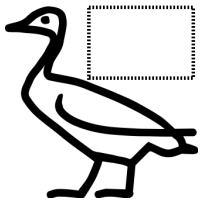


Figure 3: The rectangle that can be designated for the second group after an occurrence of **INSERT_T_R**, if the main sign is the duck.

define how the duck is to be combined with another group if we use **INSERT_T_R**. This is similar to how insertions in JSesh are implemented. Note that this does not place any restrictions on the groups that can be inserted.

11.3 Stacking

The rendering of **STACK** followed by two groups lets (roughly) the centers of the two groups coincide. The rendering is simply the addition of the curves of the two constituent groups.

For some sign combinations, a more satisfactory realization may result if not the exact centers of the groups, but points a little distance away from the centers are chosen to coincide. For example, in the center of coincides with a point a little to the right of the center of . This can be realized by letting a font assign an *anchor* to a sign, which defines a 'conceptual' center, different from the center of the bounding box. (Cf. the notion of *anchor* in OpenType.) It may also be realized in the font through substitutions of entire stacked groups by optimized glyphs.

12 Outside Unicode

Because Unicode has its limitations, extended formats with additional functionality are needed for advanced purposes. It is highly desirable that Unicode and the extended formats can be converted seamlessly one to the other. In particular, converting Unicode to extended formats should be a matter of converting syntax only, and converting extended formats to Unicode should be a matter of systematically removing functionality that is unavailable in Unicode, while retaining an acceptable rendering. We mention RES in particular as a format whose functionality has the proposed Unicode system as a strict subset. We also address absolute positioning and scaling in JSesh.

12.1 Scaling

RES allows tweaking of the natural size of a sign, before the processing of group structure. This can be helpful to improve the rendering, but it is not essential to obtain an acceptable rendering.

Scaling in RES may also be applied on only the height or only the width of a sign. For example, a sign such as may be manually flattened to become , which gives a more satisfactory result if it occurs in a vertical group together with more signs, and a similar distortion of the shape may be witnessed in original inscriptions.[4] The two shapes shown here in fact exist as separate code points in Unicode, which should definitely be avoided for future extensions of the sign list. A font may well include flattened graphical

[4] Cf. pp. 4-5 of [5].

variants, which it may prefer over the original shape depending on context, but there should be only one code point per sign.

The **EMPTY** of this proposal is a special case, with zero width and height, of the 'empty' symbol in RES, which can be parameterized with arbitrary width and height. It is a useful auxiliary symbol to influence placement of neighboring signs.

12.2 Rotation

Some rotation of signs has semantic significance. For example,  is a logogram for the object depicted, namely a mace, while the tilted form  , suggesting a mace being applied, is a determinative in words such as "smite". These two signs have two distinct code points in the existing Unicode set, and this is entirely justifiable.

There are other cases however, such as  ,  ,  ,  , that are pure graphical variants. The choice between any particular inclination and orientation was probably made by a scribe on a whim, partially depending on how well it would fit within the graphical context of other signs. The same holds for a number of thin signs that are used in lying or upright form, such as and  .

The fact that all these are currently separate code points in Unicode is difficult to justify other than by pragmatic considerations: rotation is difficult to realize using off-the-shelf font technology. For this reason we abstain from proposing generic rotation in Unicode at this time. We would make a note however that once font technology has evolved further, introduction of rotation as a primitive in the encoding of hieroglyphic text should definitely be considered.

Of course, RES and JSesh possess primitives for rotation by arbitrary angles.

12.3 Mirroring

Some mirroring of signs has semantic significance. For example, the sign depicting forward walking legs  is used as determinative in words involving (forward) movement, while the mirrored sign  is used as determinative in words involving backward movement. Having separate code points for a sign and its mirrored counterpart is fully justifiable in such cases.

However, some occurrences of mirroring were motivated by other considerations, as for example, symmetry of signs within groups. In such cases, a mirrored sign should be seen as a different graphical realization of the same sign, and does not deserve a separate code point. Moreover, such mirroring is not very common and for many applications the mirroring can be ignored. We therefore will not argue at this time in favour of including generic mirroring in Unicode. Extended formats, such as RES, do include a primitive for mirroring.

12.4 Groups

Rendering of groups can be fine-tuned in several ways. For example, one may override the default inter-sign distance to obtain  rather than . In RES this is done by `R8-[sep=0]R8-[sep=0]R8`. It is definitely not justifiable to have a separate code point for a combination of signs with particular inter-sign distances, even when that distance is 0.

As explained in Section 11.1, the width of a horizontal subgroup is reduced to 1 and the height of a vertical subgroup is reduced to 1, before the scaling of the enclosing group is considered. In some cases, this is undesirable. For example, in  , the top-most two signs should appear with the same scaling as the

bottom sign. This is achieved in RES by changing the above value 1 to something else, or by avoiding prior scaling down of the horizontal group altogether, by using `inf` in `D28*D28:[size=inf]D28`.

In some later periods of Ancient Egyptian history, lines tended to be much higher than 1 unit, and this greatly affects layout of signs, due to the interaction between the natural sizes of signs and the line height. In RES, the line height (and the column width) can be adjusted.

None of the above seem essential for Unicode and will not be proposed at this time.

12.5 Insertion

RES allows fine-tuning of the x and y positions of a group that is inserted into another, as well as fine-tuning of the minimum distance between the two groups. If the distance is chosen to be 0, then the second group may touch the first.

12.6 Stacking

In RES, the stacking primitive can be extended with functionality to let one group erase the underlying curves of the other group. For example, we may obtain `stack[x=0.4](S12, D58)`  , `stack[x=0.4,under](S12, D58)`  , or `stack[x=0.4,on](S12, D58)`  . We know of no examples where this difference in appearance has semantic significance, and would therefore not consider any corresponding primitive for inclusion in Unicode.

As also illustrated by the above examples, RES allows fine-tuning of the relative positions of stacked signs. For Unicode, we rely on the font to choose suitable positioning, as explained in Section 11.3.

12.7 Modify

RES includes a primitive that replaces the physical bounding box by a virtual bounding box. This is a powerful operation that can in rare cases be useful to give complex groups a more pleasing realization than would otherwise be possible. It can also be used to let a part of a sign be rendered outside a line of text. Additionally, one may erase parts of a sign. For many applications however the intricacies of the modify primitive do not outweigh its benefits, and consequently we are not considering adoption of an analogue for Unicode.

12.8 Absolute positioning and scaling

There is a strong demand from the Egyptological community for rendering exact appearances from original texts, mainly for publication purposes. JSesh therefore allows expression of absolute positioning and scaling. For example, `S34\R30{{0,357,51}}**G5{{194,0,97}}` expresses that sign S34 is to be rotated by 30 degrees, scaled by factor 0.51, and placed at (x, y) coordinate (0.0, 0.357), while G5 is scaled by factor 0.97 and placed at coordinate (0.194, 0.0); coordinates refer to the top-left corners of bounding boxes of signs. In this syntax, `**` connects a number of signs together that are formatted by absolute scaling and positioning relative to the same reference point (0.0, 0.0). If the triple is absent, it defaults to `{{0,0,100}}`.

The disadvantage of absolute scaling and positioning is that it makes the encoding dependent on the exact shapes and natural sizes of signs, in other words on the font, which complicates exchange of encodings between tools. Absolute scaling and positioning is therefore best avoided, unless it is essential to the application.

12.9 Shading

In our domain it is the rule rather than the exception that manuscripts are only partially legible, as the passing of several millennia has left few textual artefacts completely unscathed. It is important to let an encoding of a text express that identities of certain damaged signs are merely hypothesized, in order to avoid incorrect interpretations. JSesh, PLOTTEXT and RES all allow shading (also called hatching) of signs or parts of signs with varying precision, to indicate damage to the text. RES is the most flexible of the three, allowing the bounding box of a sign or inter-sign distance to be divided into smaller rectangles, through repeated partitioning into two equal parts. For each such smaller rectangle, shading can be individually set.

Our current proposal does not include shading, for two reasons:

- We are unsure about a satisfactory syntax to express shading as part of plain-text encoding.
- We anticipate that the UTC will want to delegate shading to a higher-level protocol.

However, because shading is very important to our domain, we intend to revisit this matter in the near future.

13 Case studies

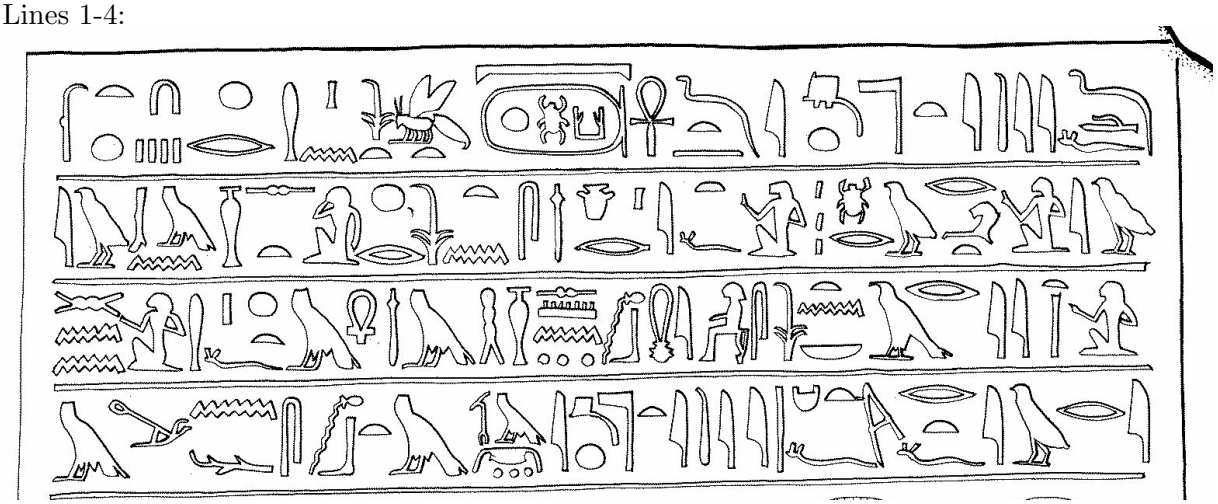
On the following pages we discuss excerpts from two Middle-Egyptian hieroglyphic texts (both from the 12th dynasty). The main purpose is to demonstrate to the reader that encoding of hieroglyphic texts is far from straightforward. This is because encoding requires use of discrete signs and discrete control characters, whereas in reality the placement of signs, their scaling, and even the shapes of individual sign occurrences were more fluid, motivated by esthetical concerns that are hard to capture in simple terms.

For highly normalized scripts, one may well argue that each inscription should allow for precisely one encoding. For Ancient Egyptian however, this position is untenable, as there is no one-to-one mapping between relative positionings of signs in original inscriptions and control characters. Hence, there are often alternative ways of analyzing the observed spatial layout, which may lead to different encodings. However, we hope the examples will at least convince the reader that the repertoire of primitives of this proposal is sufficiently powerful to create encodings that capture most of the essence of the spatial layout found in real texts.

We present excerpts from line drawings that are reproduced with kind permission from Richard Parkinson and ©Trustees of the British Museum. For the convenience of the reader, the excerpts are printed mirrored (the originals read right to left rather than left to right), so that the text direction matches that of the given normalized transcriptions. More information on the stelae is available in easily accessible form in [2].

13.1 Stela of Ity and Iuri (BM EA586)

Lines 1-4:

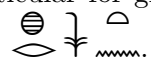


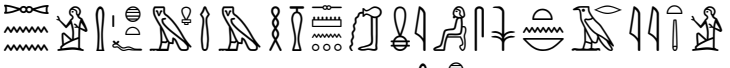
(1) 

The sky sign above a cartouche is unusual. A suitable encoding scheme for hieroglyphic text should be flexible enough to deal with the unexpected.

In this line we see two typical insertions of groups into the bottom-left corner of the cobra. Note in the original how the two cobras are shaped slightly differently, to nicely bend around the inserted groups, depending on the shapes of the inserted groups.

(2) 

In the original we see typical 'squeezing together' of several groups to make efficient use of the available space. It may not be a coincidence that this happens in particular for groups of signs that together form writings of words or of frozen expressions such as  and  . We have here not attempted to express this in the encoding, but one may consider using the kerning operation.

(3) 

The most plausible encoding seems  . What is interesting is that the stroke belongs to the preceding , and so does the viper (as suffix pronoun). The reading order is in fact . This means that however one wishes to analyze the layout of the group, the reading order is not straightforwardly reflected in the structure of the encoding.

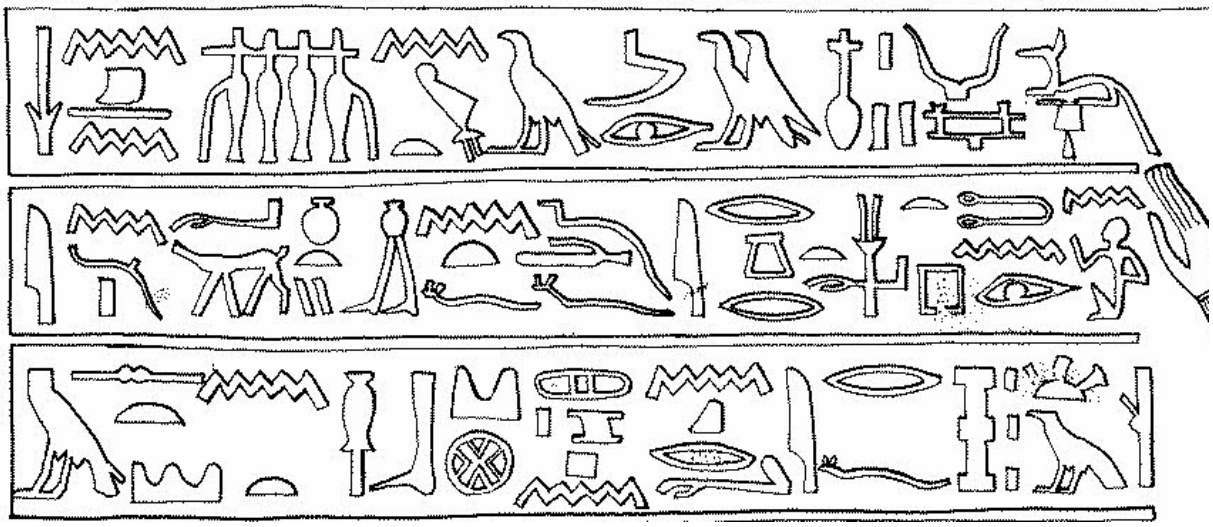
Should one encode  , with an insertion in the top-right corner, or perhaps  , as a horizontal group with or without kerning? The two signs belong to the writing of the same word, so perhaps the former is more appropriate. This is despite the fact that more than half of the smaller sign is outside the bounding box of the main sign. One may ask the same question for  , which similarly lacks a straightforward answer. All this shows that, like for other graphemic systems, coherence in the encoding can only be achieved by adopting general normalizing principles, here regarding the uses of the control characters.

(4)  and 

In this line, we see several more groups that are squeezed together. Some of this is expressed in the transcription, throughout using kerning rather than insertions.

13.2 Stela of Intef son of Senet (BM EA581)

Lines 1-3:



(1) 

The rotation of  was justifiably ignored. One may argue whether to encode the three vertical strokes as two underneath each other followed by a third, or as one above two others.

(2) 

A clear insertion occurs in  . From the appearance in the original, it is not so clear whether to encode  or  , so an insertion of one sign, with another sign below, or an insertion of a vertical group.

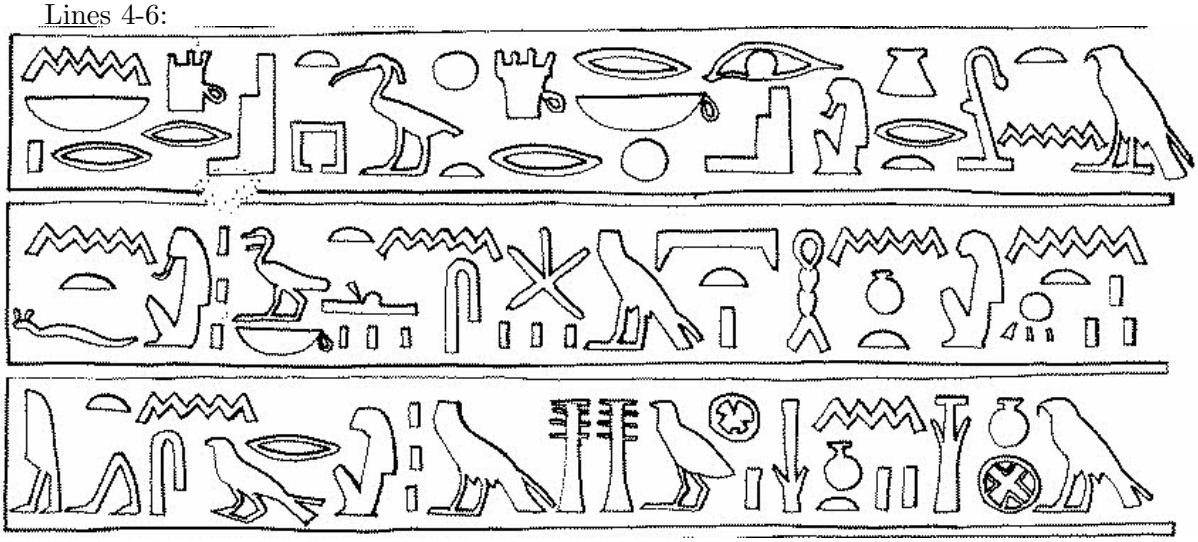
Should  be seen as a stacking (the stacked sign in fact exists as character in Unicode) with an insertion of  in the upper-left corner? Note that in fact belongs to the preceding word, and therefore the insertion is completely accidental, through opportunistic use of available space, rather than a 'ligature' taken from any imaginary finite list of such sign combinations. As alternatives one could encode  or  (the latter using an empty sign below to push the up), possibly with kerning.

(3) 

In the encoding we have disentangled the first 4 signs into two groups, with the next 2 signs belonging to a third group, and have not used any kerning to try to mimic the original appearance.

In the original, the signs  and are squeezed towards one another, as these two signs in this combination typically are. Some form of kerning could also express the relative positioning of  and  .

Lines 4-6:



(4) 

In  we see a classical case of two insertions into a bird sign. One may argue whether to encode  or  . In various places in this line we see groups that are squeezed together, and one would be tempted to resort to kerning to let the transcription resemble the original appearance more faithfully. The sign  occurs in a different graphical variant in the original.

(5) 

In  we see a vertical group within a horizontal group within a vertical group. If one prefers to read the three strokes as three separate signs (there is a character for the three strokes together), we even obtain four levels of nesting. Another example of three or four levels is  . Here once more, it is confirmed that deep nesting of groups was nothing out of the ordinary in classical Middle Egyptian inscriptions.

(6) 

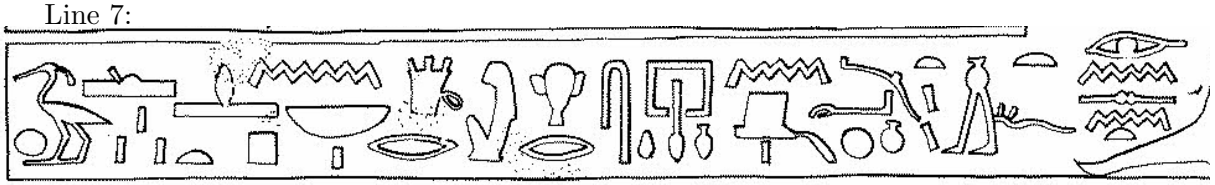
The transcription  seems acceptable, but looks rather unnatural, and is much unlike the original. Kerning might help. An alternative encoding might be  .

Given the appearance of the original, one may argue whether to encode  , with two insertions into the chick, or  , with only one insertion, and a separate stroke following it (which could be pushed down using the empty sign).

In  we find one more example of a vertical group within a horizontal group within a vertical group.

The  preceding the  in line (4) is a mistake and so is the final  in line (6); our encoding gives the signs that were carved rather than the ones that were intended.

Line 7:



(7) 

The first group is a classical insertion into the bottom-left corner of a bird. Note that in line (4), the same sign is inserted into the same bird, but there it was in the top-right corner. This once more confirms the importance of an explicit encoding of the positions of inserted signs and groups, rather than relying on defaults that may give the desired result only some of the time.

We would prefer to see  as vertical group, possibly with kerning, rather than as an insertion. This is despite the original appearance, which has the stroke well within the bounding box of the bigger sign.

In  we see a group of three signs inserted into a fourth sign. In the original the inserted group extends below the bounding box of the bigger sign. One may ask why one would not encode instead the insertion of only one sign (the arm), with two more signs underneath, which would roughly match the physical appearance of the original. Our justification for the insertion of the entire group of three signs is that these signs belong closely together, as part of the writing of a word, together with the following  and the two oblique strokes.

In the encoding we have used the character  for two oblique strokes, because there is no Unicode character for its graphical variant with the two strokes underneath each other as it appears in the original.

The last group consists of five signs arranged vertically, just above the foot of a figure drawn to the right of the text.

14 Conclusions

The Ancient Egyptian writing system is not simple by any stretch of the imagination. A satisfactory encoding will therefore necessarily involve a few non-trivial elements. Moreover, Ancient Egyptian writing is very different from other writing systems, which makes implementation using existing rendering engines challenging. Nonetheless, proof-of-concept exists for critical parts of the proposed encoding.

References

[1] J. Buurman, N. Grimal, M. Hainsworth, J. Hallof, and D. van der Plas. *Inventaire des signes hiéroglyphiques en vue de leur saisie informatique.* Institut de France, Paris, 1988.

[2] M. Collier and B. Manley. *How to read Egyptian hieroglyphs: A step-by-step guide to teach yourself.* British Museum Press, 1998.

[3] M. Everson and B. Richmond. Proposal to encode Egyptian hieroglyphs in the SMP of the UCS. Working Group Document ISO/IEC JTC1/SC2/WG2 N3237, International Organization for Standardization, 2007.

[4] E. Graefe. *Mittelägyptische Grammatik für Anfänger.* Harrassowitz Verlag, Wiesbaden, 1994.

[5] J. Moje. *Untersuchungen zur Hieroglyphischen Paläographie und Klassifizierung der Privatstelen der 19. Dynastie*. Harrassowitz Verlag, 2007.

[6] M.-J. Nederhof. A revised encoding scheme for hieroglyphic. In *Proceedings of the 14th Table Ronde Informatique et Égyptologie*, July 2002. On CD-ROM.

[7] M.-J. Nederhof. The Manuel de Codage encoding of hieroglyphs impedes development of corpora. In S. Polis and J. Winand, editors, *Texts, Languages & Information Technology in Egyptology*, pages 103–110. Presses Universitaires de Liège, 2013.

[8] M.-J. Nederhof. ORC of handwritten transcriptions of Ancient Egyptian hieroglyphic text. In *Altertumswissenschaften in a Digital Age: Egyptology, Papyrology and beyond*, Leipzig, 2015.

[9] S. Polis, A.-C. Honnay, and J. Winand. Building an annotated corpus of Late Egyptian. In S. Polis and J. Winand, editors, *Texts, Languages & Information Technology in Egyptology*, pages 25–44. Presses Universitaires de Liège, 2013.

[10] S. Rosmorduc. JSesh Hieroglyphic Editor. `http://jseshdoc.qenherkhopeshef.org`, 2016.

[11] K. Sethe. *Urkunden der 18. Dynastie, Volume I.* Hinrichs, Leipzig, 1927.

[12] N. Stief. Hieroglyphen, Koptisch, Umschrift, u.a. – ein Textausgabesystem. *Göttinger Miszellen*, 86:37–44, 1985.

[13] N. Stief. PLOTTEXT. `https://www.hrz.uni-bonn.de/rechner-und-software/pc-anwendungen/textverarbeitung/plottext-1`, 2003.

A Structure of hieroglyphic encoding

For a sequence of signs and control characters to have their intended meanings, it should comply with the following (Backus-Naur) specification. Lower-case non-bold names are classes. Bold-face names represent characters, with upper-case boldface names representing particular characters, and **sign** representing any hieroglyph. The pipe symbol | separates alternatives. Square brackets [ ] indicate optional elements, round brackets followed by an asterisk ( )* indicate repetition zero or more times, and round brackets followed by a plus symbol ( )$^+$ indicate repetition one or more times.

In the below we need classes for different levels of operator precedence. The variable of $\boldsymbol{i}$ can stand for one of the levels **0**, **1** or **2** (more powerful systems could allow for **3** and **4** as well). Different control characters exist for these values. So we have for example the binary operators $\mathbf{HOR_0}$, $\mathbf{HOR_1}$, $\mathbf{HOR_2}$ for horizontal grouping.

The following states that a fragment of hieroglyphic text consists of one or more groups of the lowest level, and that a general group may be a vertical group, a horizontal group, or a basic group.

fragment ::= (group$_{\mathbf{0}}$)$^+$
group$_{\boldsymbol{i}}$ ::= vertical_group$_{\boldsymbol{i}}$ | horizontal_group$_{\boldsymbol{i}}$ | basic_group$_{\boldsymbol{i}}$

The following states that vertical and horizontal groups consist of subgroups separated by the appropriate operator, optionally followed by a kerning modifier **KERN**. A subgroup of a vertical group may not be another vertical group and a subgroup of a horizontal group may not be another horizontal group. If a horizontal group contains a vertical group, the latter must be one level of precedence up.

vertical_group$_i$ ::= vert_subgroup$_i$ (vert_separator$_i$ vert_subgroup$_i$)$^+$
vert_subgroup$_i$ ::= horizontal_group$_i$ | basic_group$_i$
vert_separator$_i$ ::= **VERT**$_i$ [**KERN**]
horizontal_group$_i$ ::= hor_subgroup$_i$ (hor_separator$_i$ hor_subgroup$_i$)$^+$
hor_subgroup$_i$ ::= vertical_group$_{i+1}$ | basic_group$_i$
hor_separator$_i$ ::= **HOR**$_i$ [**KERN**]

A basic group is empty, or it is a core group, followed by optional insertions, with subgroups that must all be one level of precedence up. A core group is a sign by itself, or is a stacking of two subgroups, the first of which is a horizontal arrangement of one or more signs and the second a vertical arrangement of one or more signs.

basic_group$_i$ ::= **EMPTY** |
core_group [**INSERT_T_L**$_i$ group$_{i+1}$] [**INSERT_B_L**$_i$ group$_{i+1}$]
[**INSERT_T_R**$_i$ group$_{i+1}$] [**INSERT_B_R**$_i$ group$_{i+1}$]
[**INSERT_CENTER**$_i$ group$_{i+1}$]
core_group ::= **sign** | stack_hor_subgroup **STACK** stack_vert_subgroup
stack_hor_subgroup ::= **sign (STACKHOR sign)***
stack_vert_subgroup ::= **sign (STACKVERT sign)***

A different way of looking at the above is that for the same level of precedence, the insertion operators bind more tightly than the horizontal operators, which bind more tightly than the vertical operators. Stacking binds more tightly than any other operator of any level.

A.1 A return to bracketed notation ?

An alternative syntax that has been discussed recently replaces each transition to a higher level in operator precedence by a pair of brackets. The formal syntax then becomes as follows:

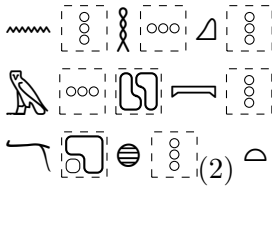
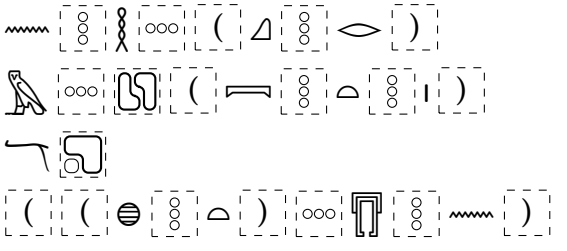
fragment ::= (group)$^+$
group ::= vertical_group | horizontal_group | basic_group

vertical_group ::= vert_subgroup (vert_separator vert_subgroup)$^+$
vert_subgroup ::= horizontal_group | basic_group
vert_separator ::= **VERT** [**KERN**]
horizontal_group ::= hor_subgroup (hor_separator hor_subgroup)$^+$
hor_subgroup ::= **OPEN** vertical_group **CLOSE** | basic_group
hor_separator ::= **HOR** [**KERN**]

basic_group ::= **EMPTY** |
core_group [**INSERT_T_L** nested_group] [**INSERT_B_L** nested_group]
[**INSERT_T_R** nested_group] [**INSERT_B_R** nested_group]
[**INSERT_CENTER** nested_group]
nested_group ::= **OPEN** vertical_group **CLOSE** |
OPEN horizontal_group **CLOSE** |
core_group

where core_group is as before. Note we have here assumed nested_groups do not directly contain nested_groups. We have done this as a quick fix to avoid a situation where a pair of brackets is optional; we understand that in Unicode it is desirable that only one sequence of characters be allowed for each appearance. In hieroglyphic texts, insertions inside insertions are very rare, but they do occur. To refine

Table 10: Notation with several levels of operator precedence versus bracketed notation.

Appearance	Levels of operator precedence	Bracketed notation
		

the syntax to allow this, one could define a nested_group to be a basic_group that is enclosed in a pair of brackets if and only if it is *not* a core_group by itself, so if it is a vertical or horizontal group or if it is a core_group followed by at least one insertion operator. A Backus-Naur specification of this is possible but tedious. An alternative is to always require brackets, which may not be user-friendly.

Table 10 compares the old syntax and the prospective new syntax, with  and  for **OPEN** and **CLOSE**.

B Realization in general-purpose programming languages

The encoding that this document proposes is a functional subset of RES, which has been implemented in C, Java and JavaScript, with the ideal formatting described in Section 11. There is a graphical editor at:

`https://mjn.host.cs.st-andrews.ac.uk/egyptian/res/js/edit.html`

which allows experimentation with the JavaScript implementation. The existence of these implementations shows that the functionality of the proposed encoding can be realized. The differences in syntax are inessential.

C Realization in OpenType

Here we present a revision of our original implementation described in L2/16-177 and L2/16-210, now using additional concepts within OpenType that were suggested to us by Andrew Glass. In the spirit of academic openness we wish to inform interested parties as soon as possible of our progress, not only to announce *that* implementation in OpenType is possible, but also to explain *how*. At the same time, we must stress the preliminary nature of our implementation, as we are aware that a wide range of potential further improvements awaits to be explored, also in the light of a second implementation, by Andrew Glass, which may be reported elsewhere.

Our original implementation was highly restricted, allowing only a few signs and a few groups. The current implementation includes many more signs, potentially to be extended to the entire Unicode set, and allows a wide variety of groups. The basic idea of the design however is the same as before: the signs are interspersed with auxiliary symbols, which are initially 'nil' and subsequently obtain their actual values through two passes:

- The first pass propagates 'bottom-up' the sums of widths and maxima of heights of parts of horizontal groups, and the maxima of widths and sums of heights of parts of vertical groups.

- The second pass propagates 'top-down' the available widths and heights, leading to suitable scalings and positionings, through the ratio of the available dimensions and the unscaled dimensions of subgroups.

Our progress is largely thanks to the introduction of the OpenType lookup flags IgnoreBaseGlyphs and UseMarkFilteringSet. The importance of these flags is that they allow certain information to flow across the input stream without needing to consider symbols that represent unrelated information. For example, while the maximum height is determined of a sequence of signs in a horizontal group, through substitution rules that simulate a 'maximum' operation on neighboring heights, the auxiliary symbols representing e.g. widths can be ignored, as can the signs themselves. This is very different from the original implementation, where all possibilities of intervening auxiliary symbols were enumerated (despite the use of so-called classes), leading to a exponential blow-up in the number of such auxiliary symbols per input sign, in the interpretation of substitution rules.

As in our earlier implementation, OpenType feature files are not created by hand, but generated by a Python script, which manipulates a font through FontForge. This allowed us to make progress relatively quickly. We do not believe actual OpenType feature file notation would be helpful to the reader, and therefore we use an ad hoc notation to formulate our design. Our notation will also abstract away from implementational details, such as the granularity of scaling and positioning values. That is, where we write open or closed intervals, we implicitly assume only a finite number of values in those intervals. For example, $\mathbf{(0, 1]}$ may, depending on context, represent the finite set $\mathbf{\{0.2, 0.4, 0.6, 0.8, 1.0\}}$. In other contexts it may represent $\mathbf{\{1/3, 2/3, 3/3\}}$, etc.

The generation of a font is parameterized by a few constants, which can be changed by changing a single line of the Python script. These determine for example the granularity in the intervals exemplified above. Further constants are the maximum number of signs in a horizontal group, denoted by $\boldsymbol{N_h}$, and the maximum number of subgroups of a vertical group, denoted by $\boldsymbol{N_v}$. (In the present implementation a horizontal group consists only of signs, not of nested vertical groups.)

The minimum and maximum scaling factors are denoted by constants $\boldsymbol{S_{\mathbf{min}}}$ and $\boldsymbol{S_{\mathbf{max}}}$. There is another constant that represents the step of the scaling factor. That could typically be $\boldsymbol{\sqrt{2}}$, so that the allowed scalings are $\boldsymbol{S_{\mathbf{min}} = 2^{0/2} S_{\mathbf{min}}}$, $\boldsymbol{2^{1/2} S_{\mathbf{min}}}$, $\boldsymbol{2^{2/2} S_{\mathbf{min}}}$, $\boldsymbol{\ldots}$, $\boldsymbol{2^{k/2} S_{\mathbf{min}} = S_{\mathbf{max}}}$, for some $\boldsymbol{k}$. This constant may be implicit in notation of intervals such as $\boldsymbol{[S_{\mathbf{min}}, S_{\mathbf{max}}]}$, as explained earlier.

The most important concept in OpenType features is that of substitution rules, which each replace a pattern by another pattern, possibly with a left and right context. For example, A B $\mapsto$ C denotes that two consecutive characters A B should be replaced by C, and the contextual rule C D **{** A $\mapsto$ B **}** E F denotes that A may be replaced by B if it is immediately preceded by C D and immediately followed by E F. That not all rules of this form are allowed in OpenType should not concern us here.

Substitution rules can be told to ignore certain classes of characters. For example, in our notation, the 'code':

With @myclass:
  C **{** A $\mapsto$ B **}**

expresses that A is replaced by B if the first character to the left of A that is in @myclass is C, ignoring intervening characters between $\boldsymbol{C}$ and $\boldsymbol{A}$ that are not in @myclass.

C.1 The auxiliary symbols

For convenience, the auxiliary symbols are arranged into 'records'. There is one record for each horizontal group (hrec) and each vertical group (vrec), as well as one record for each suffix of a horizontal or vertical group (shrec, svrec). Lastly, there is one record for each sign (grec). We will use a notation that supports this organization of auxiliary symbols and records. For example, the auxiliary value hrec.w($\boldsymbol{w}$), for some

number $\boldsymbol{w}$, denotes a field within a record for a horizontal group, meaning that $\boldsymbol{w}$ is the unscaled width of that group.

An hrec consists of the following fields:

- hrec.bw($\boldsymbol{w}$), for $\boldsymbol{w} \in \boldsymbol{(0, 1]}$. The meaning is the same as hrec.w below, but bounded to be maximally 1.
- hrec.w($\boldsymbol{w}$), for $\boldsymbol{w} \in \boldsymbol{(0, N_h]}$. Total unscaled width; sum of widths of subgroups.
- hrec.h($\boldsymbol{h}$), for $\boldsymbol{h} \in \boldsymbol{(0, 1]}$. Total unscaled height, maximum of heights of subgroups.
- hrec.s($\boldsymbol{s}$), for $\boldsymbol{s} \in \boldsymbol{[S_{min}, S_{max}]}$. Scaling factor as determined by ratio of available and unscaled sizes.
- hrec.tx($\boldsymbol{x}$), hrec.ty($\boldsymbol{y}$), for $\boldsymbol{x, y} \in \boldsymbol{[0, 1]}$. Target x-coordinate and y-coordinate of the lower left corner.
- hrec.tw($\boldsymbol{w}$), hrec.th($\boldsymbol{h}$), for $\boldsymbol{w, h} \in \boldsymbol{[0, 1]}$. Target width and height of the group.

An shrec consists of the following fields:

- shrec.w($\boldsymbol{w}$), shrec.h($\boldsymbol{h}$), for $\boldsymbol{w} \in \boldsymbol{[0, N_h - 1]}$, $\boldsymbol{h} \in \boldsymbol{[0, 1]}$.
- shrec.tx($\boldsymbol{x}$), shrec.ty($\boldsymbol{y}$), for $\boldsymbol{x, y} \in \boldsymbol{[0, 1]}$.
- shrec.tw($\boldsymbol{w}$), shrec.th($\boldsymbol{h}$), for $\boldsymbol{w, h} \in \boldsymbol{[0, 1]}$.

The meanings of the fields are similar to those of hrec, but the values only pertain to a suffix of the horizontal group.

Similarly, a vrec consists of:

- vrec.w($\boldsymbol{w}$), for $\boldsymbol{w} \in \boldsymbol{(0, 1]}$. Total unscaled width; maximum of widths of subgroups.
- vrec.bh($\boldsymbol{h}$), for $\boldsymbol{h} \in \boldsymbol{(0, 1]}$. The meaning is the same as vrec.h below, but bounded to be maximally 1.
- vrec.h($\boldsymbol{h}$), for $\boldsymbol{h} \in \boldsymbol{(0, N_v]}$. Total unscaled height, sum of heights of subgroups.
- vrec.s($\boldsymbol{s}$), for $\boldsymbol{s} \in \boldsymbol{[S_{min}, S_{max}]}$. As before.
- vrec.tx($\boldsymbol{x}$), vrec.ty($\boldsymbol{y}$), for $\boldsymbol{x, y} \in \boldsymbol{[0, 1]}$. Target x-coordinate and y-coordinate of the upper left corner.
- vrec.tw($\boldsymbol{w}$), vrec.th($\boldsymbol{h}$), for $\boldsymbol{w, h} \in \boldsymbol{[0, 1]}$. As before.

and an svrec consists of:

- svrec.w($\boldsymbol{w}$), svrec.h($\boldsymbol{h}$), for $\boldsymbol{w} \in \boldsymbol{[0, 1]}$, for $\boldsymbol{h} \in \boldsymbol{[0, N_v - 1]}$.
- svrec.tx($\boldsymbol{x}$), svrec.ty($\boldsymbol{y}$), for $\boldsymbol{x, y} \in \boldsymbol{[0, 1]}$.
- svrec.tw($\boldsymbol{w}$), svrec.th($\boldsymbol{h}$), for $\boldsymbol{w, h} \in \boldsymbol{[0, 1]}$.

The record grec($\boldsymbol{g}$) for a sign $\boldsymbol{g}$ consists of:

- grec.anchor. Intermediate symbol for positioning $\boldsymbol{g}$.
- $\boldsymbol{g}$ itself, later to be replaced by a scaled sign.
- grec.s($\boldsymbol{s}$), for $\boldsymbol{s} \in \boldsymbol{[S_{min}, 1]}$. Overall scaling factor for sign.
- grec.w($\boldsymbol{w}$), grec.h($\boldsymbol{h}$). Unscaled dimensions of $\boldsymbol{g}$.

- grec.sw($\boldsymbol{s}$), for $\boldsymbol{s} \in [\boldsymbol{S_{min}}, \boldsymbol{S_{max}}]$. Scaling factor due to ratio of widths.
- grec.sh($\boldsymbol{s}$), for $\boldsymbol{s} \in [\boldsymbol{S_{min}}, \boldsymbol{1}]$. Scaling factor due to ratio of heights, capped to 1.
- grec.tx($\boldsymbol{x}$), grec.ty($\boldsymbol{y}$), for $\boldsymbol{x}, \boldsymbol{y} \in [\boldsymbol{0}, \boldsymbol{1}]$. Target x-coordinate and y-coordinate of the lower left corner.
- grec.tw($\boldsymbol{w}$), grec.th($\boldsymbol{h}$), for $\boldsymbol{w}, \boldsymbol{h} \in [\boldsymbol{0}, \boldsymbol{1}]$. As before.

As mentioned before, most fields of most records are initially 'nil'. Where we use hrec, shrec, vrec, svrec in substitution rules, this stands for the corresponding sequences of fields with all 'nil' values.

An expression such as @hrec.w stands for the class of all symbols hrec.w($\boldsymbol{w}$) for all allowable values $\boldsymbol{w}$, including 'nil'.

C.2 Initialization of auxiliary symbols

Add brackets and records for signs

Whereas the encoding of our earlier proposals had brackets for horizontal and vertical groups, the UTC wanted us to change to a notation with binary operators with varying operator precedence. As bracket-like delimiters are still needed to interpret the encoding, we need to reconstruct them. Our solution is to first place all brackets around all signs, and then to remove some brackets depending on operator precedence.

For the first step, we want to achieve the result of applying:

> For each sign $\boldsymbol{g}$:
> $\boldsymbol{g} \mapsto$ [(grec($\boldsymbol{g}$))]

To save precious space however from the OpenType tables (limited to 64KB), the same is achieved through several intermediate steps, with auxiliary symbols grec.preaux, grec.postaux, and grec.dim($\boldsymbol{w}$,$\boldsymbol{h}$) for width $\boldsymbol{w}$ and height $\boldsymbol{h}$.

> For each sign $\boldsymbol{g}$ of width $\boldsymbol{w}$ and height $\boldsymbol{h}$:
> $\boldsymbol{g} \mapsto$ grec.preaux $\boldsymbol{g}$ grec.dim($\boldsymbol{w}$,$\boldsymbol{h}$)
>
> grec.preaux $\mapsto$ [(grec.anchor
>
> For $\boldsymbol{w} \in (\boldsymbol{0}, \boldsymbol{1}]$, $\boldsymbol{h} \in (\boldsymbol{0}, \boldsymbol{1}]$:
> grec.dim($\boldsymbol{w}$,$\boldsymbol{h}$) $\mapsto$ grec.s(nil) grec.w($\boldsymbol{w}$) grec.h($\boldsymbol{h}$) grec.postaux
>
> grec.postaux $\mapsto$ grec.sw(nil) grec.sh(nil) grec.tx(nil) grec.ty(nil) grec.tw(nil), grec.th(nil))]

For example, with $*$ short for an operator of horizontal grouping and **:** for an operator of vertical grouping, the input could be "A$*$B$*$C : B$*$C : F" and after the first step, we would have:

> [(grec(A))] $*$ [(grec(B))] $*$ [(grec(C))] : [(grec(D))] $*$ [(grec(E))] : [(grec(F))]

Remove superfluous brackets

To remove the brackets that do not delimit the appropriate groups, we do:

>] $*$ [$\mapsto$ $*$
>] : [$\mapsto$:

and then:

>) $*$ ($\mapsto$ $*$

In the running example we would now have:

> [(grec(A) $*$ grec(B) $*$ grec(C)) : (grec(D) $*$ grec(E)) : (grec(F))]

Add records at brackets and at operators

To prepare for the bottom-up analyses, we insert records between the signs and operators:

(↦ (hrec
∗ ↦ shrec ∗
) ↦ shrec(w=0,h=0))
[↦ [vrec
: ↦ svrec :
] ↦ svrec(w=0,h=0)] adv(nil)

Here shrec(w=0,h=0) and svrec(w=0,h=0) denote suffix records for horizontal and vertical groups, respectively, meant for empty suffixes, with all values being nil, except those for width and height, which are 0.

In the running example we would now have:

[(hrec grec(A) shrec ∗ grec(B) shrec ∗ grec(C) shrec(w=0,h=0)) svrec :
(hrec grec(D) shrec ∗ grec(E) shrec(w=0,h=0)) svrec :
(hrec grec(F) shrec(w=0,h=0)) svrec(w=0,h=0)] adv(nil)

C.3 Right-to-left analysis of horizontal groups

Sum widths from right to left

Here we ignore all glyphs, except those relevant to widths within horizontal groups. We then let the width of a suffix be the sum of the width of its first sign ($\boldsymbol{w_1}$) and the width of the suffix after that ($\boldsymbol{w_2}$). At the beginning of a horizontal group, we need to compute the bounded width (maximally 1):

With @hrec.bw, @hrec.w, @shrec.w, @grec.w:
For $\boldsymbol{n = 0, \ldots, N_h - 2}$:
For $\boldsymbol{w_1 \in (0, 1]}$, $\boldsymbol{w_2 \in [0, n]}$:
{ shrec.w(nil) ↦ shrec.w($\boldsymbol{w_1 + w_2}$) } grec.w($\boldsymbol{w_1}$) shrec.w($\boldsymbol{w_2}$)
For $\boldsymbol{w_1 \in (0, 1]}$, $\boldsymbol{w_2 \in [0, N_h - 1]}$:
{ hrec.w(nil) ↦ hrec.w($\boldsymbol{w_1 + w_2}$) } grec.w($\boldsymbol{w_1}$) shrec.w($\boldsymbol{w_2}$)
For $\boldsymbol{w \in (0, N_h]}$:
{ hrec.bw(nil) ↦ hrec.bw(min(w,1)) } hrec.w($\boldsymbol{w}$)

For the first horizontal group in the running example, we would now have:

(hrec(bw=$\boldsymbol{w'_3}$,w=$\boldsymbol{w_3}$) grec(A,w=$\boldsymbol{w_A}$) shrec(w=$\boldsymbol{w_2}$) ∗
grec(B,w=$\boldsymbol{w_B}$) shrec(w=$\boldsymbol{w_1}$) ∗
grec(C,w=$\boldsymbol{w_C}$) shrec(w=$\boldsymbol{w_0}$))

where $\boldsymbol{w_0 = 0}$, $\boldsymbol{w_1 = w_C}$, $\boldsymbol{w_2 = w_B + w_C}$, $\boldsymbol{w_3 = w_A + w_B + w_C}$, and $\boldsymbol{w'_3 = w_3}$ if $\boldsymbol{w_3 \leq 1}$ and $\boldsymbol{w'_3 = 1}$ otherwise. Here we use ad hoc notation such as grec(A,w=$\boldsymbol{w_A}$) to denote a record for sign A, with field grec.w($\boldsymbol{w_A}$) and values of other fields left unspecified.

Maximize heights from right to left

Here we ignore all glyphs, except those relevant to heights within horizontal groups. The task is now to maximize heights, just as above we summed widths:

With @hrec.h, @shrec.h, @grec.h:
  For $\boldsymbol{n = 0, \ldots, N_h - 2}$:
    For $\boldsymbol{h_1 \in (0, 1]}$, $\boldsymbol{h_2 \in [0, 1]}$:
      { shrec.h(nil) ↦ shrec.h(max($\boldsymbol{h_1}$,$\boldsymbol{h_2}$)) } grec.h($\boldsymbol{h_1}$) shrec.h($\boldsymbol{h_2}$)
  For $\boldsymbol{h_1 \in (0, 1]}$, $\boldsymbol{h_2 \in [0, 1]}$:
    { hrec.h(nil) ↦ hrec.h(max($\boldsymbol{h_1}$,$\boldsymbol{h_2}$)) } grec.h($\boldsymbol{h_1}$) shrec.h($\boldsymbol{h_2}$)

C.4 Right-to-left analysis of vertical groups

Here we analyze vertical groups, similar to how we analyzed horizontal groups, except that we swap addition and maximization for widths and heights. From each horizontal group, the analysis now only needs the values of hrec.bw and hrec.h that were determined earlier. All other glyphs (i.e. signs and auxiliary glyphs) from the horizontal groups are ignored.

Maximize widths from right to left

With @vrec.w, @svrec.w, @hrec.bw:
  For $\boldsymbol{n = 0, \ldots, N_v - 2}$:
    For $\boldsymbol{w_1 \in (0, 1]}$, $\boldsymbol{w_2 \in [0, 1]}$:
      { svrec.w(nil) ↦ svrec.w(max($\boldsymbol{w_1}$,$\boldsymbol{w_2}$)) } hrec.bw($\boldsymbol{w_1}$) svrec.w($\boldsymbol{w_2}$)
  For $\boldsymbol{w_1 \in (0, 1]}$, $\boldsymbol{w_2 \in [0, 1]}$:
    { vrec.w(nil) ↦ vrec.w(max($\boldsymbol{w_1}$,$\boldsymbol{w_2}$)) } hrec.bw($\boldsymbol{w_1}$) svrec.w($\boldsymbol{w_2}$)

Sum heights from right to left

With @vrec.bh, @vrec.h, @svrec.h, @hrec.h:
  For $\boldsymbol{n = 0, \ldots, N_v - 2}$:
    For $\boldsymbol{h_1 \in (0, 1]}$, $\boldsymbol{h_2 \in [0, n]}$:
      { svrec.h(nil) ↦ svrec.h($\boldsymbol{h_1 + h_2}$) } hrec.h($\boldsymbol{h_1}$) svrec.h($\boldsymbol{h_2}$)
  For $\boldsymbol{h_1 \in (0, 1]}$, $\boldsymbol{h_2 \in [0, N_v - 1]}$:
    { vrec.h(nil) ↦ vrec.h($\boldsymbol{h_1 + h_2}$) } hrec.h($\boldsymbol{h_1}$) svrec.h($\boldsymbol{h_2}$)
  For $\boldsymbol{h \in (0, N_v]}$:
    { vrec.bh(nil) ↦ vrec.bh(min(h,1)) } vrec.h($\boldsymbol{h}$)

C.5 Determining global layout of vertical group

Once the width and height of a top-level group have been determined, we can set the target width and height, that is, the dimensions within which the subgroups will be laid out. Where subgroups have natural dimensions that are bigger, one needs to scale them down. Where subgroups have natural dimensions that are smaller, there needs to be padding with whitespace.

First, we will assume the relevant top-level group appears in a horizontal line of text of height 1. Later we will consider horizontal/vertical groups that are placed inside other signs, with an insertion operator.

Determine target x

By default, a group is placed from the current x-coordinate onward.

vrec.tx(nil) ↦ vrec.tx(0)

Determine target y

By default, a group is placed from the top of the line downward, which is here represented by y-coordinate 1.

vrec.ty(nil) $\mapsto$ vrec.ty(1)

Determine target width

By default, the target width of a vertical group is the computed width.

With @vrec.w, @vrec.tw:
  For $\boldsymbol{w} \in \mathbf{(0, 1]}$:
    vrec.w($\boldsymbol{w}$) **{** vrec.tw(nil) $\mapsto$ vrec.tw($\boldsymbol{w}$) **}**

Determine target height

By default, the target height is 1, which concurs with the assumed height of a line.

vrec.th(nil) $\mapsto$ vrec.th(1)

C.6 Left-to-right propagation in vertical groups

Propagate target x-coordinate

The target x-coordinate is propagated to all suffixes of a vertical groups, and from these suffixes to the records of horizontal groups just before:

With @vrec.tx, @svrec.tx:
  For $\boldsymbol{x} \in \mathbf{[0, 1]}$:
    vrec.tx($\boldsymbol{x}$) **{** svrec.tx(nil) $\mapsto$ svrec.tx($\boldsymbol{x}$) **}**
  For $\boldsymbol{n} = \boldsymbol{N_v} - \mathbf{1}, \ldots, \mathbf{1}$:
    For $\boldsymbol{x} \in \mathbf{[0, 1]}$:
      svrec.tx($\boldsymbol{x}$) **{** svrec.tx(nil) $\mapsto$ svrec.tx($\boldsymbol{x}$) **}**

With @hrec.tx, @svrec.tx:
  For $\boldsymbol{x} \in \mathbf{[0, 1]}$:
    { hrec.tx(nil) $\mapsto$ hrec.tx($\boldsymbol{x}$) **}** svrec.tx($\boldsymbol{x}$)

Propagate target width

The target width is propagated in the same way. In addition, we set the advance of the group, copied from the target width.

With @vrec.tw, @svrec.tw:
  For $\boldsymbol{w} \in \mathbf{[0, 1]}$:
    vrec.tw($\boldsymbol{w}$) **{** svrec.tw(nil) $\mapsto$ svrec.tw($\boldsymbol{w}$) **}**
  For $\boldsymbol{n} = \boldsymbol{N_v} - \mathbf{1}, \ldots, \mathbf{1}$:
    For $\boldsymbol{w} \in \mathbf{[0, 1]}$:
      svrec.tw($\boldsymbol{w}$) **{** svrec.tw(nil) $\mapsto$ svrec.tw($\boldsymbol{w}$) **}**

With @hrec.tw, @svrec.w:
  For $\boldsymbol{w} \in \mathbf{[0, 1]}$:
    { hrec.tw(nil) $\mapsto$ hrec.tw($\boldsymbol{w}$) **}** svrec.tw($\boldsymbol{w}$)

With @vrec.tw:
vrec.tw($\boldsymbol{w}$) { adv(nil) $\mapsto$ adv($\boldsymbol{w}$) }

Propagate target y coordinate and height

First, we take the ratio of the target height and the unscaled height of the vertical group. This determines the scaling factor of the first horizontal group. (The division operation is to be suitably rounded off to one from a finite number of available values, which may include values greater than 1, in which case there will be padding with whitespace.)

With @vrec.h, @vrec.th, @hrec.s:
For $\boldsymbol{h_1} \in \boldsymbol{(0, N_v]}$, $\boldsymbol{h_2} \in \boldsymbol{[0, 1]}$:
vrec.h($\boldsymbol{h_1}$) vrec.th($\boldsymbol{h_2}$) { hrec.s(nil) $\mapsto$ hrec.s($\boldsymbol{h_2/h_1}$) }

Subsequently, the target height of the first horizontal group is determined by multiplying the scaling factor with its unscaled height:

With @hrec.h, @hrec.s, @hrec.th:
For $\boldsymbol{h} \in \boldsymbol{(0, 1]}$, $\boldsymbol{s} \in \boldsymbol{[S_{min}, S_{max}]}$:
hrec.h($\boldsymbol{h}$) hrec.s($\boldsymbol{s}$) { hrec.th(nil) $\mapsto$ hrec.th($\boldsymbol{s * h}$) }

By subtracting the target height of the first subgroup, we obtain the target height for the remaining subgroups:

With @vrec.th, @hrec.th, @svrec.th:
For $\boldsymbol{h_1} \in \boldsymbol{[0, 1]}$, $\boldsymbol{h_2} \in \boldsymbol{[0, 1]}$:
vrec.th($\boldsymbol{h_1}$) hrec.th($\boldsymbol{h_2}$) { svrec.th(nil) $\mapsto$ svrec.th($\boldsymbol{h_1 - h_2}$) }

Similarly, by subtracting the target height of the first subgroup from the target y-coordinate, we obtain the target y-coordinate for the remaining subgroups:

With @vrec.ty, @hrec.th, @svrec.ty:
For $\boldsymbol{y} \in \boldsymbol{[0, 1]}$, $\boldsymbol{h} \in \boldsymbol{[0, 1]}$:
vrec.ty($\boldsymbol{y}$) hrec.th($\boldsymbol{h}$) { svrec.ty(nil) $\mapsto$ svrec.ty($\boldsymbol{y - h}$) }

The same needs to be repeated, for every suffix of the vertical group, making sure we do not confuse svrec elements with horizontal subgroups of following vertical groups (which is why we consider @] as a singleton class in the 'With' clauses):

For $\boldsymbol{n = N_v - 1, \ldots, 1}$:
With @svrec.h, @svrec.th, @hrec.s, @]:
For $\boldsymbol{h_1} \in \boldsymbol{(0, n]}$, $\boldsymbol{h_2} \in \boldsymbol{[0, 1]}$:
svrec.h($\boldsymbol{h_1}$) svrec.th($\boldsymbol{h_2}$) { hrec.s(nil) $\mapsto$ hrec.s($\boldsymbol{h_2/h_1}$) }
With @svrec.th, @hrec.h, @hrec.s, @hrec.th, @]:
For $\boldsymbol{h} \in \boldsymbol{(0, 1]}$, $\boldsymbol{s} \in \boldsymbol{[S_{min}, S_{max}]}$:
hrec.h($\boldsymbol{h}$) hrec.s($\boldsymbol{s}$) { hrec.th(nil) $\mapsto$ hrec.th($\boldsymbol{s * h}$) }
With @svrec.th, @hrec.th, @]:
For $\boldsymbol{h_1} \in \boldsymbol{[0, 1]}$, $\boldsymbol{h_2} \in \boldsymbol{[0, 1]}$:
svrec.th($\boldsymbol{h_1}$) hrec.th($\boldsymbol{h_2}$) { svrec.th(nil) $\mapsto$ svrec.th($\boldsymbol{h_1 - h_2}$) }
With @svrec.ty, @hrec.th, @]:
For $\boldsymbol{y} \in \boldsymbol{[0, 1]}$, $\boldsymbol{h} \in \boldsymbol{[0, 1]}$:
svrec.ty($\boldsymbol{y}$) hrec.th($\boldsymbol{h}$) { svrec.ty(nil) $\mapsto$ svrec.ty($\boldsymbol{y - h}$) }

Finally, the target y-coordinate of a subgroup is copied from the following suffix:

With @hrec.ty, @svrec.ty:
  For $\boldsymbol{y} \in [\mathbf{0}, \mathbf{1}]$:
    { hrec.ty(nil) $\mapsto$ hrec.ty($\boldsymbol{y}$) } svrec.ty($\boldsymbol{y}$)

C.7 Left-to-right propagation in horizontal groups

Propagation in horizontal groups is similar to that in vertical groups, but swapping the roles of y-coordinate and height with x-coordinate and width.

Propagate target x-coordinate and width

With @hrec.w, @hrec.tw, @grec.sw:
  For $\boldsymbol{w_1} \in (\mathbf{0}, \boldsymbol{N_h}]$, $\boldsymbol{w_2} \in [\mathbf{0}, \mathbf{1}]$:
    hrec.w($\boldsymbol{w_1}$) hrec.tw($\boldsymbol{w_2}$) { grec.sw(nil) $\mapsto$ grec.sw($\boldsymbol{w_2}/\boldsymbol{w_1}$) }
With @hrec.tw, @grec.w, @grec.sw, @grec.tw:
  For $\boldsymbol{w} \in (\mathbf{0}, \mathbf{1}]$, $\boldsymbol{s} \in [\boldsymbol{S_{min}}, \boldsymbol{S_{max}}]$:
    grec.w($\boldsymbol{w}$) grec.sw($\boldsymbol{s}$) { grec.tw(nil) $\mapsto$ grec.tw($\boldsymbol{s} * \boldsymbol{w}$) }
With @hrec.tw, @grec.tw, @shrec.tw:
  For $\boldsymbol{w_1} \in [\mathbf{0}, \mathbf{1}]$, $\boldsymbol{w_2} \in [\mathbf{0}, \mathbf{1}]$:
    hrec.tw($\boldsymbol{w_1}$) grec.tw($\boldsymbol{w_2}$) { shrec.tw(nil) $\mapsto$ shrec.tw($\boldsymbol{w_1} - \boldsymbol{w_2}$) }
With @hrec.tx, @grec.tw, @shrec.tx:
  For $\boldsymbol{x} \in [\mathbf{0}, \mathbf{1}]$, $\boldsymbol{w} \in [\mathbf{0}, \mathbf{1}]$:
    hrec.tx($\boldsymbol{x}$) grec.tw($\boldsymbol{w}$) { shrec.tx(nil) $\mapsto$ shrec.tx($\boldsymbol{x} + \boldsymbol{w}$) }

For $\boldsymbol{n} = \boldsymbol{N_h} - \mathbf{1}, \ldots, \mathbf{1}$:
  With @shrec.w, @shrec.tw, @grec.sw, @):
    For $\boldsymbol{w_1} \in (\mathbf{0}, \boldsymbol{n}]$, $\boldsymbol{w_2} \in [\mathbf{0}, \mathbf{1}]$:
      shrec.w($\boldsymbol{w_1}$) shrec.tw($\boldsymbol{w_2}$) { grec.sw(nil) $\mapsto$ grec.sw($\boldsymbol{w_2}/\boldsymbol{w_1}$) }
  With @shrec.tw, @grec.w, @grec.sw, @grec.tw, @):
    For $\boldsymbol{w} \in (\mathbf{0}, \mathbf{1}]$, $\boldsymbol{s} \in [\boldsymbol{S_{min}}, \boldsymbol{S_{max}}]$:
      grec.w($\boldsymbol{w}$) grec.sw($\boldsymbol{s}$) { grec.tw(nil) $\mapsto$ grec.tw($\boldsymbol{s} * \boldsymbol{w}$) }
  With @shrec.tw, @grec.tw, @):
    For $\boldsymbol{w_1} \in [\mathbf{0}, \mathbf{1}]$, $\boldsymbol{w_2} \in [\mathbf{0}, \mathbf{1}]$:
      shrec.tw($\boldsymbol{w_1}$) grec.tw($\boldsymbol{w_2}$) { shrec.tw(nil) $\mapsto$ shrec.tw($\boldsymbol{w_1} - \boldsymbol{w_2}$) }
  With @shrec.tx, @grec.tw, @):
    For $\boldsymbol{x} \in [\mathbf{0}, \mathbf{1}]$, $\boldsymbol{w} \in [\mathbf{0}, \mathbf{1}]$:
      shrec.tx($\boldsymbol{x}$) grec.tw($\boldsymbol{w}$) { shrec.tx(nil) $\mapsto$ shrec.tx($\boldsymbol{x} + \boldsymbol{w}$) }

The target x-coordinate of a sign is copied from the preceding hrec or shrec rather than the following shrec, which seems more convenient and natural.

With @hrec.tx, @shrec.tx, @grec.tx:
  For $\boldsymbol{x} \in [\mathbf{0}, \mathbf{1}]$:
    hrec.tx($\boldsymbol{x}$) { grec.tx(nil) $\mapsto$ grec.tx($\boldsymbol{x}$) }
  For $\boldsymbol{x} \in [\mathbf{0}, \mathbf{1}]$:
    shrec.tx($\boldsymbol{x}$) { grec.tx(nil) $\mapsto$ grec.tx($\boldsymbol{x}$) }

Propagate target y coordinate

With @hrec.ty, @shrec.ty:
  For $\boldsymbol{y} \in [\mathbf{0}, \mathbf{1}]$:
    hrec.ty($\boldsymbol{y}$) { shrec.ty(nil) $\mapsto$ shrec.ty($\boldsymbol{y}$) }
  For $\boldsymbol{n} = \boldsymbol{N_h} - \mathbf{1}, \ldots, \mathbf{1}$:
    For $\boldsymbol{y} \in [\mathbf{0}, \mathbf{1}]$:
      shrec.ty($\boldsymbol{y}$) { shrec.ty(nil) $\mapsto$ shrec.ty($\boldsymbol{y}$) }

With @grec.ty, @shrec.ty:
  For $\boldsymbol{y} \in [\mathbf{0}, \mathbf{1}]$:
    { grec.ty(nil) $\mapsto$ grec.ty($\boldsymbol{y}$) } shrec.ty($\boldsymbol{y}$)

Propagate target height

With @hrec.th, @shrec.th:
  For $\boldsymbol{h} \in [\mathbf{0}, \mathbf{1}]$:
    hrec.th($\boldsymbol{h}$) { shrec.th(nil) $\mapsto$ shrec.th($\boldsymbol{h}$) }
  For $\boldsymbol{n} = \boldsymbol{N_h} - \mathbf{1}, \ldots, \mathbf{1}$:
    For $\boldsymbol{h} \in [\mathbf{0}, \mathbf{1}]$:
      shrec.th($\boldsymbol{h}$) { shrec.th(nil) $\mapsto$ shrec.th($\boldsymbol{h}$) }

With @grec.th, @shrec.th:
  For $\boldsymbol{h} \in [\mathbf{0}, \mathbf{1}]$:
    { grec.th(nil) $\mapsto$ grec.th($\boldsymbol{h}$) } shrec.th($\boldsymbol{y}$)

C.8 Final scaling and positioning of signs

Determine vertical scaling factors for signs and combine with horizontal scaling factors

Signs in a horizontal group may need to be scaled down because their natural width or their natural height may exceed the target width or target height. We computed the ratio of the target width and natural width before. That ratio could exceed 1, which was useful for computing padding between signs. Here we need to make the scaling factor maximally 1. Then we need to compute the scaling factor that is the ratio of the target height and the natural height, capped to be maximally 1. The total scaling factor is the minimum of the two scaling factors.

For $\boldsymbol{s} \in [\boldsymbol{S_{min}}, \boldsymbol{S_{max}}]$:
  grec.sw($\boldsymbol{s}$) $\mapsto$ grec.sw(min($\boldsymbol{s}$,1))
With @grec.h, @grec.sh, @grec.th:
  For $\boldsymbol{h_1} \in (\mathbf{0}, \mathbf{1}]$, $\boldsymbol{h_2} \in [\mathbf{0}, \mathbf{1}]$:
    grec.h($\boldsymbol{h_1}$) { grec.sh(nil) $\mapsto$ grec.sh(min($\boldsymbol{h_2}/\boldsymbol{h_1}$,1)) } grec.th($\boldsymbol{h_2}$)
With @grec.s, @grec.sw, @grec.sh:
  For $\boldsymbol{s_1} \in [\boldsymbol{S_{min}}, \mathbf{1}]$, $\boldsymbol{s_2} \in [\boldsymbol{S_{min}}, \mathbf{1}]$:
    { grec.s(nil) $\mapsto$ grec.s(min($\boldsymbol{s_1}$,$\boldsymbol{s_2}$)) } grec.sw($\boldsymbol{s_1}$) grec.sh($\boldsymbol{s_2}$)

Scale signs

An unscaled sign $\boldsymbol{g}$ is replaced by scaled($\boldsymbol{g}$,$\boldsymbol{s}$), which is the same sign scaled down by factor $\boldsymbol{s} < \mathbf{1}$:

With @signs, @grec.s:
  For $\boldsymbol{s} \in [\boldsymbol{S_{min}}, \mathbf{1})$:
    { $\boldsymbol{g} \mapsto$ scaled($\boldsymbol{g}$,$\boldsymbol{s}$) } grec.s($\boldsymbol{s}$)

Position signs

We will not introduce more notation for OpenType positioning rules. Instead we will describe in words how signs are positioned. Recall that at this stage we have, for each sign occurrence, a subsequence consisting of the auxiliary symbol grec.anchor, the scaled sign, and auxiliary symbols of the form grec.x($\boldsymbol{x}$) and grec.y($\boldsymbol{y}$), which determine where the sign should be placed relative to the bottom-left corner of the group. We proceed in two steps. First we position the zero-width, zero-height glyph grec.anchor at $\boldsymbol{(x, y)}$. Next, we position the scaled sign at $\boldsymbol{(0, 0)}$ relative to grec.anchor. The signs in the font are given advance 0, so that this positioning can be done for any number of signs in a group, relative to the same bottom-left corner of the group.

At this point we should mention that the actual sizes of the signs in the font are slightly smaller than the width and height they are given in the substitution rules. This means that appropriate inter-sign space does not need to be manipulated explicitly in our design.

Advance

The auxiliary symbols adv($\boldsymbol{w}$), with $\boldsymbol{w}$ being the computed width of a top-level group, are assigned the advance $\boldsymbol{w}$, so that the next top-level group will be positioned correctly.

C.9 Insertion

Let us now consider how we can combine the above with insertions. Assume for example a sequence of the form A insert_b_l B∗C:D, where insert_b_l, the operator for insertion in the bottom-left corner, has a lower precedence than the operators ∗ and : for horizontal and vertical grouping. The first sign A could for example be the cobra.

Before applying the rules in Section C.5 and following, we would now have (omitting several records and several fields irrelevant to the discussion):

[(A)] adv(nil) insert_b_l [vrec.ty(nil) vrec.tw(nil) vrec.th(nil) (B ∗ C) : (D)] adv(nil)

In order for the inserted group to be positioned inside sign A, we need to set the vrec.ty, vrec.tw and vrec.th values to be something other than the defaults. It suffices to add rules such as:

With @signs, @insert_b_l, @grec.ty:
  A insert_b_l { vrec.ty(nil) ↦ vrec.ty(0.8) }
With @signs, @insert_b_l, @grec.tw:
  A insert_b_l { vrec.tw(nil) ↦ vrec.tw(0.8) }
With @signs, @insert_b_l, @grec.th:
  A insert_b_l { vrec.th(nil) ↦ vrec.th(0.8) }
adv(nil) insert_b_l ↦ insert_b_l

We now have:

[(A)] insert_b_l [vrec.ty(0.8) vrec.tw(0.8) vrec.th(0.8) (B ∗ C) : (D)] adv(nil)

Now the rules from Section C.5 and following function as before except that the inserted group is formatted within the free corner of the cobra. With a small adjustment to the rules presented earlier, the advance is now copied from the width of the cobra, rather than from the width of the inserted group, so that later we would have:

[(A)] insert_b_l [vrec.ty(0.8) vrec.tw(0.8) vrec.th(0.8) (B ∗ C) : (D)] adv(1)

assuming the cobra has width 1. Note we removed the first adv(nil) earlier, to avoid a premature advance.

We have verified that the above design works as required in the implementation. Further refinements can be made however. It is noteworthy that although the insertions are the most innovative element of our encoding, their implementation is simple compared to that of horizontal and vertical grouping.

C.10 Stacking

For stacking two signs, we need to make the following adjustments:

- In the bottom-up phase, the maxima of the widths and heights of the two signs are taken, for the purpose of the analysis of the unscaled widths and heights of enclosing groups.
- In the top-down phase, the stacked signs share their target coordinates and target dimensions.

It is relatively straightforward to extend this to stacking of groups.

C.11 Discussion and outlook

We feel no need to hide our impression that OpenType is the wrong technology to use for a writing system as involved as Ancient Egyptian. The process of designing the presented solution was extremely laborious, and the smallest mistakes in the Python code producing the substitution rules tended to break the functionality completely, making debugging very hard. Moreover, the design may strike the reader as decidedly inelegant. Nonetheless, we come to the conclusion that proof-of-concept has been delivered for realization of the control characters in OpenType, through our implementation, as well as the implementation by Andrew Glass, which may be reported elsewhere.

A few obvious improvements await realization:

- We have investigated groupings consisting of signs within horizontal groups within vertical groups. Having more than two levels of nesting is straightforwardly realized by repeating the same pattern, in each subsequent level switching the roles of addition and maximization. Also our mechanism for 'parsing' of expressions with operator precedence can be extended in the obvious way to several levels of operator precedence, with one pair of brackets for each level, initially adding all brackets around all signs, and then systematically removing brackets on either side of operators that have equal or higher precedence.
- If we have a horizontal group consisting of, say, three identical signs, then our design does not guarantee that each of the three is given the same scaling, due to rounding error, which leads to an ugly rendering. This could be solved by computing only one scaling factor per subgroup, which is propagated to each suffix.
- Similarly, padding is not guaranteed to be uniform between neighboring subgroups. One possible solution, suggested to us by Andrew Glass, is to detach the granularity of the space between subgroups from the granularity of the x-coordinates and y-coordinates, and to position a subgroup relative to its preceding neighbour or to its directly enclosing group, through mark-to-mark positioning.
- We currently do not center scaled signs within the target width and height. Doing so would require few, simple adjustments.

The community developing OpenType implementations could support us in the following ways:

- Creating far more readable and precise documentation, detailing not only the syntax, but also the semantics of the concepts that exist in OpenType feature files.

- Creating benchmarks with (non-standard) use of lookups. We have found particular combinations of lookups of substitution rules that gave three completely different outcomes in three different implementations of OpenType. Mentioned benchmarks could help developers of OpenType implementations to guarantee consistent behaviour.
- In particular, we would wish that multiple calls of the same lookup give predictable results. At present, we avoid calling the same lookup more than once, because of the varying behaviour of different implementations. Regrettably that implies the exact same substitution rules need to be repeated a number of times, wasting precious resources and bringing the 64KB limitation closer than it would otherwise be.