Japanese HWR

Steven B. Poggel
steven.poggel@gmail.com

February 5, 2010

Contents

0.1 Abstract

In this work I present and application that uses state of the art Chinese/Japanese handwriting recognition methods in order to provide an Kanji teaching application with an error correction.

Conceptually, the application is an e-learning environment for Japanese characters, intended for the foreign learner of the Japanese language. In order to provide more than a multiple choice method, like most other systems, the application contains a handwriting recognition engine that can be used preferably with a hand-held device like a PDA, but generally any stylus input device.

Chapter 1

Handwriting Recognition Engine

1.1 Data Capturing

Each handwriting recognition process begins with the data capturing. The user's handwriting must be captured and fed into the system. The data capturing is therefore a crucial part of the whole process. In this system a GUI is used for the capturing of the pen movements on a writing surface.

1.1.1 Writing Surface

The writing surface module, the view is split into two parts. The writing surface GUI and the writing surface background module. The technical design of the data input GUI is described in section **??**.

1.1.1.1 Writing Surface GUI

The GUI works with pen-down and pen-up events. It has a cross in the middle in order to partition the writing surface the same way, character practicing paper sheets are usually partitioned. The GUI class is listening to *pen-down*, *pen-up* and *pen-move* events. These are the equivalent in the mobile world for regular mouse-down, mouse-up and mouse-move events. However, there is one difference - the pen-move event can only be captured between a pen-down and a pen-up event. An earlier conceptual idea for the HWR engine included using the mouse-move events during the input of a character between the strokes. This could not be realised, however, because the series of pen-move events can only be captured when there has been a previous pen-down event and no pen-up event yet. The GUI captures the events and passes the point coordinates on to the background class.

1.1.1.2 Writing Surfaces Background Module

When a pen-down event is detected, the background class of the GUI starts listening for the pen movement. All point coordinates and the time of their capturing are stored in two separate lists with the same indeces. An alternative solution would have been to store a number of instances of a custom-made encapsulated Point class including the time stamp in one list only, however, using two separate lists resulted in increased speed. Therfore, the point coordinates are stored in the frameworks Point class that does not account for time stamps. Therefore a separate list is used for the timestamps only.

The background module of the writing surface mainly administrates the capturing of the pen trajectory and sends it to the recognition module as soon as a stroke is finished. Therefore the system does not receive a separate signal indicating that the drawing of the character is finished. That design creates as segmentation problem that is solved with the *clear* button and the *clear* message. The segmentation of characters is left undetermined - only the beginning of it is determined through the *clear* message that is sent to the moblie view from the controller - or the clear message that is sent to the controller because the user clicked the *reset* button. After a stroke is finished the according point and timestamp sequences are passed to the main part of the HWR engine.

1.2 Data Format

1.2.1 Requirements of the Data Format

The data format for the recognition process underlies a number of requirements. Firstly, it has to be stated that two main data formats are necessary. One for the actual recognition process as a data structure in the RAM. Secondly, there needs be a storage format for the data base that represents the handwriting of a character, radical, stroke, point list or simple point. The requirements for both data structures are:

- **Expressiveness**: The data structure should be able to fully represent a complete character and all sub-elements that belong to it. That requirement is due to the structural approach to handwriting recognition that is the basis for the error handling.
- **Well-definedness**: The information structure should be unambigous, two different characters must have a different structure.

The storage structure has some additional requirements:

- **Accessibility**: The data should be formatted in a way that a human editor can access the data, but it should also be prepared for programmatical access.
- **Well-formedness**: There should be a way to check if the data is well-formed
- **Parsabilty**: It should be possible to create the run-time data structure from the storage data structure.

The run-time data structure should in addition provide

- **Serialisability**: It must be possible to create the storage data structure from the run-time data structure.

Any combination of two formats used should meet the above requirements, at best in a uniform way. That is, any combination of storage format and run-time format should ideally be compatible without the need of a complex data format converter.

1.2.2 Existing Handwriting Formats

There are some existing formats for the description of handwriting. Their intended use is for handwriting recognition systems. The formats that will be reviewed in this section are the *Microsoft ISF* format in section (1.2.2.1), which is a binary format. The other formats are text based and human-readable. In this short review, the formats will be checked against the requirements defined in section (1.2.1). We will consider *InkML* in section (1.2.2.2), *UNIPEN* in section (1.2.2.3), *hwDataset* in section (1.2.2.4) and *UPX* in section (1.2.2.5). The central question is, if one of these formats meets the above requirements in a sufficient way. This is unclear, because the most of the formats have been originally developed mainly for alphabetic scripts or for usage scenarios different from the one proposed in this chapter.

1.2.2.1 The Microsoft Ink Serialized Format (*ISF*)

The *Ink Serialized Format* (ISF) format is owned by the Microsoft Corporation. However, it is not a proprietary format, the specification is freely available. Since the format is purely binary, it can not meet one of the requirements for the storage data structure: It is not feasible for a human to read and write files in that format, therefore ISF does not provide the necessary *accessibility* (**?**). The format could still be used as a run-time data structure, if it meets the requested criteria. The requirement of *serialisability* is generally met. A binary format allows for fast and reliable programmatical access to the data, it can be stored in binary files on a data storage medium. The problem that accrues from this type of serialisation is that the storage format, the format used in a file or database, would again not be human-editable and therefore fail to meet the *accessibility* requirement. In order to provide a serialisation that results in a human-readable format, a format converter would be needed. Another problem originates from the ISF format specification:
It accounts only for the description of mouse coordinates or pen trajectories, but does not offer any structures for linguistic information. It fails to meet the ideal, a unified format for linguistic and graphic information, as it is lacking some degree of *expressiveness*.

1.2.2.2 InkML

InkML is a markup language for describing electronic pen trajectories. Hence the name *ink*. *ML* stands for markup language, as is customary for XML-based and SGML-based languages. The specification is currently in a draft status, maintained by the World Wide Web Consortium (W3C) (**?**). Judged from the viewpoint of the defined requirements, InkML is a candidate for the storage format. As an XML format it automatically fulfils the criterion of *well-formedness*: Any piece of InkML code can be evaluated with common techniques, since InkML has a well-defined scheme description.

The main elements in the simplest form of InkML are the **<ink>** and the **<trace>** element. These elements allow for a very simple form of a pen trajectory, basically a flat list of point coordinates. The <ink> element is the root element of any InkML code. The <trace> element holds point coordinates.

Listing 1.1: Demonstration of the *trace* tag

```
<ink>
  <trace>
    282 45, 281 59, 284 73, 285 87, 287 101, 288 115, 290 129,
    291 143, 294 157, 294 171, 294 185, 296 199, 300 213
  </trace>
</ink>
```

Listing (1.1) covers the general gist of the InkML format. Despite being an XML format, it is a flat format, each value pair, separated by commas, represents one point in a coordinate notation.

It is possible to add *time* information to the trace. In order to do so, a time channel needs to be defined. Listing (1.2) shows the definition and use of a time channel. The example shows a time channel whose values for a given point are the relative to the timestamp refered to to by *#ts001* (**?**). Below there are two *timestamp* tags. The first one has the ID *ts001* and is referred to by both the time channel and the second time stamp that defines a time offset to time stamp *ts001* with the *timeOffset* attribute. The possibility to define time channels enables the InkML format to hold information about the time at which a sample point has been taken. This is useful for the recognition mechanism used in this application, because it compares point coordinates as well as time stamps.

Listing 1.2: Demonstration of the *time channel*

```
<channel name="T"
         type="integer"
         units="ms"
         respectTo="#ts001"/>

<timestamp xml:id="ts001"
           time="1265122414000"/>
<timestamp xml:id="ts002"
           timeOffset="600000"
           timestampRef="#ts001"/>
```

The InkML data structure fulfills several of the requirements necessary for a storage data structure. InkML is

- *well-defined*: Any point sequence that has different coordinate values than another point sequence can be distinguished from the other.
- *accessible*: As an XML format it can be handled programmatically, but is also human-readable and can be edited by a human with a simple interface like a text editor.
- *well-formed*: As an XML format it can be validated according to it's specification.
- *parsabale*: As an XML format, InkML does not need to be parsed with custom-made methods, since most modern high-level languages offer an access method to XML tree structures.

However, InkML is lacking some *expressiveness*. Only pen trajectories and their time stamps can be expressed in the InkML format. InkML is not designed to hold any of the other information compulsory for the structural handwriting recognition. In the case of character recognition, it needs to be able to account for more than coordinate points and time stamps. There needs to be a way to encode structural information about the characters and sub-elements of the characters. InkML does not to be a sufficient format for the given task.

1.2.2.3 The Standard UNIPEN Format

The standard UNIPEN format specification is a file format definition for a flat text file. It contains tags in dot-notation. For example the tag *.COMMENT* means that the following free text should be ignored, the tag *.KEYWORD* is used to define a new keyword, while the tag *RESERVED* states that the text that comes after that tag is a reserved word within the UNIPEN format. The format is self-defined with the three keywords above. Any new keyword is defined with *.KEYWORD* (?; ?).

As a data structure for the purpose of a handwriting recognition, UNIPEN can serve as a storage format, which is the primary purpose for the existence of the format. The UNIPEN format accounts for both pen trajectories as well as information about the characters that have been written.

Listing 1.3: Demonstration of the *UNIPEN* format

```
.COMMENT ##########################################################################
.COORD   X Y

.SEGMENT TEXT 235:0-297:9 OK "Kurosu␣Masaaki"
.SEGMENT CHARACTER 235:0-255:9 OK  "JISx3975␣'Kuro'"

.PEN_DOWN
  486 -1456
  488 -1454
  490 -1452
  488 -1450
  488 -1450
  486 -1452
  480 -1456
  474 -1466
  464 -1480
  452 -1492
  440 -1506
  428 -1524
.PEN_UP
  406 -1556
  394 -1574
  384 -1590
  374 -1602
.PEN_DOWN
```

Listing (1.3) is an excerpt from a data file created by (?). It shows an example of a UNIPEN data structure. The *.COORD* tag defines the structure of the pen coordinates. The *.SEGMENT* tag with the *TEXT* modifier informs about the full text of the next number of segments. The *.SEGMENT* tag with the *CHARACTER* modifier informs about the character that is represented with the sequence of pen coordinates. The other information given is the start and end position within the file and the character code in JIS encoding. The *.PEN_DOWN* tag can be understood as a flag. All pen coordinates following this tag have been captured as *pen down* coordinates. The tag *.PEN_UP* sets the opposite flag: The coordinates stated after this tag were captured while the pen was not touching the writing surface. The last *.PEN_DOWN* tag is the beginning of a new coordinate sequence for the next stroke. From a requirements point of view the UNIPEN provides a high standard, as it meets a great number of requirements for the storage structure.
The UNIPEN format is:

- partially **expressive**: UNIPEN can represent a complete character and the pen trajectory along with it. It does however, not account for capturing the time information. At a constant sample rate the time stamps of the individual points could theoratically be calculated, if time stamp of the first point is encoded in the meta information. However, that solution is not optimal, because it needs constant sampling, even between a *pen-up* and a *pen-down* event. There are input devices that do not offer those coordinates and the recognition system presented here does not rely on them for that reason. The examples given are encoded in JIS, but it seems possible and feasible to use the format with unicode encoding.

- **well-defined**: The UNIPEN is unambigous, two different characters do have a different data structure, because the JIS code can serve as an ID.
- **accessibile**: The data resides in flat text files, designed for human editing. It is accessble programmatically, too, but has a weakness on that requirement compared to an XML-based format.
- **parsable**: The flat file format can be parsed easily. Due to its procedural nature it is also possible to create a run-time data structure that is conceptually based on the storage structure and therefore serialisable.

The main weakness of the flat format file is the lack of *well-formedness*. There is no automatic way to check a document against a predefined format specification. It can not be assessed if a file is well-formed. Additionally, the expressive power of the UNIPEN format would be seriously challenged if it should provide for sub-structures of characters. In order to do so, it would be necessary to create new tag definitions for the Radicals of the Kanji. **?** (**?**) describe some more shortcomings of the UNIPEN format. It can be concluded that the standard UNIPEN format is not suitable for the task given.

1.2.2.4 Handwriting Dataset (*hwDataset*)

The *Handwriting Dataset* (hwDataset) is an XML format that is a complement to InkML. It is inspired by the UNIPEN format. The *hwDataset* format attempts to close the gap between pure ink data and the annotations that are needed for handwriting recognition (**?**). The format contains three main parts. The *datasetInfo*, the *datasetDefs* and the *hwData*. The datasetInfo element holds any metadata, like *name*, *category* and the like. The dataSetDefs encompasses information about *data sources*, different *writers* and their features like *handedness*, *gender*, *age*. The *hwData* element in the XML code organises the handwriting data hiearchically. Each hierarchy level contains one or more *hwTrace* elements. The hwTrace element refers to an InkML file, containing the actual handwriting data. A detailed description of the hwDataset format can be found in (**?**).

The format provides an XML scheme description and thus meets the *well-formedness* requirement. It is *well-defined*, because it accounts for unique structures. The *parsabilty* is not an issue for any XML format, since methods for accessing XML trees are provided by modern programming libraries. The *accessibility* is provided, the format has a clearly defined structure that can easily be understood by humans. The hwDataset format can structure many desirable information for the handwriting learning system proposed in this chapter. From an *expressiveness* viewpoint it can create substructures for different parts of a sequence of pen trajectories. The *hierarchy level* (H) of a trace accounts for the expressive power necessary. The H(n) elements of the hwData elements are designed to hold meaningful names like *PARAGRAPH* or *WORD*. It may be possible to use the format even to denote different parts of the same character. There is no reason why one should not introduce an H(n) definition for *RADICAL* or *GRAPHEM*.

1.2.2.5 The UNIPEN XML Format (*UPX*)

The format *Unipen for XML* (UPX) can be seen as an XML version of the UNIPEN format. The UNIPEN standard described in section (1.2.2.3) does not bear any resemblance with the InkML format shown in section (1.2.2.2). Both formats fulfil a number of the requirements presented in section (1.2.1), but both formats fail to fulfil them all. The hwDataset format described in section (1.2.2.4) solves many of the problems UNIPEN faces, as it brings together a superset of the expressiveness of UNIPEN with the well-defined InkML format in a uniform way. According to **?** (**?**) hwDataset was designed to support new data collection. The motivation to create another format came about from the fact that existing UNIPEN resources should be made available in a modern XML-based format that unifies InkML and UNIPEN. hwDataset comes very close to this aim, but does not necessarily account for a conversion of UNIPEN resources. **?** (**?**) attempt to create a format that allows for a well-defined conversion from

UNIPEN. The UNIPEN foundation considers UPX as the new de facto standard format for storing annotated databases of online pen input (?).

UPX is a multi file format. It resembles hwDataset in the way that it comprises both meta information and data annotations as well as pure InkML data in separate files. Within the UPX *hwData* element there can be any number of *hLevel* elements in order to define the hierarchy of the trajectory. Listing (1.4) shows an example after (?). In the listing, the *hLevel* tag is demonstrated. The word *sexy* is described hierarchically. Firstly, in the outermost *hLevel* tag, the *level* attribute specifies a *WORD*. Then the inner *hLevel* elements specify a lower level of hierarchy, the characters. Note that not all characters are defined individualy in this example. The *hwTraces* elements surround a number of *traceView* elements that in turn point to the corresponding positions in an InkML file that holds the pen trajectory. The meta data information that can be stored in UPX has a similar expressive power as the hwDataset format. ? (?) give a full format description of the UPX format in their technical report.

Listing 1.4: Demonstration of the *hLevel* tag in UPX

```
<hLevel id="WORD_0" level="WORD">
  <label id="WORD_0">
    <alternate rank="1" score="1"/>sexy</alternate>
  </label>
  <hwTraces>
    <inkml:traceView traceRef="w0629.inkml#dataSet_0_traces"
                     from="22"
                     to="24"/>
  </hwTraces>

  <hLevel id="CHAR_0" level="CHAR">
    <label id="CHAR_0">
      <alternate rank="1" score="1"/>e</alternate>
    </label>
    <hwTraces>
      <inkml:traceView traceRef="w0629.inkml#dataSet_0_traces"
                       from="22:91"
                       to="22:161"/>
    </hwTraces>
  </hLevel>

  <hLevel id="CHAR_1" level="CHAR">
    <label id="CHAR_1">
      <alternate rank="1" score="1"/>y</alternate>
    </label>
    <hwTraces>
      <inkml:traceView traceRef="w0629.inkml#dataSet_0_traces"
                       from="24:61"
                       to="24:162"/>
    </hwTraces>
  </hLevel>
</hLevel>
```

It seems as if the UPX format was able to fulfil all requirements. The UPX format fulfils the requirements of

- *Well-definedness*: As an XML structure providing IDs and names for each element and substructure, the format is well-defined.
- *Accessibility*: As an XML format it is both human and machine readable and editable.
- *Well-formedness*: As an XML format with a defined scheme it can be validated for its syntactic features.
- *Parsabilty*: As an XML format it can be easily parsed into a tree structure.

- *Expressiveness*: The format provides the missing link between the standard UNIPEN format and the InkML format. Additionally, it can describe hierarchical structures.

Since any XML format translates directly into a tree structure, the data structure for the run-time does not need to be defined separately. Conversing pen trajectories in InkML format into a data stream has been studied (?). These techniques however, are not necessary for the proposed system. The data exchange is provided as technique available within the framework without any conversion (see section ??).

1.2.3 Data Format Description

The data format description could trivially be described with the UPX description. In this case however, UPX needs to be adapted to the special needs. This section describes the adaption of UPX to the requirements of an on-line handwriting recognition for Kanji characters in a learning environment. The section describes both the file format and the run-time data structure.

1.2.3.1 Point Data Format

In UPX, points are not represented directly, but in an outsourced InkML file, where they are part of the InkML standard pen trajectory definition. In InkML files, points are stored in traces as specified in the InkML format description.

Listing 1.5: Definition of the trace format

```
<timestamp xml:id="ts001"
           time="1265122414000"/>
<traceFormat xml:id="kanjiStrokeTrace">
  <channel name="X" type="decimal" />
  <channel name="Y" type="decimal" />
  <channel name="T" type="integer"
                    units="ms"
                    respectTo="#ts001" />
</traceFormat>
```

Listing (1.5) shows the trace format description as used in the system. The format consists of three channels that are given in ordered sequence. There are no intermittent channels. The first two channels are X and Y for the pen coordinates. The third channel is T for a time stamp. The pen coordinates can be given as *decimal* numbers, while the time stamp (in milliseconds) must be an *integer*. The *respectTo* attribute in the time channel indicates that all values must be interpreted as offsets to the time stamp with the id *ts001*.

An actual trace would then contain three values, each value triple separated by a comma. The trace format definition ensures the correct interpreation of the values.

Listing 1.6: A sample trace

```
<trace id="id123abc">
  45 76 0, 3 5 100, 4 -8 200, 4 -5 300,
  9 -2 400, ...
</trace>
```

Listing (1.6) shows a trace. Given the file contains the trace format definition shown in listing (1.5), the trace in listing (1.6) is interpreted in table (1.1).

The run-time data structure for a point holds the coordinate values and an integer value for the time stamp. The conversion between the run-time instance and the trace format is trivial and needs no description.

1.2.3.2 Stroke Data Format

There is no explicit format for a stroke in the storage structure. A stroke is simply the trace between a pen-down event and the following pen-up event. All the points captured between form one stroke.

Trace	Channel X	Channel Y	Channel T	Comment
45 76 0	45	76	1265122414000	The first trace initialises the values. The time stamp value is read from timestamp *ts001*.
3 5 100	48	81	1265122414100	The delta values are added to the existing data.
4 -8 200	52	73	1265122414200	The time stamp values are relative to *ts001*, not to the last time stamp, unlike the pen co-ordinate deltas that are added to the current value trace by trace.
4 -5 300	56	68	1265122414300	Negative values are treated as regular delta values.
9 -2 400	65	66	1265122414400	The values change as expected.

Table 1.1: Sample trace interpreation

Thus, in the InkML file format a stroke is simply a trace. If a greater number of strokes comes together, each one of them will be a trace and they can be groupe together as a tracegroup element in InkML. It would be possible to create a separate level for strokes, as demonstrated in listing (). However, since every *hLevel* element for a stroke would contain exactly one *hwTrace* element, it seem redundant to do so. Listing (1.7) shows a potential data structure for a stroke. As stated befere, this data structure is redundant in the way, that an *hLevel* definition would contain only exactly one trace. If a trace is interpreted as a stroke, the additional *hLevel* for strokes becomes unnecessary.

Listing 1.7: An example of how a stroke could be represented in UPX

```
<hLevel level="stroke" id="STROKE1">
<!-- The first stroke in this Radical of handwriting. -->
  <label id="STROKE1label" labelSrcRef="labelref_DW" labelType="truth">
    <alternate>[unicode value]</alternate>
  </label>
  <hwTraces>
    <inkml:traceView traceRef="/example.inkml" from="12" to="12"/>
  </hwTraces>
</hLevel>
```

The run-time data structure for a stroke is generated from a trace. Despite not being represented in the storage structure, the run-time data structure for a stroke plays a crucial part in the recognition process. The actual data wihtin the structure is purely a list of point instances. Any other numerical values are computed from the point list, with algorithms that are described in detail in section (1.5).

1.2.3.3 Radical Data Format

The Radical data format is a data structure that works with the UPX *hLevel* element for hierarchical description. Conceptually, a Radical is a hierarchical element in the composition of a Kanji character. With the UPX format specification in hand it is fairly straightforward to define an appropriate data structure. Listing (1.8) shows a UPX data structure for a Radical. The Radical 木 consists of four strokes. Since strokes are not represented explicitely in the data format, there are four *traceView* elements under the *hwTraces* tag.

Listing 1.8: A Radical representation in UPX

```
<hLevel level="radical" id="RADICAL75">
<!-- The first Radical in this Kanji character of handwriting. -->
  <label id="RADICAL75label" labelSrcRef="labelref_DW" labelType="truth">
      <alternate>U+6728</alternate><!-- Unicode value -->
  </label>
  <hwTraces>
    <inkml:traceView traceRef="/example.inkml" from="12" to="12"/>
    <inkml:traceView traceRef="/example.inkml" from="13" to="15"/>
    <inkml:traceView traceRef="/example.inkml" from="16" to="19"/>
    <inkml:traceView traceRef="/example.inkml" from="20" to="23"/>
  </hwTraces>
</hLevel>
```

The run-time data structure of a Radical is a collection stroke instances. All values for recognition of a Radical are computed from the strokes instances by using different permutations of the strokes to match a database entry. A more detailed description of the Radical recognition process will be given in section (1.6).

1.2.3.4 Character Data Format

The character data format is an ordered list of radicals, according to their standard sequence of writing within the character. In listing (1.9) the character 東 is shown. It consists of the radicals 日 and 木. The storage data structure for characters in general is totally straightforward and exploits the advantages of the UPX format. In the concrete example, the outermost *hLevel* element is used for the character and additional descripitons, the two inner *hLevel* elements are used for the two Radicals of that character. Each *traceView* tag witihn the Radicals represents a stroke. Thus, the data structure implicitely encodes the stroke number of a Radical. Here, both the Radicals 日 and 木 have four strokes each, the Kanji character 東 is comprised of eight strokes in total.

The stroke order of 東 is shown in figure 1.1. Notice, that the Radicals of this Kanji are not drawn one after another. The very first stroke is a part of the 木-Radical, the four following strokes constitute the complete 日-Radical and the last three strokes, again, are part of the 木-Radical. The UPX structure shown in listing (1.9) accounts for that by not specifying a total order of Radicals, but rather a preferred order. The traces are spatialised to their Radical, the order of drawing does effect the InkML traces and ultimately the recognition process, but not the UPX data structure.

Listing 1.9: A character representation in UPX

```
<hLevel level="character" id="CHARACTER71">
<!-- The character 71 consists of the Radicals 72 and 75 -->
  <label id="CHARACTER71label" labelSrcRef="labelref_DW" labelType="truth">
    <alternate>U+6771</alternate><!-- Unicode value -->
  </label>

  <hLevel level="radical" id="RADICAL72">
  <!-- The first Radical in this Kanji character of handwriting. -->
    <label id="RADICAL72label"
           labelSrcRef="labelref_DW"
           labelType="truth">
      <alternate>U+65E5</alternate><!-- Unicode value -->
    </label>
    <hwTraces>
      <inkml:traceView traceRef="/example.inkml" from="0" to="3"/>
      <inkml:traceView traceRef="/example.inkml" from="4" to="7"/>
      <inkml:traceView traceRef="/example.inkml" from="8" to="9"/>
      <inkml:traceView traceRef="/example.inkml" from="10" to="11"/>
    </hwTraces>
  </hLevel>

  <hLevel level="radical" id="RADICAL75">
  <!-- The second Radical in this Kanji character of handwriting. -->
    <label id="RADICAL75label"
           labelSrcRef="labelref_DW"
           labelType="truth">
      <alternate>U+6728</alternate><!-- Unicode value -->
    </label>
    <hwTraces>
      <inkml:traceView traceRef="/example.inkml" from="12" to="12"/>
      <inkml:traceView traceRef="/example.inkml" from="13" to="15"/>
      <inkml:traceView traceRef="/example.inkml" from="16" to="19"/>
      <inkml:traceView traceRef="/example.inkml" from="20" to="23"/>
    </hwTraces>
  </hLevel>
</hLevel>
```

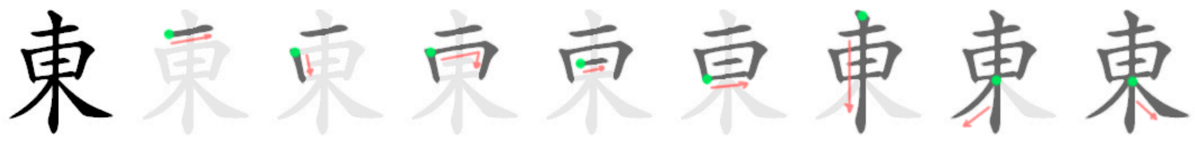

Figure 1.1: Stroke order of the Kanji 東

1.3 Database

1.3.1 Database Organisation

The database used in the system consits of two parts. The first part is centered around handwriting information about characters and their substructures, he second part is concerned with lexical information only. The **handwriting part** is a UPX file structure that holds character descriptions, Kanji character unicode values, as well as Radical descriptions and pointers to InkML files holding pen trajectories representing the data structures given in the UPX file. The **lexical part** is an XML file in a format described by **?** (?). The XML file contains lexical information about characters, such as their key Radicals, their Readings, and their translation into a number of languages. The handwriting part of the database has been discussed in section (1.2.3). The storage data structure layed out there is the structure used. The lexical part is provided[1] in an XML text file.

Listing (1.10) shows a sample database entry of the multi-index database created by **?** (?). The *character* element is the root of an entry. The *literal* tag holds the digital form of the character. Under the *codepoint* tag, different code points can be stored, in the current version it is Unicode and JIS0208 (?; ?). The *radical* element holds the number of the key radical of the character. In the current example for the character 東 the key radical is 木, which is number 75 in the classical Radical index. Miscellaneous information that may help a human finding the character in a paper-based dictionary are stored in the *misc* element. The file format provides room for the character index in different paper-based dictionaries and study books in the *dic_number* element. For example the *dic_ref* with the attribute *dr_type="sh_kk"* holds the character index in (?). This information can be very useful for a learner when referring to study books. The *reading_meaning* element holds information about the pronunciation of the character and translations into a number of different languages. The *nanori* elements describe additional Japanese readings that occur only in person- and place names.

With this organisational approach it is possible for different parties to work on different information structures simultaneously. These can be accessed, modified and improved separately. This is due to the encapsulation of the different data structures.

Listing 1.10: Sample lexicon entry for lexical data

```
<!-- Entry for Kanji: 東 -->
<character>
  <literal>東</literal>
  <codepoint>
    <cp_value cp_type="ucs">6771</cp_value>
    <cp_value cp_type="jis208">37-76</cp_value>
  </codepoint>
```

[1]Kanjidict2 is freely available from `http://www.csse.monash.edu.au/~jwb/kanjidic2/`

```
<radical>
  <rad_value rad_type="classical">75</rad_value>
  <rad_value rad_type="nelson_c">4</rad_value>
</radical>
<misc>
  <grade>2</grade>
  <stroke_count>8</stroke_count>
  <freq>37</freq>
  <jlpt>4</jlpt>
</misc>
<dic_number>
  <dic_ref dr_type="nelson_c">213</dic_ref>
  <dic_ref dr_type="nelson_n">2596</dic_ref>
  <dic_ref dr_type="halpern_njecd">3568</dic_ref>
  <dic_ref dr_type="halpern_kkld">2221</dic_ref>
  <dic_ref dr_type="heisig">504</dic_ref>
  <dic_ref dr_type="gakken">11</dic_ref>
  <dic_ref dr_type="oneill_names">771</dic_ref>
  <dic_ref dr_type="oneill_kk">27</dic_ref>
  <dic_ref dr_type="moro" m_vol="6" m_page="0172">14499</dic_ref>
  <dic_ref dr_type="henshall">184</dic_ref>
  <dic_ref dr_type="sh_kk">71</dic_ref>
  <dic_ref dr_type="sakade">121</dic_ref>
  <dic_ref dr_type="jf_cards">63</dic_ref>
  <dic_ref dr_type="henshall3">201</dic_ref>
  <dic_ref dr_type="tutt_cards">164</dic_ref>
  <dic_ref dr_type="crowley">108</dic_ref>
  <dic_ref dr_type="kanji_in_context">39</dic_ref>
  <dic_ref dr_type="busy_people">2.10</dic_ref>
  <dic_ref dr_type="kodansha_compact">1053</dic_ref>
  <dic_ref dr_type="maniette">516</dic_ref>
</dic_number>
<query_code>
  <q_code qc_type="skip">4-8-3</q_code>
  <q_code qc_type="sh_desc">0a8.9</q_code>
  <q_code qc_type="four_corner">5090.6</q_code>
  <q_code qc_type="deroo">1564</q_code>
</query_code>
<reading_meaning>
  <rmgroup>
    <reading r_type="pinyin">dong1</reading>
    <reading r_type="korean_r">dong</reading>
    <reading r_type="korean_h">동</reading>
    <reading r_type="ja_on">トウ</reading>
    <reading r_type="ja_kun">ひがし</reading>
    <meaning>east</meaning>
    <meaning m_lang="fr">Est</meaning>
    <meaning m_lang="es">Este</meaning>
    <meaning m_lang="pt">Oriente</meaning>
  </rmgroup>
  <nanori>あい</nanori>
  <nanori>あがり</nanori>
  <nanori>あずま</nanori>
  <nanori>あづま</nanori>
  <nanori>こ</nanori>
  <nanori>さき</nanori>
```

```
      <nanori>しの</nanori>
      <nanori>とお</nanori>
      <nanori>はる</nanori>
      <nanori>ひが</nanori>
      <nanori>もと</nanori>
    </reading_meaning>
  </character>
```

1.3.2 Alternative Database Structures

The structure and organisation of the databases as it is now, seems optimal among several alternatives. The reason for that is that the data is stored in separate and self-containd units. The indexing refeencing between the different structures works seamless via the unicode codepoints. Another adavantage of using the available formats. KanjiDict2 is work in progress and updates may be available for other languages or for additional characters. Using the KanjiDict2 the way it is allows for reliablity. The same is true for for UPX and InkML: UNIPEN has been the de facto standard for handwriting data exchange for a long time and UPX is considered the official successor of UNIPEN and InkML is a W3C recommendation[2].

1.3.2.1 A Unified File Format

An alternative solution to the split information approach would be to establish a unified file format for the pen trajectories, the UPX metadata and the lexical information. The advantage of that approach is that all data would be in one single place. However, a number of disadvantages would emerge from that approach, too. The file formats, InkML, UPX and KanjiDict2 already exist. Creating a new file format based on those would on one hand not create much additional value, but on the other hand lead to not using available de facto standards. Every time, a new version of the lexical data is issued or every time one would want to update the gold standard pen trajectory of a character, the formats would have to be linked together.

1.3.2.2 Relational Database

Establishing a relational database to hold the different data structures sounds like a feasible plan. With consideration of the three formats InkML, UPX and KanjiDict2 it should be possible to design a relational database that holds the different information in a structured manner. However, again the same problem arises. Updates and extensions of the formtats of the data would create the need for a conversion tool. If the XML specification of one of the formats changes, the table structure in the relational database would have to be adapted, too. Alternatively, the actual XML code could be stored in relational database tables. That approach however seem awkward. XML is a format for structuring data in human readable text files. All three formats, InkML, UPX and KanjiDict2 have been designed to be human-editable and therefore accessible without any special software or tools but a text editor. Storing these XML structures in a relational database would compromise the advantage of a relational database that lies in querying and would also make editing the data a more complex task. Additionally, the application is a desktop application for a single user environment. Depending on what relational database was used that could force shipment of the database software along with the Kanji learning system. The conclusion is that once InkML and UPX have been standardised and are therefore more reliable formats it could be benefitial to design a relational database structure that mirrors these formats. Then, the relational database could be used in an online environment, where a server application performs the recognition for a web client. For the time being and the technical equipment given it does not seem useful to employ a relational database.

[2]However, until the end of 2009 InkML has not been standardised by the W3C and UPX is still in a planning stage concerning standardisation.

1.4 Recognition Architecture

1.5 Stroke Recognition Process

1.5.1 Advanced Point Lists

what happens to the points? nothing, really - the magic happens when normalisation and the other stuff starts. why this section? what's the purpose? oh, right - angles and vectors instead of simple points. from one point to the next, or rather from on point to ten points down the line, to get a rougher direction. vectors make it interesting. impacts on curve handling! gradient and stuff can be measured in the vector representation (even without any boxes) making the point list a cool mathematical object! show code samples in pseudocode if necessary. report about the cool stuff.

what's the similarity measure for points and strokes? show requirements. what alternatives were there to consider?

1.5.2 Normalisation

Many handwriting recognition systems perform some kind of normalisation. See section **??** for a review of common techniques. In this system, only size normalisation is performed. Other normalisation techniques reported in literature were often necessary, because the input devices returned unsufficient data. With modern input devices many of the normalisation techniques that were used to clean the data from different kinds of noise are not necessary.

In order not to lose any data, the complete list of points is maintained, there is no point reduction. Since the prototype is a learning application and not a pure handwriting recognition, the main issue seems the size of the character. The size normalisation is done in two steps. Each complex that is more complex than a point has a bounding box around that can be used for scaling.

1.5.2.1 Boxing

The term *bounding box* needs not much introduction. It is the smallest box surrounding a graphic object. In this prototype, bounding boxes are rectangles with the same height and width. Rectangles are defined in this prototyp as a handle point, height and a width. values 'x' and 'y' as coordinates, 'h' is the y-height of the rectangle, 'w' is the x-width. In order to find the square bounding box of any sequence of points, a method finds the four outermost points 'C' with the minimal and maximal values for 'x' and 'y'. The handle point 'P' and the width and height of the bounding box 'B' can be calculated as follows.
$C_{xmin} :=< x_{min}, y_1 >$
$C_{xmax} :=< x_{max}, y_2 >$
$C_{ymin} :=< x_1, y_{min} >$
$C_{ymax} :=< x_2, y_{max} >$
$P_{intermediate} :=< x_{min}, y_{min} >$
$w_{intermediate} := x_{max} - x_{min}$
$h_{intermediate} := y_{max} - y_{min}$
$s := max(w_{intermediate}, h_{intermediate})$
$$P := \begin{cases} < x, y > & \text{if } h_{intermediate} \text{ is greater than } w_{intermediate} \\ -(n+1)/2 & \text{if } n \text{ is odd} \end{cases}$$
The other are always

how is boxing done? show requirements. what alternatives were there to consider? is it useful to have a similarity measure for bounding boxes? yes! but why? explain! size of the boxes! - think of characters that only have two strokes.

1.5.2.2 Scaling

s. 42-45 how is scaling done? show requirements. what alternatives were there to consider?

1.5.3 Curve Handling

see **??**

S 14, 16, 17 how is curver handling done? show requirements. what alternatives were there to consider?

stroke matching with angles instead of point position. s. 24

1.5.4 Dynamic Time Warping

what's the similarity measure for points and strokes? show requirements. what alternatives were there to consider?

s. 51 how is dynamic time warping done here? pointer to papers or hwr - chapter, don't explain DTW here. show requirements why DTW? what alternatives were there to consider? none - it is the alternative. to all the other stuff I've been doing. however, what about 3D time warping?

1.6 Radical Recognition Process

1.7 Character Recognition Process

1.8 Error Handling

see section **??** in chapter **??** for possible sources of error

1.8.1 Error Recognition

why this section? to demonstrate own achievements of error recognition. the reader should know how it is done technically.

what goes into this section? the aspects of finding errors. finding errors is not a straightforward trivial task - whenever something does not match it is an error - doesn't work like that. instead, firstly, it needs to be made sure that it actually is an error. meaning - not a recognition error, but a user error. secondly, the type of error needs be identified. see section **??** (or handwritten page 58) for sources of error.

how will this section be written? technical - first describe how the error recognition integrates into the recognition process, then how errors are identified.

1.8.2 Error Processing

why this section? actually the 'handling' or 'processing' aspect could be described in the recognition section 1.8.1 as well. so this section is only for a better overview, for document structure, thematically they are the same section. thus they are put together under Error Handling 1.8.

what goes into this section?