Encoding SignWriting Layout
An Initial Disussion: 2

Martin Hosken,
SIL Non-Roman Script Initiative (NRSI)

Introduction

For those following the progress of thinking, this document differs from the previous in that it removes **triple**, **quad** and **chain**. I realised that a shared attachment point is nothing special, it is just multiple things attached to the same base at the same base_pos. The tuple has also been modified to allow for default base_pos and even the base is optional now. So a tuple can be a single symbol now!

This document is a discussion document. It is not a full Unicode proposal. There is no code chart or codepoints allocated. It is an initial description of a possible encoding model for SignWriting layout. It is presented for review by interested parties. IT IS INCOMPLETE. This means you *will* find problems with it. I am hoping that you will so that a future document may not have such problems. As you review this document may I suggest some questions you might want to ask yourself:

- Are all the attachment points I need listed or are any redundant?
- Is there a sign, used in a sign language, that I can think of that this model won't represent? If so, please let me know of the sign and how you would both write and sign it. I am not good enough at signwriting to correctly read every sign that comes my way.

Due to how different this encoding model is to all the other scripts in Unicode, it should also be born in mind that there is no intention that this encoding be adopted without at least some proof of concept implementation.

SignWriting is a two dimensional script. As such it differs from other scripts in that meaning is expressed both by the symbols used in a sign and their positional relationship to each other. In other scripts, the positional relationship between symbols (or characters) is an inherent attribute of the symbol. In SignWriting, two symbols may be in many different positional relationships and it each of those relationships carries its own meaning.

SignWriting is used to write sign languages which in their turn are writing systems. This is different from other complex layout systems like mathematics, in that whole texts are written in SignWriting rather than just displays of single signs. Such texts need to be searched, stored and processed just like any other plain text. Until now, specialist software has been used for the creation of SignWriting based documents, but these are not Unicode based and are restricted to treating SignWritten texts as graphics. For example: http://ase.wikipedia.wmflabs.org/wiki/Main_Page shows an attempt to produce a wikipedia style site using graphics to display the text. The site is searchable at the symbol level but not for specific signs. The desire to treat SignWriting as any other script with regard to text processing, is clearly evident. And while SignWriting has its uniqueness, it is still worth encoding as plain text, assuming that uniqueness does not introduce insurmountable problems in encoding. This is in direct contrast to things like mathematical equiations, which make no claims to be a writing system or to be representing language or text. Mathematical equations are insertions into text in a language. They do not for the text themselves. As such they can be handled using out of band layout information.

Requirements

There are various requirements that either come from the SignWriting community or from Unicode principles and approaches to encoding. We will use these in designing the encoding.

- Stored symbol order should reflect required symbol order. Thus the encoding should not reorder symbols from their intended stored order. This precludes recursive groupings that may result in sequential symbols being far separated when stored.
- Stored symbol order should reflect the core desire for sorting. The core requirement here is that the dominant hand should be stored before the supporting hand. The reason for this is that there is no way to distinguish the dominant hand from the supporting hand except by their relative stored order. Other supporting symbols such as the head or shoulders can be identified and handled appropriately by processes such as collation.
- The encoding should aim to simplify input as much as possible. While it is acknowledged that input order is not the same as collation required storage order, the encoding should aim to make the task of an IME as simple as possible.
- Consistency. There should be a clear normalisation to ensure a single stored form for a given sign.
    - Comparability. Different inputtings of the same sign with the same spelling should be stored in the same way, at least after normalisation. This allows for searching of signs and sub signs.
- Simplicity. Only the information necessary to spell the sign should need to be input. This is not always completely achievable, for example the order of symbol entry is not necessarily part of the spelling of a sign, but is part of the encoding of the sign.
    - Unicode is a plain text standard. This means that only the contrasts required for legibility need be encoded. If two signs with different meanings (and layouts) end up being stored with the same encoding, then there is insufficient detail in the encoding model. Whereas if two signs with the same meaning and layout end up being stored with different encodings, then there is too much detail in the encoding model.
    - Typographical finesse is not part of the encoding. Only positional information that carries semantic meaning needs to be stored. Positional adjustments for the sake of style and beauty are implementation issues and should not be part of the encoding. Care therefore needs to be taken that where meaning based contrasts occur in one sign, those contrasts do not get used for stylistic ends in other signs.
    - Where extra expressive information is required over standard spelling, such expression should be able to be encoded, but should be considered to be extra information over conventional spelling. This is analogous to the use of diacritical marks in IPA to give precise phonetic representation over the implicitly emic nature of most use of IPA for writing speech.
- Stateless. This means that we can't have any groupings that involve begin and end markers. In addition, to find where a character fits, we need to be able to just scan backwards and state the maximum back distance needed to scan to find the start of an independent group.
    - We can say to scan back until space or a subsign command to identify a subsign. Nested functions are the most tricky. And we don't want any bounded groups where you have to scan for end and start groups to find out where you are.
    - The other problem with endgroup characters is that you have to scan forward until you find them to ascertain the context of a character. This means needing scan forward parsers and that is not wanted in Unicode.

Encoding Model

Rather than trying to describe a model by derivation, we present it as it is and then examine why it meets the encoding need.

The core component of the encoding is the Tuple. A Tuple, in this context, consists of the following components in order:

attach An attaching symbol. This is the symbol we are adding to the sign and which is to be attached to the following *base* symbol.

attach_pos This is an optional position description that gives the location on the *attach* that should coincide with the specified location (*base_pos*) on the *base*. If not present, then it defaults to the touching north position.

base_pos This is the location on the *base* symbol that the *attach* attaches to. It is optional (but must be present if there is an *attach_pos* present). If not present, then it defaults to be the touching south position.

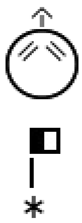
base This is the base symbol or reference to an already attached symbol, that is the symbol relative to which the *attach* will be positioned. This is optional only if there are no positions in the tuple. It defaults to back2. Thus will attach to the previously specified symbol.

Each of the elements in a tuple is a sequence of codes that specify the given information. For example, for a symbol the element will consist of the base symbol + fill + rotation sequence. In the case of a head, the element may also include its modifiers.

In describing character sequences, we use a functional type syntax including parentheses. In reality the parentheses count for nothing and are not part of the actual character sequence, they only exist to help break up the sequence for easier reading and analysis. The basic tuple specification is therefore:

(*attach*, [[[*attach_pos,*] *base_pos*,] *base*])

For example, consider the following sign:



(ASL 'me?', 4.864) This would be encoded, using our abstract syntax, as:

(hand, n_close, s_close, head+modifiers)(modifier, tip, back3)

The sign is made up from two tuples. The first tuple declares the hand (which comes with its fill and rotatino) and positions it in relation to the head, complete with its modifiers. The second tuple declares the touch modifier and positions it in relation to the hand. In describing symbols and positions, we do not use a formal syntax, but use informal names that refer to symbols in the sign. Notice that the second tuple does not declare a new hand, but refers back to the one that was declared in the first tuple. This back reference is a core concept in the encoding.

Back References

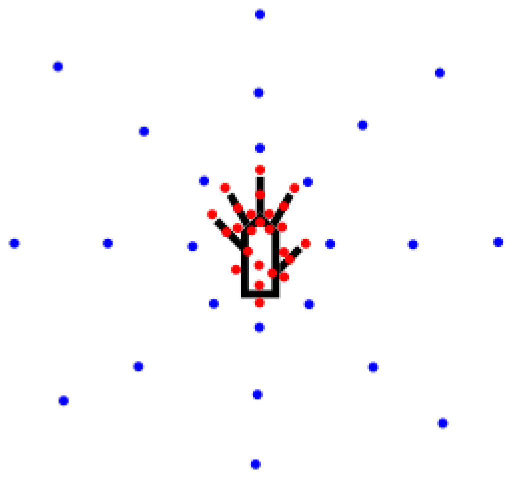
Each time a normal symbol code sequence is used in a sign, it adds a new symbol to the sign. But that symbol needs to be positioned in relation to another symbol. In all cases but the first tuple, where there is no existing symbol to locate the first symbol in relation to, the symbol is located in relation to a symbol that has already been added to the sign. So we can't just use a symbol as the location. Instead we use a special reference code to reference an already existing symbol in the sign.

Back references work by counting backwards real symbols from the current location in the string. Thus for the location of the modifier, back1 would refer to the modifier itself (which is nonsensical, since how can a symbol be position in relation to itself), back2 refers to the head and back3 refers to the hand.

In fact, all tuples, except the first tuple in a sign (or subsign) has a back reference as its *base*. Given, also, that the *base* is optional (with a default value of back2), we can only tell if a symbol is the start or end of a tuple by looking back. If it is preceded by a position, then it is a base. If it is preceded by something other than another symbol, then it is an *attach*. If it preceded by a symbol then we need to look further back. If that symbol is itself a preceded by a position, back reference or a symbol then the symbol we are concerned with is a *base*. This is because only the first tuple in a sequence can contain two symbols, after that all symbols are *base*s of the tuples they are in. And the first tuple in a sequence must be preceded either by a space or non SignWriting character or a **next** type character or punctuation indicating a new sign or sub sign. Is this contextual definition too stateful?

Positioning

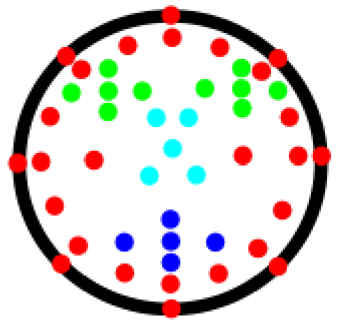
A core concept in the encoding of SignWriting is the expression of the relative positions of symbols. In this section we look at the position elements of a tuple. The basic concept when positioning symbols is that of the attachment point. An attachment point is a position relative to a symbol. A symbol is then positioned relative to another symbol (its base) such that the appropriate attachment point on the attaching symbol coincides with the appropriate attachment point on the base symbol. A typical symbol has a lot of these attachment points. There are two classes of attachment point. There are those that are independent of rotation and are expressed in terms of the centre of the symbol. We call these external attachment points because they are further out from the symbol. The other class are those attachment points that are usually within or very close to the symbol and identify positional features within the symbol. We call these internal attachment points. In this example, we show the external attachment points in blue and the internal attachment points are in red.



The blue attachment points correspond to the compass points at 3 distances from the symbol. These three distances correspond to the three distances that articulators can be apart: close, medium and far. The red attachment points correspond to places on the symbol that another articulator may touch or be over. In this example, there are 5 groups of attachment points: finger tips, finger middles, finger bases within the palm, between fingers and palm positiions (palm centre, palm base inside and outside, palm sides). A different type of symbol may have different internal attachment points. A movement arrow, for example would have very few internal attachment points (head and tail), while having the same external attachment points.

The choice of which attachment point to use for positioning is part of the spelling of a sign. Thus which attachment point to use is based on the meaning or interpretation of the sign and not on typographical finesse. The actual position of an attachment point, or even if an attachment point is used, is up to the font implementor. For example, in the case of the far external attachment points, the font may implement the distance very loosely in terms of a general direction and leave the specific positioning up to the impact that other symbols in the sign have on that distance.

The head is particularly rich with internal attachment points. Notice that addition of modifiers to a head does not add attachment points. They are all there in every head.



The points have been grouped by colour. Green is used for those around the eyes; light blue for the nose; dark blue for the mouth. There are two sets of face psotions in red: the edge of the face and inside the edge for on the face itself.

Notice that some of these attachment points are very close together. Again, the choice of attachment point is not based on where it is on the face so much as the intended location on the face of the signer when they articulate the sign. Thus if there is no eye modifier, there is not much sense in using an eye based attachment point. In such a case the point of interest is the face and not the eye and so a face based attachment point should be used.

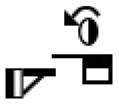
Which attachment points are needed is only in the earliest analysis stage and so there are probably many more that need to be added, particularly among the internal attachment points.

A position code sequence is made up of two components: distance and direction. Each pair is used to identify an attachment point. For internal attachment points, the direction is notional and the distance is used to identify a set of attachment points from which one is chosen using the direction code.

For each direction we can have a default distance, which is touching and therefore does not need to be defined. And for each distance there is a default direction, which is n and as such does not need to be defined. But there is a difference between what the default position is for an *attach_pos* which is touching s and *base_pos* which if absent takes the touching n. This allows for chaining of symbols, particularly movement arrows. Notice that since touching n and touching s are internal positions, they rotate according to the rotation of the symbol they are on. For an arrow, the tip is touching n and the tail is touching s. So typically one will attach the tail to a base and then attach something else to the tip of the movement arrow. This means that chained symbols will, by default, chain in the general direction of the rotation of the symbols. Since both touching and n do not exist as codes, we need a **default** code corresponding to n_touching.

These defaults probably need more discussion.

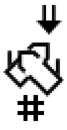
Many symbols do not have nearly so many attachment points as a face. For such symbols, referencing an undefined attachment point is an error.


(ASL 'about', 4.835) It is not uncommon for two articulators to be in an obvious positional relationship but with one of them slightly offset to represent a time or locational offset. For this we can add a positional modifier: **offset** which says to shift the position a fixed amount (which is left up to the font designer) tangentially to the attachment point (i.e. at 90° to a line between the attachment point and the centre of the glyph) in a clockwise direction. If applied to an attaching glyph, this will have the effect of moving the glyph in the opposite direction. If applied to a base position it will move the attaching glyph in the same direction. In addition, if an offset is applied to a base, then all attaching symbols are affected. Therefore an **offset** may only be applied to an attachment position. To get an offset in the opposite direction use an **offset_back** code. Thus this sign would be encoded:

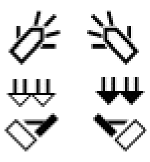
(right_hand, default+offset_back, default, left hand)(wrist+fill, s_close, n_close, back3)

Ordering Tuples

It may seem rather odd that the tuple order puts the attached symbol before the base symbol. The typical order for things in Unicode is that the base is stored before the diacritic. To examine the reason for this consider the sign:


(ASL 'proof' 4.373) In this sign, the dominant hand is rendered on top of the supporting hand. It is clear that the base character here is the supporting hand and that the dominant hand attaches to it. But when sorting signs, we want to sort signs by the dominant hand first. And since there is no way to know which hand is dominant apart from the ordering of the hand symbols in the sign, we must come up with a consistent ordering. That ordering needs to have the dominant hand first. So if we want to have the attaching hand first, we have to have an order of *attach* followed by *base*.

Another question is: what order should the tuples occur in. If we just describe things in terms of other things, aren't we saying that you can put symbols down in any order?


(ASL 'wet', 4.286) There are at least two ways we could encode this sign using tuples:

(right_hand, w_close, e_close, left_hand)(filled_arrows, s_close, back3)(right_hand, ne_close, default, back2)(unfilled_arrows, s_close, back4)(left_hand, nw_close, default, back2)

which stores all the movement and resulting hand shape for the dominant hand before storing the same for the supporting hand. Or there is:

(right_hand, w_close, e_close, left_hand)(filled_arrows, s_close, back3)(unfilled_arrows, s_close, back3)
(right_hand, ne_close, default, back3)(left_hand, nw_close, default, back3)

which stores the hands then the movements then the final hands.

There is an existing recommendation for ordering of symbols in a sign which is called SignSpelling. This is used for the purposes of collation. We have already stated that sorting requires the dominant hand is stored before the supporting hand, but it goes on to say that of the two encodings of ASL 'wet', the second gives the best sorting and is the more common way for people to consider the sign.

Normalisation

Is there some way that we can automatically order tuples correctly? There are two algorithmic approaches that can be taken when sorting tuples:

Breadth First

In this normal form tuples are sorted on the following basis:

- Those closest in path length (number of intervening necessary tuples) to the initial symbol are stored first.
- Heads and limbs take precedence over hands that take precedence over modifiers that take precedence over movements.
- Dominant articulators take precedence over supporting symbols. Of course, deciding what is a dominant articulator symbol is the essence of the problem!

This results in a top down breadth first ordering with all articulators declared before movements declared before subsequent articulator positions and so on. This more closely reflects the order that signspelling gives and so is more suited to sorting. The advantage of this ordering is that the distance of the longest back reference is minimised.

Depth First

In this form the sum of the total back reference distance is minimised. The sorting is on the following basis:

- The tuple with the resulting shortest back reference takes precedence.
- Heads and limbs take precedence over hands that take precedence over modifiers that take precedence over movements.
- Dominant articulators take precedence over supporting symbols.

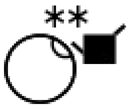
This results in a top down depth first ordering with all the movement and subsequent positions of the dominant articulator occurring before the supporting articulator movements and positions. This more closely reflects what a functional model would have as a resulting order of symbols. The advantage of this ordering is that there are more occurrences of a back references of length 2 which could, potentially, be removed as default values.

Notice that it is only the first sorting rule in the two algorithms that differs.

In addition we add the rule that a tuple may not start with a back reference. That is, we don't declare bases in terms of attached symbols. This constraint also stops cycles in the tree, as does not declaring a back1.

More Complex Tuples

Unfortunately, simple tuples are not always sufficient. Here we introduce some special tuples that are marked with a special function code at their start. Consider:



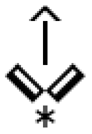
(ASL 'cow', 4.58) This is in effect using a shared attachment point. But since that shared attachment point is defined on the base, we can simply use two tuples for this:

(hand, e_close, nw_touching, head)(modifier, s_close, nw_touching, back2)

This does raise a spelling question of whether in such cases the modifier attaches to the head or the hand. This is also a valid spelling:

(hand, e_close, nw_touching, head)(modifier, s_close, e_close, back3)

More complex shared attachment points are handled in the same way

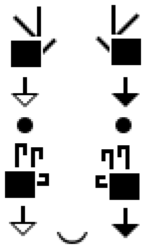


(ASL 'boat', 4.78) This is encoded:

(right_hand, default, default, left_hand)(modifier)(arrow, s_med, default, back3)

Often, when using movement dynamics, the dynamic is positioned in relation to the final hands in a symbol. The problem is that these hands have already been positioned in relation to other, unrelated symbols and so there is no common contact point to position the dynamic against. Instead we say that the dynamic is positioned relative to the midpoint between two symbols:

Consider the sign:



(ASL 'selfish', 4.99) This might be encoded:

(right_hand, w_mid, e_mid, left_hand) (filled_arrow**,** s, back3) (unfilled_arrow, s, back3) (dot, back3) (dot, back3) (right_hand, back3) (left_hand, back3) (filled_arrow, s, back3) (unfilled_arrow, s, back3) **between**(dynamic, n, w, back3, e, back2)

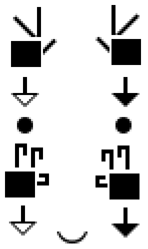
The **between** tuple here says to position the dynamic below a point halfway between the two attachment points on either side of the final hands. This gives us a syntax of **between** as:

between(*attach*, *attach_pos,* [*base1_pos*], *base1*, [*base2_pos*], *base2*)

Comparing Models

In considering the model used here, there is a natural question as to why a more functional model is not more appropriate. The encoding of Ideographic Description Characters uses a functional model where codes are operators taking following parameters, which my themselves be operators with their own parameters and so on.

Consider the sign we just analysed:



(ASL 'selfish', 4.99) This might be encoded:

(**right_hand**, w_mid, e_mid, **left_hand**) (**filled_arrow,** s, back3) (**unfilled_arrow**, s, back3) (**dot**, back3) (**dot**, back3) (**right_hand**, back3) (**left_hand**, back3) (**filled_arrow**, s, back3) (**unfilled_arrow**, s, back3) between(**dynamic**, s, w, back3, e, back2)

If this were to be stored using a functional model in which the nodes in the tree are put together as parameters to functions, we might get something like:

single(single(**left_hand**, s, single(**unfilled_arrow**, s, single(**dot**, s, single(**left_hand**, s, reftag(**unfilled_arrow**))))), w single(**right_hand**, s, single(**filled_arrow**, s, single(**dot**, s, single(**right_hand**, s, single(reftag(**filled_arrow**), between(ref1(), ref2()), n, **dynamic**))))))

Notice how in the functional model the two dots are nowhere near each other in the stored order of symbols. In the tuple model, the dots are adjacent as is required for the signspelling of this symbol. This signspelling order also reflects the desired input order, even if the initial input order would be left_hand, right_hand, while the signspelling calls for the right_hand, left_hand order as shown in the example. In effect, the functional model cannot handle the non-chaining order required. Functional models require that visually adjacent elements be adjacent parameters, although subtrees may intervene between parameters to a function due to recursion.

Another consideration is that of operator length. If one compares the longest tuple with the longest function one will notice that the whole sign is in a function in the functional model and each half is in its own function. This results in poor locality which makes implementation harder. Tuples, on the other hand, has very good locality. The substrings are short. There is no way to ignore the fact that symbols that may be stored far apart, need to interact. The back references are the non-locality in tuples and such non-locality is made very explicit.

Tuples have the ability to be stored in different orders, and this brings normalisation issues. But a functional model is not immune to this. For example, in the above example, the left_hand and right_hand chains could have been swapped. And the restricted order of a functional model is exactly what causes problems for other processes like keyboarding or collation.

Implementation Considerations

SignWriting is probably the most complex script needing to be implemented. In this section we examine some implementation issues for various processes.

Rendering

Regardless of the encoding model, we do not think that there is any existing smart font technology that can handle SignWriting. Graphite might just be able to handle it, but it would gain from some minor extensions to the language to make the descriptions much more manageable. OpenType is also close but would need the capability to call a mark type feature with non-adjacent glyphs. Again this is probably not hard, and could probably be achieved by the addition of a new feature.

IMEs

One of the complexities of SignWriting is that the keying order is often different from the intended storage order. People want to type the location before the thing being located. This makes logical sense. To get a storage order that reflects sorting, an IME is going to need to do some reordering of keystrokes from keying order to stored order. Thankfully, the tuple model limits the extent of that reordering to a single tuple. Thus if one were to type a head and then a hand, the IME would need to reorder that as hand followed by head. This is not easy, but is possible. In addition when we consider the idea of location chaining, that adds to the complexity of data input. The expectation, therefore, is that a special IME will be needed for data entry of SignWriting. It is expected that a basic IME will be just about possible using a generic IME engine such as Keyman[1].

Collation

According to SignSpelling, the preferred collation order is based on the dominant hand, supporting hand and then the head. But if the dominant hand is anchored on the head, then the head will occur before the supporting hand. The sorting issue can be resolved by making the head and all its modifiers primary ignorable. But even then it is very likely that the encoding order will end up being somewhere between a keying order and a SignSpelling order. This is probably sufficient because there is not yet the experience in sign collation to have fixed the later aspects of the order. It is our impression that so long as the dominant hand is considered first, that may be sufficient. If, in addition the supporting hand is considered second, the precise order after that would be acceptable so long as it was consistent, regardless of its actual order.

Searching

Searching for symbol sequences in tuple is not as easy as it might be. Consider the question: I want to find symbol Y which is attached to symbol X in positional relation P, what regular expression would be needed?

We start by defining some sub expressions and classes. Sub expressions are italicised while single character classes are not.

[1] http://www.tavultesoft.com/keyman

p = pos sequence (distance direction? | direction | default){1,2}
b = base character sequence excluding head (symbol fill? rotation?)
r = class of backreference characters
h = head sequence (head modifiers*)
number on its own is a backreference 2-6

As with most searching problems, the more we can constrain the problem, the shorter the regular expression can be. In this example, we know that X and Y are in direct relationship. This limits how far the symbols are going to be apart, but due to ordering questions, this can be still quite far.

If we can say that X and Y are not part of the initial tuple (i.e. are not the dominant or supporting hand), then we can use this regular expression:

X*p*rYP2|X*p*r*bp*rYP3|X*p*r*bp*r*bp*rYP4|X*p*r*bp*r*bp*r*bp*rYP5

At this stage of analysis, we will say that what is really (*p*r|r?) is simply *p*r. We will return to this later.

If we say that X is the dominant hand, then the regular expression would be:

XPY|X*pb*YP3|X*pbbp*rYP4|X*pbbp*r*bp*rYP5|X*pbbp*r*bp*r*bp*rYP6

Finally, if X is the supporting hand, then we would use:

XYP2|X*bp*rYP3|X*bp*r*bp*rYP4|X*bp*r*bp*r*bp*rYP5

And if we want to say X can occur in any of those positions, then we join all those expressions together as alternatives.

XPY|X(*p*r)?YP2|X((*p*r)?*bp*r)|(*pb*)YP3|X((*p*r)?*bp*r)|(*pb*))*bp*rYP4|X((*p*r)?*bp*r)(*pb*))*bp*r*bp*rYP5

If we factor out ((pr)?bpr)|(pb) as a single subexpression representing a symbol within a tuple and call it *s*. While we are at it, we can put back the approximation regarding *p* and r. We also define *t* as being a tuple.

t = *b*(*p*r|r?)

s = ((*p*r|r?)*t*)|(*p*?*b*)

The expression becomes:

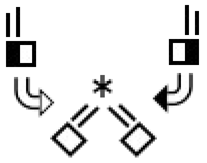
XPY|X(*p*r|r?)?YP2|X*s*YP3|X*st*YP4|X*s*(*t*){2}YP5|X*s*(*t*){3}YP6|X*s*(*t*){4}YP7|...

Even simplified, this expression is still rather complex (and, in this document, untested).

Analysing this search expression, we might see that if the tuple order were changed from (attach, attach_pos, base_pos, base) to (base, base_pos, attach_pos, attach) and so the base always precedes the attach, and if we then chained in sequence all the symbols that attach to each other, then searching for this kind of sequence would become trivial. But that is a big proviso and the impression given by the SignWriting community is that they do not want to go that way.

More Signs

We finish this discussion by examining some more signs and how they would be encoded. Each has its own challenges and raises questions worthy of input from other experts in the field and particularly from the SignWriting community.



(ASL 'introduce', 4.26). What makes this sign fun is that the sign starts with unpositioned hands and ends in a positioned relationship. If we consider the initial hand position to be positioned, then we have a cycle.

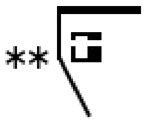
(right_hand, e_far, w_far, left_hand)(filled_arrow, s, back3)(unfilled_arrow, s, back3)(right_hand, ne, back3) (left_hand, nw, back3) (**back2**, tip, **back1**) (modifier, s, tip, back2)

So much for no back1 and for no cycles. I am assuming that the spelling here is that the arrows point to the hands and not to the contact point. If we say they point to the contact point, we now have a contact point based on two bases, so it would need a **between**.

(right_hand, w_far, e_far, left_hand)(filled_arrow, s, back3)(unfilled_arrow, s, back3) **between**(modifier, s, back3, back2) (right_hand, default, back2) (left_hand, default, back3)

Is it possible to encode this sign backwards and so remove the cycle?

(right_open_hand, default, default, left_open_hand) (modifier) (filled_arrow, w_med, default, back4) (unfilled_arrow, w_med, default, back4) (right_hand, s, s, back3) (left_hand, s, s, back3)

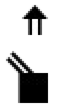


(ASL 'drugs', 4.9) What makes this sign interesting is the limbs and that we have 4 things anchored at the same place:

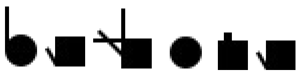
(hand, e, bottom, vertical_limb)(shoulder, left, top, back2)(angled_limb, top, bottom, back3) (modifier, e, bottom, back4)

There are some spelling questions here:

- Does the hand attach to the bottom of the vertical limb or the top of the angled limb?
- What symbol order should be used for this sign and why?



(ASL 'North Dakota', 4.39) This is primarily finger spelling. And ideally we want finger spelling to be simply a sequence of symbols if we can pull that off. The problem is that we have used sequence to group sub signs. The other alternative is that we use sequence to do left to right and then have a sub sign marker.



(n_hand, s, arrow) (d_hand, w, a_hand) (k_hand, w, back2) (o_hand, w, back2) (t_hand, w, back2) (a_hand, w, back2)

That's not very pretty. Can we do better. Could we have a marker for subsigns and then just to do symbols across or down the page?

(n_hand, s, arrow) **next_down**() (d_hand, a_hand, k_hand, o_hand, t_hand, a_hand)

Given the specification of a tuple, we can, if we adjust the symbol to have the hand spelling rendered vertically. We can parse the above string, including all its defaults in italics, as:

(n_hand, *n_touching*, s, arrow) **next_down**() (d_hand, *n_touching*, *s_touching*, a_hand) (k_hand, *n_touching*, *s_touching*, *back2*) (o_hand, *n_touching*, *s_touching*, *back2*) (t_hand, *n_touching*, *s_touching*, *back2*) (a_hand, *n_touching*, *s_touching*, *back2*)

Notice that this works because the rotation of each of the hand spelling symbols is vertically oriented. If the hands were rotated, then so would the chaining, which would not be as desired.

This whole question of good defaults for absent elements could do with more thought. The consideration requirements are:

- non-contextual resolution of default positions or references.
- As much as possible, only one way to encode the same spelling. Thus defaults are required to be used if they can be.

Notice that the second consideration is not about the concern of having two ways to spell the same sign, but encoding the same spelling two ways. These considerations raises the question whether defaults are desirable in the various places they are proposed.