Functional Arabic Morphology
Formal System and Implementation

Otakar Smrž

Doctoral Thesis

Prague 2006

INSTITUTE OF FORMAL AND APPLIED LINGUISTICS
FACULTY OF MATHEMATICS AND PHYSICS
CHARLES UNIVERSITY IN PRAGUE
Abstract

We present an implementation of a Haskell library for processing the Arabic language in the ArabTgX transliteration (Lagally, 2004), a non-trivial and multi-purpose notation for encoding Arabic orthographies and phonetic transcriptions in parallel. Our approach relies on the Pure Functional Parsing library developed in (Ljunglöf, 2002), which we accommodate to our problem and partly extend. In the general view, we describe two alternative algorithms for longest-match deterministic parsing and rewriting, present the monadic-style grammars formalizing the relation of the script and the sound in Arabic, and promote modular design in systems for modeling or processing natural languages.
Preface

The world is an ocean.

Barbora Vidová Hladká, Jan Hajič, Khalil Sima’an, Nizar Habash, Ali El Dada
Eva Hajičová, Eva Panevová, Petr Sgall
Mark Liberman, Chris Cieri, Mohamed Maamouri, Ann Bies, Wigdan Mekki, Dalila Tabessi
Benjamin Pierce
Tim Buckwalter
Petr Zemánek and the PADT team
 Jiří Fleissig, Charif Bahbouh
Zdeněk Žabokrtský, Petr Pajas
Enjoy!
Contents

1 Introduction .......................................................... 7

2 Writing & Reading Arabic ........................................... 8
   2.1 Orthography and Buckwalter Transliteration ............... 8
   2.2 ArabTEX Notation ............................................. 9
      2.2.1 Standard notation ..................................... 11
      2.2.2 Extended notation .................................... 16
   2.3 Buckwalter Transliteration .................................. 16
   2.4 Notation of ArabTEX ......................................... 16

3 Grammatical Categories .......................................... 17
   3.1 Functional Arabic Morphology ............................... 17
      3.1.1 The Tokenization Problem ............................ 17
      3.1.2 Existing Morphological Systems ..................... 18
      3.1.3 Functional and Illusory Categories .................. 19
      3.1.4 Approximating the Functional Model ................. 22

4 Impressive Haskell ................................................ 25
   4.1 Encode Arabic in Haskell .................................. 25

5 Deterministic Parsing .............................................. 26
   5.1 Encode Mapper in Perl ...................................... 26
   5.2 Encode Arabic in Perl ..................................... 32

6 Encode Arabic ...................................................... 33
   6.1 Encode Arabic in Haskell .................................. 33
   6.2 Introduction ................................................ 33
   6.3 Functional Parsing ......................................... 34
   6.4 Encode Mapper ............................................... 36
      6.4.1 Longest Match Insight ............................... 39
      6.4.2 Breadth-First Algorithm ............................. 41
      6.4.3 Depth-First Algorithm ............................... 45
   6.5 Encode Extend .............................................. 47
# CONTENTS

6.6 Encode Arabic .......................................................... 50  
6.6.1 Arab\TeX Encoding Concept ..................................... 51  
6.6.2 Encode Arabic Arab\TeX ........................................... 52  
6.6.3 Encode Arabic Arab\TeX ZDMG ................................... 59  
6.7 Discussion ............................................................... 61  
6.8 Conclusion ............................................................... 61  

7 Functional Morphology .................................................. 66  
7.1 On Hamza Writing ...................................................... 66  
7.2 Error Messages .......................................................... 67  

8 Lexicon versus Treebank ................................................. 70  
8.1 Functional Description of Language ................................. 70  
8.2 Analytical Dependency Description ................................. 73  

9 MorphoTrees ............................................................... 79  
9.1 The MorphoTrees Hierarchy .......................................... 79  
9.2 MorphoTrees Disambiguation in TrEd ............................... 82  
9.3 Discussion to MorphoTrees .......................................... 83  

A Other Listings ............................................................. 87
List of Tables
## List of Figures

2.1 Letters of Arabic orthography ........................................ 10  
3.1 Tokenization of input strings ....................................... 18  
3.2 Functional and illusory categories versus agreement .......... 22  
5.1 Simple Mapper for the rules without lookahead. ............... 28  
5.2 Tricky Mapper for the rules including lookahead. ............ 30  
6.1 Trie structures illustrating the longest match algorithms. ..... 40  
6.2 Interpreting the \texttt{[Char]} input in the Arab\TeX{} notation. 52  
8.1 Analysis of the sentence ‘It was not easy for him…’ .......... 74  
8.2 Analysis of the sentence ‘And according to first statistics,…’ 75  
8.3 Analysis of clausal structure of the example text (see page 73). 76  
8.4 Analysis of coordinations. ....................................... 77  
9.1 Analyses of \texttt{AlY} transformed into the MorphoTrees hierarchy. 80  
9.2 Annotation of MorphoTrees using restrictions on particular values/categories in the morphological tags. 81  
9.3 Discussion of partitioning and tokenization of input strings. 83
Chapter 1

Introduction

In this thesis, we are going to develop a computational model of the morphological processes in Arabic. With this system, we will become able to derive and inflect words, as well as recognize word forms and analyze their grammatical functions.

The approach to building our morphological model will strive to be comprehensive with respect to linguistic generalization, and high-level and modern with respect to the programming techniques that we will employ. We will promote its flexibility, reusability and extensibility. We will describe the linguistic concept and will try to implement it in a very similar way, avoiding tweaking of the concepts due to temporary technical limitations.

(Beesley, 2001, Beesley and Karttunen, 2003)
(Kiraz, 2001, Walther, 2002)
(Buckwalter, 2002, 2004a)
(Habash and Rambow, 2006, Habash et al., 2005)

I would not have made this so long except that I do not have the leisure to make it shorter.

Blaise Pascal in Lettres Provinciales
Chapter 2

Writing & Reading Arabic

In the linguistic context, morphology is the study of word forms. In formal language theory, the symbols for representing words are an inseparable part of the definition of the language. In human languages, the concept is a little different—an utterance can have multiple representations, depending on the means of communication and the conventions for recording it.

The scope of our computational morphological model should not be limited to texts written in one customary orthography, for reasons that we will discuss later. This chapter will explore the interplay between the writing system and the phonetic transcription of Arabic. We will introduce in detail the ArabTEX notation (Lagally, 2004), a morphophonemic transliteration scheme that we will use as the representation of choice for our general-purpose morphological model.

Following an excellent article on this issue (Beesley, 1997, 1998), let us summarize our understanding of the present terms by emphasizing these characteristics:

- **orthography** set of conventions for representing a language using an associated set of symbols
- **transcription** alternative, phonetically or phonologically motivated representation of the language, possibly romanization
- **transliteration** orthography with carefully substituted symbols, still preserving the original orthographic conventions
- **encoding** transliteration mapping the orthographic symbols into numbers implemented as characters or bytes

2.1 Orthography and Buckwalter Transliteration

The standard Arabic orthography is based on the Arabic script. It has an alphabet of over 30 letters written from right to left in a cursive manner, and of about 10 other symbols written optionally as diacritics above or below the letters.
Phonologically, Arabic includes 28 consonants that are evenly divided into two
groups according to their potential to assimilate with the definite article \textit{al-}. They are
called solar consonants \textit{al-ḥurūf aš-šamsīya} and lunar consonants \textit{al-ḥurūf al-qamarīya} in
the linguistic tradition.

There are only six vocalic phonemes in Modern Standard Arabic, namely the short
\(a, i, u\) and the long \(ā, ī, ū\). In the dialects, the range of vowels is extended, but the
script does not have any extra graphemes for encoding them. On the other hand, the
diacritics also include symbols indicating an absence of a vowel after a consonant, or
marking the indefinite article \(-n\), which combines with the vocalic endings into distinct
orthographic symbols.

Several letters of the script serve as allographs representing the single phoneme
\textit{hamza}, the glottal stop. Other allographs are motivated by morphophonemic changes
or the historical development of the language and the script (Fischer, 2001, Holes, 2004,
pages 3–34, resp. 89–95).

The set of letters is shown in Figure 2.1. The survey of conventions for using them
and interleaving them with the diacritics will emerge shortly as a side-effect of our
describing the Arab\TeX notation.

Buckwalter transliteration (Buckwalter, 2002, 2004b) is a lossless romanization of
the contemporary Arabic script, and is a one-to-one mapping between the relevant
Unicode code points for Arabic and lower ASCII characters. It is part of Figure 2.1.\footnote{Buckwalter transliteration will be displayed in the \textit{upright} typewriter font, whereas the Arab\TeX notation will use the \textit{italic} typewriter shape.}

\section*{2.2 Arab\TeX Notation}

The Arab\TeX typesetting system (Lagally, 2004) defines its own Arabic script meta-
encoding that covers both contemporary and historical orthography to an exceptional
extent. The notation is human-readable as well as very natural to learn to write with.
Its design is inspired by the standard phonetic transcription of Arabic, which it mimics,
yet some distinctions are introduced to make the conversion to the original script or the
transcription unambiguous.

Unlike other transliteration concepts based on the strict one-to-one substitution of
graphemes, Arab\TeX interprets the input characters in context in order to get the proper
meaning. Finding the glyphs of letters (initial, medial, final, isolated) and their liga-
tures is not the issue of encoding, but of visualizing only. Nonetheless, definite article
assimilation, inference of \textit{hamza} carriers and silent \textit{alifs}, treatment of auxiliary vowels,
optional quoting of diacritics or capitalization, resolution of notational variants, and
mode-dependent processing are the challenges for parsing the notation successfully.

Arab\TeX’s implementation is documented in (Lagally, 1992), but the parsing al-
gorithm for the notation has not been published. The \TeX code of it is organized into
### Lunar Consonants

<table>
<thead>
<tr>
<th>Lunar Consonant</th>
<th>Solar Consonant</th>
</tr>
</thead>
<tbody>
<tr>
<td>hamza</td>
<td>t</td>
</tr>
<tr>
<td>b</td>
<td>t</td>
</tr>
<tr>
<td>ġ</td>
<td>d</td>
</tr>
<tr>
<td>ḥ</td>
<td>d</td>
</tr>
<tr>
<td>ḫ</td>
<td>r</td>
</tr>
<tr>
<td>y</td>
<td>r</td>
</tr>
<tr>
<td>ayn</td>
<td>z</td>
</tr>
<tr>
<td>ġ</td>
<td>s</td>
</tr>
<tr>
<td>f</td>
<td>s</td>
</tr>
<tr>
<td>q</td>
<td>s</td>
</tr>
<tr>
<td>k</td>
<td>d</td>
</tr>
<tr>
<td>m</td>
<td>t</td>
</tr>
<tr>
<td>h</td>
<td>z</td>
</tr>
<tr>
<td>w</td>
<td>l</td>
</tr>
<tr>
<td>y</td>
<td>n</td>
</tr>
</tbody>
</table>

### Variants of ḍalif

<table>
<thead>
<tr>
<th>ḍalif (ā)</th>
<th>A  A  1</th>
</tr>
</thead>
<tbody>
<tr>
<td>wasila</td>
<td>ā</td>
</tr>
</tbody>
</table>

### Suffix-only Letters

<table>
<thead>
<tr>
<th>ḍalif maqṣūra (ā)</th>
<th>Y  Y  ʔ</th>
</tr>
</thead>
<tbody>
<tr>
<td>tā· marbāṭa (t/h)</td>
<td>T  P</td>
</tr>
</tbody>
</table>

### Variants of ḍamza

<table>
<thead>
<tr>
<th>madda</th>
<th>ā</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ġ</td>
<td>ā</td>
<td>1</td>
</tr>
<tr>
<td>ḫ</td>
<td>ā</td>
<td>1</td>
</tr>
<tr>
<td>ḫ</td>
<td>ā</td>
<td>1</td>
</tr>
<tr>
<td>ḫ</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Non-Arabic Consonants

<table>
<thead>
<tr>
<th>ḍalif (ā)</th>
<th>A  A  1</th>
</tr>
</thead>
<tbody>
<tr>
<td>wasila</td>
<td>ā</td>
</tr>
<tr>
<td>tā· marbāṭa (t/h)</td>
<td>T  P</td>
</tr>
</tbody>
</table>

Figure 2.1 The letters ḍurūf خِرُوف of the Arabic orthography (extended with graphemes for some non-Arabic consonants) and their corresponding Buckwalter transliteration, ArabTEX notation (in the mode with explicit ḍamza carriers), and phonetic transcription, listed in the right-to-left order.
2.2. ArabTeX Notation

deterministic-parsing macros, yet the complexity of the whole system makes consistent modifications or extensions by other users very difficult, if not impossible.

We are going to describe our own implementations of the interpreter in Chapter 6, where we will show how to decode the notation and its proposed extensions. To encode the Arabic script or its phonetic transcription into the ArabTeX notation requires some heuristics, if we want to achieve linguistically appropriate results.

2.2.1 Standard notation

Let us first deal with the notational conventions that the current version of ArabTeX supports (Lagally, 2004).

Our explanation will take the perspective of a child learning how the sounds are represented in the writing, rather than of a calligrapher trying to encrypt the individual graphemes into the notation. This may bring some difficulty to those who think of the language primarily through orthography, and not phonology.

Phonemes The notation for consonants is listed in Figure 2.1. Short vowels are coded as a, i and u, the long ones A, I and U. We concatenate consonants and vowels in their natural order.

\[
\begin{align*}
\text{darasa} & \quad \text{darasa} & \text{Dar} & \quad \text{darasa} & \quad \text{‘he studied’} \\
\text{sAfarat} & \quad \text{sāfarat} & \text{Ṣāf} & \quad \text{saAfarato} & \quad \text{‘she travelled’} \\
\text{ya`glisUna} & \quad \text{yāglisāna} & \text{Yāglīs} & \quad \text{yajolisuwna} & \quad \text{‘they sit’} \\
\text{kitAbuhA} & \quad \text{kitābūha} & \text{Kīt} & \quad \text{kitaAbuhaA} & \quad \text{‘her book’} \\
.\text{hAsUbI} & \quad \text{ḥāsūbī} & \text{ḥAs} & \quad \text{HaAsuwbiy} & \quad \text{‘my computer’}
\end{align*}
\]

Long vowels produce a combination of a diacritic and a letter in the script. Doubling of a consonant is indicated with the ṣadda ᵃ̂ diacritic, while no vowel after a consonant results in the sukūn ٷ. These rules interoperate, so Ṣan and ɬ can often, even though not always, behave like ɬw and iy in the orthographic representation.

\[
\begin{align*}
.\text{sarra.ha} & \quad \text{sarrāḥa} & \text{Sār} & \quad \text{Sar`aHa} & \quad \text{‘he explained’} \\
`\text{gayyidUna} & \quad \text{gayyidāna} & \text{G āy} & \quad \text{jay`idwna} & \quad \text{‘good ones’} \\
\text{qawIyayni} & \quad \text{qawīyayni} & \text{Qawī} & \quad \text{qawi`ayoni} & \quad \text{‘two strong ones’} \\
`\text{adUwuhu} & \quad \text{adūwuhu} & \text{Adu} & \quad \text{Eaduw`uhu} & \quad \text{‘his enemy’} \\
\text{zuwwArunA} & \quad \text{zuwwārunā} & \text{Zuw} & \quad \text{zuw`aArunaA} & \quad \text{‘our visitors’} \\
\text{tuwAfiqu} & \quad \text{tuwāfiqu} & \text{Tuw} & \quad \text{tuwaAfiqu} & \quad \text{‘you agree’}
\end{align*}
\]

The consonant T and the long vowel Y can only appear as final letters, otherwise the former changes to ɬ and the latter to A or ay. Determination of the carrier for hamza is subject to complex rules, but phonologically, there is just one ɬ consonant.
 Articles  The definite article al– is connected by a hyphen with the word it modifies. If assimilation is to take place, either the word’s initial consonant is doubled, or the l of the article is replaced with that consonant directly.

\[
al-qamaru \quad al-qamaru \quad Aaloqamaru \quad \text{‘the moon’}
al-\¨s\¨samsu \quad a\¨-\¨s\¨samsu \quad Aal\¨-\¨amosu \quad \text{‘the sun’}
a\¨-\¨s\¨samsu \quad a\¨-\¨s\¨samsu \quad Aal\¨-\¨amosu \quad \text{‘the sun’}
al-lawnu \quad al-lawnu \quad Aall\¨-awonu \quad \text{‘the color’}
al-l\iawnu \quad al-l\iawnu \quad Aall\¨-awonu \quad \text{‘the color’}
al-\’al\iwAnu \quad al-\’al\iwanu \quad AaloOal\iowaAnu \quad \text{‘the colors’}
al-\’UIY \quad al-\’UIY \quad AaloOuw\'l\iY \quad \text{‘the first’}
\]

The indefinite article N must be distinguished by capitalization. Whether or not the orthography requires an additional silent alif, need not be indicated explicitly.

\[
baytun \quad baytun \quad bayotN \quad \text{‘a house nom.’}
baytIN \quad baytIN \quad bayotK \quad \text{‘a house gen.’}
baytAN \quad baytAN \quad bayotFA \quad \text{‘a house acc.’}
madiNaTuN \quad madiNatun \quad madiynapN \quad \text{‘a city nom.’}
madiNaTiN \quad madiNatIn \quad madiynapK \quad \text{‘a city gen.’}
madiNaTaN \quad madiNatAn \quad madiynapF \quad \text{‘a city acc.’}
\]

It is, however, possible to enforce a silent prolonging letter after an indefinite article. Most notably, it is used for the phonologically motivated ending aNY.

\[
siwaNY \quad siwAN \quad siwFY \quad \text{‘equality’}
siwaNY \quad siwAN \quad siwFY \quad \text{‘equality’}
\]

Revision of
August 7, 2006
2.2. ArabTeX Notation

Extras The silent alif also appears at the end of some verbal forms. It is coded UA if representing ۳, and AW if standing for aw.

\[\begin{align*}
\text{katabU} & \quad \text{katabu} & \quad \text{katabuU} & \quad \text{‘they wrote’} \\
\text{ya.sIrU} & \quad \text{yaṣiru} & \quad \text{yaSiyruU} & \quad \text{‘that they become’} \\
\text{da’aWA} & \quad \text{dawu} & \quad \text{daEawO} & \quad \text{‘they called’} \\
\text{tatamannaW} & \quad \text{tatamanna} & \quad \text{tatamanawO} & \quad \text{‘that you wish pl.’} \\
\text{raW} & \quad \text{raw} & \quad \text{rawO} & \quad \text{‘do see pl.’} \\
\text{insaW} & \quad \text{insaw} & \quad \text{AinosawO} & \quad \text{‘do forget pl.’}
\end{align*}\]

The phonological auxiliary vowels that are prefixed are preserved in the notation, yet, they can be elided in speech or turned into was.la in the script. The auxiliary vowels that are suffixed can be marked as such by a hyphen, if one prefers so.

\[\begin{align*}
\_\text{dawU} & \quad \text{a}^\text{s}-\text{s}^\text{a}‘n\text{i} & \quad \text{dawu}^\text{š-šani} & \quad \text{*awu} & \quad \{1\}^\text{l}-\text{aOoni} & \quad \text{ad a.} \\
\text{qala} & \quad \text{i}^\text{yqa.z} & \quad \text{qa}‘\text{la}‘\text{yqaz} & \quad \text{qa}‘\text{la} & \quad \{\text{yoqaZo}\} & \quad \text{ad b.} \\
\text{‘an-i} & \quad \text{ismI} & \quad \text{un-i}‘\text{smI} & \quad \text{Eani} & \quad \{\text{somiy}\} & \quad \text{ad c.} \\
\text{al-i-‘gtimA’u} & \quad \text{al-i-‘gtim}‘\text{u} & \quad \text{Aali{\text{jotimaA}E}u} & \quad \text{ad d.}
\end{align*}\]

\[\begin{align*}
\text{a.} & \quad \text{‘those concerned’} & \quad \text{b.} & \quad \text{‘he said wake up’} & \quad \text{c.} & \quad \text{‘about my name’} & \quad \text{d.} & \quad \text{‘the society’}
\end{align*}\]

The defective writing of the long ā is _a, of the short u it is _U.

\[\begin{align*}
\text{h_a_dA} & \quad \text{hādā} & \quad \text{h}‘\text{aA} & \quad \text{‘this’} \\
\text{h_a’lika} & \quad \text{dālika} & \quad \text{h}‘\text{lika} & \quad \text{‘that’} \\
\text{h_a’ulA’i} & \quad \text{haulāi} & \quad \text{h}‘\text{WulaA}‘\text{i} & \quad \text{‘these’} \\
\text{’U_l_a’ika} & \quad \text{ulāika} & \quad \text{Ouwl}‘\text{ika} & \quad \text{‘those’}
\end{align*}\]

Other historical writings of long vowels can also be expressed in the standard notation. The description of _i, _u, _A, _I, _U, as well as _aU, _aI, _aY, is given in (Lagally, 2004, Fischer, 2001), but is irrelevant for our present interests.

The dialectal pronunciation of vowels, namely e, ē and o, ū, can be reflected in the phonetic transcription only. In orthography, this makes no difference and the vowels are rendered as i, ī and u, ū, respectively. A more complex solution to the dialectal phonology will be given later in this thesis.

\[\begin{align*}
\text{sOha}_‘\text{g}” & \quad \text{sōha} & \quad \text{suwhaj} & \quad \text{‘Sohag’} \\
\text{as-suweS”} & \quad \text{as-suweis} & \quad \text{Aals}‘\text{uwiys} & \quad \text{‘Suez’} \\
\text{homs}” & \quad \text{homṣ} & \quad \text{HumoS} & \quad \text{‘Homs’} \\
\text{hosni”} & \quad \text{hosnī} & \quad \text{Husoniys} & \quad \text{‘Hosni’} \\
\text{’omar”} & \quad \text{omar} & \quad \text{Eumar} & \quad \text{‘Omar’} \\
\text{’omAn”} & \quad \text{omān} & \quad \text{EumaAn} & \quad \text{‘Oman’} \\
\text{al-ma.greb”} & \quad \text{al-mağreb} & \quad \text{Aalomagorib} & \quad \text{‘Maghreb’}
\end{align*}\]
2.2. ArabTex Notation

There are some other symbols in the notation that allow us to encode more information than is actually displayed in the script or its transcription. One of those is "suppressing, in the examples above, the printing of a sukūn. Another is |, the invisible consonant useful in some tricky situations, and finally $, the tatwil, a filler for stretching the adjacent letters apart in the cursive script.

Words Due to the minute form of certain lexical words, the Arabic grammar has developed a convention to join them to the ones that follow or precede, thus making the whole concatenation a single orthographic word.

Although by any criteria separate words, wa ‘and’, fa ‘so’, bi ‘in, by, with’ and li ‘to, for’ are written as if they were part of the word that follows them.

Functionally similar words that are “heavier” monosyllables or bisyllabic, for example, aw ‘or’, fī ‘in’, ala ‘on’, are not so written. (Holes, 2004, p. 92)

This concatenation rule applies further to prefixed $a ‘by oath particle’, sa ‘will future marker’, ka ‘like, as’, la ‘emph. part.’, as well as to suffixed personal pronouns in genitive and accusative, and variably to mā, ma ‘what’ (Fischer, 2001, p. 14).

The ArabTex notation suggests that such words be hyphenated if prefixed and merely annexed if suffixed, to ensure proper inference of hamza carriers and $awslas.

\[
\begin{align*}
\text{bi-bu.t'iN} & \quad \text{bi-bu't'in} & \quad \text{bbuTo'K} & \quad \text{‘slowly’} \\
\text{bi-intibAhiN} & \quad \text{bi-'ntibahin} & \quad \text{bi{notibAhK} & \quad \text{‘carefully’} \\
\text{bi-al-qalami} & \quad \text{bi-‘l-qalami} & \quad \text{bi{loqalami} & \quad \text{‘with the pen’} \\
\text{fa-in.sarafa} & \quad \text{fa-‘ns.arafa} & \quad \text{fa{noSarafa} & \quad \text{‘then he departed’} \\
\text{ka-‘annanI} & \quad \text{ka-annan} & \quad \text{kaOan‘aniy} & \quad \text{‘as if I’} \\
\text{li-‘ArA’ihi} & \quad \text{li–‘rA{hi} & \quad \text{li|raA}ihi} & \quad \text{‘for his opinions’} \\
\text{sa-‘u.tIka} & \quad \text{sa-‘utika} & \quad \text{saOuEotiyka} & \quad \text{‘I will give you’} \\
\text{wa-lahu} & \quad \text{wa-lahu} & \quad \text{walahu} & \quad \text{‘and he has’} \\
\text{ka_-d_alika} & \quad \text{ka-‘dali} & \quad \text{ka‘lika} & \quad \text{‘as well, like that’} \\
\text{wa-fI-mA} & \quad \text{wa-fi-mA} & \quad \text{wafiymaA} & \quad \text{‘and in what’} \\
\end{align*}
\]

The cliticized $i and la must be treated with exceptional care as they interact with the definite article and the letter after it. In the current version of the ArabTex standard, this is not very intuitive, and we propose its improvement in the next subsection.

\[
\begin{align*}
\text{lil-‘asafi} & \quad \text{lil-asafi} & \quad \text{liloOasafi} & \quad \text{‘unfortunately’} \\
\text{lin-nawmi} & \quad \text{lin-nawmi} & \quad \text{li{l}n-awomi} & \quad \text{‘for the sleep’} \\
\text{li-llaylaTi} & \quad \text{li-llaylati} & \quad \text{li{lyolapi} & \quad \text{‘for the night’} \\
\text{li-ll_ahi} & \quad \text{li-lÀhi} & \quad \text{lil‘hi} & \quad \text{‘for God’} \\
\end{align*}
\]
2.2. ArabTeX Notation

Modes ArabTeX (Lagally, 2004) is an extension to the \TeX/LaTeX typesetting system that solves the problem of producing documents in the Arabic script, a writing system used with modifications by a variety of languages of the Orient.

One of the highlights of ArabTeX is its invention of an artful high-level notation for encoding the possibly multi-lingual text in a way that allows further interpretation by the computer program. In particular, the notation can be typeset in the original orthography of the language or in some kind of transcription, under one rendering convention or another. These options are controlled by setting the interpretation environment, and no change to the data is required.

vocalization degree is controlled by the following modes and commands

```
\fullvocalize
   bi-al-`arabIyaT  bi-`l-arabiyah  bi{loEarabiy`apo
   bi-al-`arabIyaTi bi-`l-arabiyati  bi{loEarabiy`api
   lu.gaTi al-`arab  luğati `l-arab  lugapi {loEarab
   lu.gaT al-`arab"  lugah al-arab  lugap {loEarab
```

```
\vocalize
   bi-al-`arabIyaT  bi-`l-arabiyah  bAlEarabiy`ap
```

```
\novocalize
   bi-al-`arabIyaT  bi-`l-arabiyah  bAlErby`p
```

language-dependent interpretation blabla

```
\setverb
\setarab
\setfarsi
\setmaghribi
```

phonetic transcription definition \settrans{standard} \settrans{zdmg} \settrans{english} \settrans{iranica} \settrans{farsi} \settrans{lazard} \settrans{urdu} \settrans{kashmiri} \settrans{turk} \cap

```
calligraphic options \oldtanwin \newtanwin \yahdots \yahnodots \accentshigh \accentslow \spreadtrue \spreadfalse
```
### 2.2.2 Extended notation

<table>
<thead>
<tr>
<th>Arabic</th>
<th>Buckwalker Transliteration</th>
<th>English</th>
</tr>
</thead>
<tbody>
<tr>
<td>al-la_dayni</td>
<td>al-la'dayni</td>
<td>those two'</td>
</tr>
<tr>
<td>li-l-la_dayni</td>
<td>li-l-la'dayni</td>
<td>for those two'</td>
</tr>
<tr>
<td>li-al-la_dayni</td>
<td>li-l-la'dayni</td>
<td>for those two'</td>
</tr>
<tr>
<td>bi-al-la_dayni</td>
<td>bi-l-la'dayni</td>
<td>with those two'</td>
</tr>
<tr>
<td>alla_dIna</td>
<td>alla'dina</td>
<td>those'</td>
</tr>
<tr>
<td>li-l-la_dIna</td>
<td>li-l-la'dina</td>
<td>for those'</td>
</tr>
<tr>
<td>li-al-la_dIna</td>
<td>li-l-la'dina</td>
<td>for those'</td>
</tr>
<tr>
<td>bi-al-la_dIna</td>
<td>bi-l-la'dina</td>
<td>with those'</td>
</tr>
</tbody>
</table>

### 2.3 Buckwalker Transliteration

يولد جميع الناس أُخِرَارًا متساوين في الكَرَامَة وَالْحَقَوْقِ. وقد وَهُبَوا عَقَلاً وَشُحْبًا وَعَلَّمُهُمَّ أَن

يَعْمَلُ بِعَضُّهُم بِعَضَّ يَرْحَمُ الآخَانِ.

yuwaladu jamiyEu (ln`aAsi GaHoraArFA mutasaAwIyna fiy | lokaraAma pi wa|loHuquwqi. waqado wuhibuW EaqolAF wDa miyrFA waEalayohimo Oano yuEaAmila baEoDuhumo baEoDFA biruwHi (16

يولد جميع الناس أُخِرَارًا متساوين في الكَرَامَة وَالْحَقَوْقِ. وقد وَهُبَوا عَقَلاً وَشُحْبًا وَعَلَّمُهُمَّ أَن

يَعْمَلُ بِعَضُّهُم بِعَضَّ يَرْحَمُ الآخَانِ.

Yūladu ʂamū `n-nāsi ʐhrāran mutasāwâna fi ℓ-kaɾāmiṯa wa-‘l-hashed. Wa-qad wuxibû ʐaqan wa-‘damirân wa-‘alayhîm an yuq̣mila baʃuhum baʃdän bi-rño ℓ-∫̣hāi.

\cap yUladu ʂamI’u an-naissance ‘a.hrAraN mutasAwIina fi al-karAmaTî wa-al-.huqUqi.
\cap wa-qad wuhibuUA ‘aqlaN wa-.damIraN wa-’alayhîm ’an yu’Amila ba’.duhum ba’.daN bi-rU hi al-‘i_hA’i.
3.1 Functional Arabic Morphology

Arabic is a language of rich morphology, both derivational and inflectional (Holes, 2004). Due to the fact that the Arabic script does usually not encode short vowels and omits some other important phonological distinctions, the degree of morphological ambiguity is very high.

3.1.1 The Tokenization Problem

In addition, Arabic orthography prescribes to concatenate certain word forms with the preceding or the following ones, possibly changing their spelling and not just leaving out the whitespace in between them. This convention makes the boundaries of syntactic units, which we need to retrieve as tokens for our deeper levels of analysis, obscure, for they may collapse with others into one compact string of letters and be no more the distinct ‘words’.

Tokenization is an issue in many languages. However, unlike in Chinese or German or Sanskrit (cf. Huet, 2003), in Arabic there are clear limits to the number and the kind of tokens that can combine in this manner.1 This may have lead to the prevalent interpretation that the clitics, including affixed pronouns or single-letter ‘particles’, are of the same nature and status as the derivational or inflectional affxes, and inferior to some central lexical morpheme, which yet need not exist . . .

We think about it differently. For treebanking, it is essential to determine the tokens of the studied discourse in order to provide the units for the syntactic annotation. Thus, it is nothing but these units that must be promoted to tokens and considered equal in this respect, irrelevant of how they are realized in writing.

To decide in general between pure morphological affxes and the critical run-on syntactic units, we use the criterion of substitutability of the latter by its synonym or analogy that can occur isolated. See the leftmost columns in Figure 3.1 for illustration of how input strings are tokenized in PADT—which contrasts to the style of the Penn

---

1 Even if such rules differ in the standard language and the various dialects.
3.1. Functional Arabic Morphology

<table>
<thead>
<tr>
<th>String</th>
<th>Token</th>
<th>Token Tag</th>
<th>Buckwalter’s Tags</th>
<th>Token Form</th>
<th>Token Gloss</th>
</tr>
</thead>
<tbody>
<tr>
<td>سيخبرهم</td>
<td>F-------</td>
<td>FUT</td>
<td></td>
<td>sa-</td>
<td>will</td>
</tr>
<tr>
<td>بذلك</td>
<td>P-------</td>
<td>PREP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>عن طريق</td>
<td>N--------</td>
<td>2R NOUN+CASE_DEF_GEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>والإنترنت</td>
<td>Z--------</td>
<td>2D DET+NOUN_PROP+</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>وخبرها</td>
<td>S--------</td>
<td>3FS2- POSS_PRON_3FS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>سيخبرهم</td>
<td>VI1A-3MS--</td>
<td>IV3MS+IV+IVSUFF_MOOD:1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>بذلك</td>
<td>S-------</td>
<td>3MP4-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>عن طريق</td>
<td>N--------</td>
<td>2D DET+NOUN+CASE_DEF_GEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ويقال</td>
<td>A-------</td>
<td>FS2D DET+ADJ+NSUFFIX_FEM_SG+CASE_DEF_GEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>عن طريق</td>
<td>N--------</td>
<td>2D DET+CASE_DEF_GEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>عن طريق</td>
<td>N--------</td>
<td>2R NOUN+CASE_DEF_GEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>عن طريق</td>
<td>N--------</td>
<td>2D DET+NOUN+CASE_DEF_GEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>عن طريق</td>
<td>N--------</td>
<td>2D DET+CASE_DEF_GEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>عن طريق</td>
<td>N--------</td>
<td>2D DET+NOUN+CASE_DEF_GEN</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3.1** Tokenization of input strings into tokens in *he will notify them about that through SMS messages, the Internet, and other means*, and the disambiguated morphological analyses, one per token (details in Sec. 3.1.4).

Arabic Treebank (examples in Maamouri and Bies, 2004).

In the Functional Arabic Morphology (Smrz, in prep.), tokenization of this kind is inherent in the very description of morphological processes, and with MorphoTrees (Smrz and Pajas, 2004), it even becomes one of the levels in morphological disambiguation, and one of the grounds for measuring distance between divergent analyses.

Discussions can be raised about the concrete aspects of tokenizing. Habash and Rambow (2005, section 7) correctly point out that “[t]here is not a single possible or obvious tokenization scheme: a tokenization scheme is an analytical tool devised by the researcher.” Different tokenizations will imply different amount of information, and some may be linguistically not as appropriate as others (cf. Bar-Haim et al., 2005, for the case of Hebrew). We will return to this topic in Section ?? on MorphoTrees.

### 3.1.2 Existing Morphological Systems

The long evolution of computational modeling of Arabic morphology is nowadays mirrored in the influential works of (Kiraz, 2001) and (Beesley and Karttunen, 2003), and although many morphological systems are in development (Ramsay and Mansur, 2001, Soudi et al., 2001, inter alia), only (Beesley, 2001) and (Buckwalter, 2002, 2004a) are actually accessible to the interested public, meeting the prerequisite to their wider application and evaluation.

It appears from the literature and implementations (many summarized in Al-Sughaiyer and Al-Kharashi, 2004) that Arabic computational morphology has understood its role
in the sense of operations with morphs rather than morphemes (cf. El-Sadany and Hashish, 1989), and has not concerned itself systematically and to the necessary extent with its role for syntax.\(^2\)

The outline of formal grammar in (Ditters, 2001), for instance, works with grammatical categories like number, gender, humanness, definiteness, but one cannot see which of the existing systems could provide for this information correctly, as they misinterpret some morphs for bearing a category, and underspecify lexical morphemes in general as to their intrinsic morphological functions.

Certain syntactic parsers, like (Othman et al., 2003), may resort to their own morphological analyzers, but still, they do not get rid of the form of an expression and only incidentally introduce truly functional categories (see Section 3.1.3). In syntactic considerations they often call for discriminative extra-linguistic features instead.\(^3\) Commercial systems, e.g. (Chalabi, 2004), do not seem to overcome this interference either.

The missing common rationale as to what higher linguistic framework the morphology should serve for crystalizes in the number of individual, ad hoc tag sets and a very rare discussion of their motivation, completeness, relevance and actual expressive power. Even when addressing the tokenization problem of word forms tied together in the Arabic script, authors do not recognize the crucial mutual differences in their solutions, and do not define precisely the objectives of their models.

### 3.1.3 Functional and Illusory Categories

While highlighting the structure of word forms and the derivational and inflectional processes of the language, even the best computational models of Arabic morphology (Beesley, 2001, Buckwalter, 2002, 2004a, Kiraz, 2001) never question in depth the information they provide with respect to the linguistic functions the word forms represent.

In our view, the task of morphology should be to analyze word forms of a language not only by finding their internal structure, i.e. recognizing morphs, but even by strictly discriminating their functions, i.e. providing the true morphemes. This doing in such a way that it should be completely sufficient to generate the word form that represents a lexical unit and features all grammatical categories (and structural components) required by context, purely from the information comprised in the analyses.

In this regard, we find supportive the argumentation in (Stump, 2001, chapter 1), where inferential–realizational theories of inflectional morphology are preferred to the lexical or incremental approaches. Issues like concatenative and non-concatenative

\(^2\) Versteegh (1997, chapter 6) describes the traditional Arabic understanding of *ṣurūf* morphology and *nabw* grammar, syntax, where morphology studied the derivation of isolated words, while their inflection in context was part of syntax.

\(^3\) Many people call those features *semantic* (cf. El-Shishiny, 1990), but we perceive them as *ontological*—our point is that those features are bound to some limited description of *reality*, and not to the *linguistic* meaning itself.
word formation, underdetermined morphosyntactic properties of words or, on the contrary, extended exponence of some grammatical categories, are very relevant to Arabic.

In morphological description of Arabic, the senses in which grammatical categories are used in the language (cf. Fischer, 2001, Badawi et al., 2004, Holes, 2004) get very often confused or not distinguished at all by the computational systems.

Functional Arabic Morphology (Smrz, in prep.) is designed to suit the grammatical modeling of the language better, and to give access to and exert the complete control over word forms. In particular, it revives the different grammatical senses and fixes them for the categories like number, gender, case, and definiteness, which we will explore here.

For number and gender, studying the important phenomenon of agreement classifies the senses as follows:

- **functional** category involved in syntactic consideration
- **logical** in agreement with numerals and quantifiers
- **formal** in other agreement, pronominal reference
- **illusory** category identifying morphs of an expression

The information on number and gender commonly returned by Arabic morphological analyzers is functional formal for verbs, which is most relevant, but only illusory for nominal expressions.

In the latter case, it may happen that the values for the given category coincide in all the senses. If there are no decisive clues, and there are none certain in Arabic, promoting the illusory values to the functional ones is in principle conflicting:

1. In most of the systems, categories are set only by presence of a morph, thus declaring it a morpheme even if the functional categories of the whole expression are different. Lexical morphemes are not qualified in lexicons as to the logical gender nor humanness, and logical number can be inferred only if the morphological stem of the logical singular is given along with the stem in question. Impiled erroneous interpretations include illusory feminine singular for e.g. *sādah* سادة *men*, *qādah* قادة *leaders*, *quḍāh* قادة *judges*, *dakāṭirah* دكاترة *doctors* (all functional masculine plural), ill. fem. pl. for *bāṣāt* باصات *buses* (log. masc. pl., form. fem. sing.), ill. masc. dual for *aynāni* عينان *two eyes*, *birāni* بيران *two wells* (both func. fem. dual), or even rarely ill. masc. pl. for *sinūna* سنون *years* (log. fem. pl., form. fem. sing.), etc.

---

4One can recall here the Arabic terms *marnāwiy* مرنوي by meaning and *laftiy* لفتي by expression. The logical and formal agreement, or *ad sensum* resp. grammatical, are essential notions (Fischer, 2001), yet, to our knowledge, not ever implemented.
2. Internal Arabic plural, i.e. the stem of logical plural that differs from the stem of logical singular, receives analyses setting no values for number, gender nor humanness, as no morphs surround the word stem and the stem’s morpheme does not have this information in lexicon. It would not work easily to set the desired functional values by default, as this operation could only be conditioned by the pattern of consonants and vowels, and that can easily mislead: 


3. The problem concerns every nominal expression individually, be it substantive, adjective or numeral. Every speaker of Arabic could give word lists like these to exemplify the point.\footnote{Versteegh (1997, chapter 6, page 83) offers a nice example of how the supposed principle of ‘one morph one meaning’, responsible for a kind of confusion similar to what we are dealing with, complicated some traditional morphological views.}

It is also common that the oblique case, the mere denotation for homonymous morphs of genitive and accusative in dual, plural and diptotic singular (all meant as illusory), is mistaken for an independent value of a grammatical category.

Considering definiteness, one issue is the logical definiteness of an expression within a sentence, the other is the formal use of morphs, and yet the third, the illusory presence or absence of the definite or the indefinite article.

Logical definiteness is binary, i.e. an expression is syntactically either definite, or indefinite. It figures in rules of agreement and rules of propagation of definiteness.

Formal definiteness introduces, in addition to indefinite and definite, the reduced and complex definiteness values describing word formation of nomen regens in construct states and logically definite improper annexations, respectively. Let us give examples:

\textbf{indefinite}  
\textit{حُلْواتٍ} nom. \textit{السَّوْى} $\rightarrow$ gen. \textit{صُنعًا} Sanaa, \textit{حُرَرْي} acc. \textit{تَسْعَى} $\rightarrow$ nom. \textit{السَّوْى} ninety, \textit{سَانَة} $\rightarrow$ acc. \textit{السَّوْى} years

\textbf{definite}  
\textit{الحُلْواتِ} nom. \textit{السَّوْى} $\rightarrow$ gen. \textit{السَّوْى} \textit{السَّانَة} $\rightarrow$ acc. \textit{السَّانَة} the-ninety, as-\textit{سَانَة} acc. \textit{السَّانَة} the-years

\textbf{reduced}  
\textit{حُلْواتُ} nom. \textit{السَّوْى} $\rightarrow$ gen. \textit{السَّوْى} \textit{السَّانَة} $\rightarrow$ acc. \textit{السَّوْى} years

\textbf{complex}  
\textit{حُلْواتِ} $\rightarrow$ gen. \textit{السَّوْى} \textit{السَّانَة} $\rightarrow$ acc. \textit{السَّانَة} years
3.1. FUNCTIONAL ARABIC MORPHOLOGY

<table>
<thead>
<tr>
<th>Token Forms</th>
<th>Gloss</th>
<th>Illus.</th>
<th>Form.</th>
<th>Logic.</th>
<th>Analytical</th>
</tr>
</thead>
<tbody>
<tr>
<td>sahhah-at</td>
<td>they-corrected</td>
<td>FS--</td>
<td>FS. . . . .</td>
<td>Pred</td>
<td></td>
</tr>
<tr>
<td>masadir-u</td>
<td>sources</td>
<td>--1</td>
<td>--1</td>
<td>MF.</td>
<td>T</td>
</tr>
<tr>
<td>muftali-at-un</td>
<td>well-informed</td>
<td>FS1</td>
<td>FS1</td>
<td></td>
<td>Atr</td>
</tr>
<tr>
<td>raft-at-u</td>
<td>high-of</td>
<td>FS1</td>
<td>FS1</td>
<td></td>
<td>Atr</td>
</tr>
<tr>
<td>al-mustawa</td>
<td>the-level</td>
<td>--?D</td>
<td>MS2</td>
<td>D</td>
<td>MS.</td>
</tr>
<tr>
<td>tashr-at-i</td>
<td>declarations-of</td>
<td>FP3</td>
<td>FP3</td>
<td></td>
<td>Obj</td>
</tr>
<tr>
<td>-him</td>
<td>theirs</td>
<td>MP--</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-him</td>
<td>theirs</td>
<td>MP--</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lafihi</td>
<td>these</td>
<td>?--</td>
<td>FS4</td>
<td></td>
<td>Atr</td>
</tr>
<tr>
<td>gayr-a</td>
<td>other-of</td>
<td>--4</td>
<td>--4</td>
<td>FS</td>
<td></td>
</tr>
<tr>
<td>ad-daqiq-at-i</td>
<td>the-accurate</td>
<td>FS2D</td>
<td>FS2</td>
<td>D</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 3.2 Functional (logical and formal) and illusory categories versus agreement, for the sentence well-informed high-level sources corrected these inaccurate declarations of theirs, with indicated analytical dependencies and functions (more in Sec. 8.2). Legend: . irrelevant, - unset, ? vague, M masculine, F feminine, S singular, P plural, 1 nominative, 2 genitive, 3 oblique case, 4 accusative, D definite, I indefinite, R reduced.

two multilingual

Proper names and abstract entities can be logically definite while formally and illusorily indefinite: fi Kānūna ‘t-tānī في كانون الثاني in January, the second month of ‘Kaanoon’. There are adjectival construct states that are logically indefinite, but formally not so: rafru ‘l-mustawa رفع المستوى a high-level, high-of the-level.

Figure 3.2 (Hajic et al., 2004b) exemplifies the principal difference between the functional and the illusory categories and shows the impossibility to restore agreement, and thus to have an excellent clue for parsing, if relying on the illusory analyses.

3.1.4 Approximating the Functional Model

The pure Functional Arabic Morphology (Smrz, in prep.) is being implemented, quite coincidentally, in the Functional Morphology by Forsberg and Ranta (2004), which is a methodology and a domain-specific programming language embedded in Haskell. The key ideas in our implementation draw on (Wadler, 1997, Huet, 1997, Hudak, 2000, Ljungløf, 2002, and references therein), and we use an analogy of Encode::Arabic (Smrz, 6)

6The dropped-الحالة-plus-الحالة cases of al-iddāfah gayr al-haqqiyyah الإضافة غير الحقيقية the improper annexation clearly belong here (cf. Smrz et al., 2006).

2003–2006) and the multi-purpose notation of ArabTeX (Lagally, 2004) as a meta-encoding of both the script and phonology.

Let us now describe how we have approximated the theoretical model of Functional Arabic Morphology in the first release of PADT, with the resources then given.

The underlying morphological engine for both the Penn Arabic Treebank (PATB, Maamouri et al., 2005) and the Prague Arabic Dependency Treebank (PADT, Hajic et al., 2004a) is the Buckwalter Arabic Morphological Analyzer (Buckwalter, 2002, 2004a). It produces analyzes per orthographical string as a whole (recall Section 3.1.1), in a format like

\[(\text{morph\_composition}) \ [\text{lemma\_id}]\]

\[
morph_1/tag_1 + morph_2/tag_2 + \ldots + morph_n/tag_n\]

where the \textit{morphs} group implicitly into the prefix, stem and suffix \textit{segments},

8 and the lemma identifies the semantically dominant morph, usually the stem, if there is one. Morphs are labeled with \textit{tags} giving them the feel that they must be morphemes, which is the source of the discrepancy between illusory and functional interpretations, as discussed earlier.

Let us illustrate these terms on a common example. Buckwalter’s morphology on the string \textit{wbjAnbhA و بناتها} meaning \textit{and next to her} would yield

\[(\text{wabijAnbihA}) \ [\text{jAnib}_1]\]

\[
wa/\text{CONJ} + bi/\text{PREP} + \text{prefix(es)}
\]

\[
jAnib/\text{NOUN} + \text{stem}
\]

\[
i/\text{CASE\_DEF\_GEN} + hA/\text{POSS\_PRON\_3FS} \text{suffix(es)}
\]

with the segments now indicated explicitly. The tokens, however, read \textit{wa wa-} and, \textit{bi bi-} at, \textit{jAnib+i} \textit{gānib-i} \textit{side-of} and \textit{hA -hā} \textit{her}. Note the morph \textit{i}, which is not a \textit{clitic} by itself, unlike the other three run-on tokens.

Although there is not enough \textit{functional information} provided in this kind of analyzes, we would like to \textit{approximate} it as closely as possible. When morphs are regrouped into tokens, their original tags form sequences (central column below) which map into a vector of values of grammatical categories. The tokens of our example will receive these converted, quasi-functional, positional\textsuperscript{9} tags (left column):

\[
\begin{array}{ll}
\text{C--------} & \text{wa} \ \text{CONJ} \ \text{wa--} \\
\text{P--------} & \text{bi} \ \text{PREP} \ \text{bi-} \\
\text{N------2R} & j\text{Anib+i} \ \text{NOUN+CASE\_DEF\_GEN} \ \text{gānib-i} \\
\text{S------3FS2-} & hA \ \text{POSS\_PRON\_3FS} \ \text{-hā}
\end{array}
\]

\textsuperscript{8}Some researchers use the term \textit{segment} for what is called \textit{morph} here, and allow strings to decompose to multiple prefixes and suffixes, including the \textit{clitics}.

\textsuperscript{9}Similar notations have been used in various projects, most notably the European Multext and Multext-East projects, for languages ranging from English to Czech to Hungarian.
The positional notation starts with the major and minor part-of-speech and proceeds through mood, voice, etc. up to person, gender, number, case, and definiteness. The values of the categories are unset, i.e., rendered with −, either if they are irrelevant for the particular part-of-speech (first position), or if there are no explicit data present in the original analysis, like no illusory gender and number in "jAnib+i". On the contrary, categories may be implied in parallel, cf. suffixed possessive pronouns being treated as regular pronouns, but in functional genitive (position nine). Some values can only be set based on other knowledge, which is the case of formal reduced definiteness (position ten).

In Figure 3.1, this whole transformation is illustrated on an excerpt from a running text. Strings in the first column there break into tokens, as indicated by dotted lines, and for tokens, quasi-functional tags are derived. Glosses as well as token forms with hyphens for morph boundaries are shown in the two rightmost columns of the figure.

The complete list of mappings from Buckwalter’s tags to the quasi-functional positional ones is available from the authors.
I love you when you dance, when you freestyle in trance, so pure, such an expression.

Alanis Morissette in So Pure (Supposed Former Infatuation Junkie)

Chapter 4

Impressive Haskell

4.1 Encode Arabic in Haskell
You’ll be relieved to know that I have no opinion on it. :-)  

Larry Wall on use Perl;

Chapter 5

Deterministic Parsing

5.1 Encode Mapper in Perl

We begin by building the trie structure (prefix tree) for all patterns that are to be matched, i.e. those for which there is some rule/action in the list of rules.

Listing 5.1 Compilation of rules into a Mapper

```perl
sub compile (@) {                              # returns Mapper .. modified Aho-Corasick
    my $cls = shift @_;                       
    my ($tree, @bell, @skip, @queue, %redef); 
    my ($q, $r, $s, $t, $i, $token, $trick);

    my ($null_list, $null_hash) = ([], {});   # no need for unique memory
    my ($no_code, $no_list) = (1, 1);         # optimization indicators

    $skip[0] = undef;                         # never used .. number of list elements equal
    $bell[0] = $null_list;                     # notice .. depth-wise inheritance of lists

    for ($i = 0; $i < @_; $i += 2) {
        next unless verify_rule $_[$i], $_[$i + 1];

        $q = 0;

        foreach $token (split //, $_[$i]) {
            $tree[$q]->{$token} = ++$r unless defined $tree[$q]->{$token};
            $q = $tree[$q]->{$token};
        }

        $tree[$q] = {} unless defined $tree[$q];

        whisper "Redefining the mapping for '$_[$i]'" if $redef{$q}++;
        $bell[$q] = [ $_[$i + 1] ];
    }
}
```
5.1. Encode Mapper in Perl

We further have to define transitions for any other input symbols, pointing from the initial node to itself. We might assume that these transitions are present by default, rather than construct them during compile time—such a thing would distract us and would obfuscate the following algorithms, and is probably not crucial anyway.

For all nodes \( q \) at the edges going out of the initial node, we define their skip function \( \text{skip}(q) = 0 \) to return to the initial node.

```perl
foreach $token (map { chr } 0x00..0xFF) {
    unless (defined $tree[0]->{$token}) {
        unless (defined $option{'others'}) {
            $tree[0]->{$token} = 0;
        }
    }
    $skip{$q} = 0;
    push @queue, $q;
}
```

Consider we have these patterns: “b”, “ab”, “ba”, “a”, “abacbd”. The right hand side of their rules are indifferent now, with only the condition that there is no lookahead part associated with them. Figure [next-left] shows what the procedure will make of them so far. The Figure [next-right] shows what the following code, the completion of the skip and the bell function, will result in.

```perl
while (@queue) { # generate $skip($q) backward function and complete $bell
    $q = shift @queue;
    foreach $token (keys %{$tree[$q]}) {
        $t = $tree[$q]->{$token};
    }
}
```

Revision of
August 7, 2006
push @queue, $t;

if (defined $bell[$t]) {
    $skip[$t] = 0;

    if (UNIVERSAL::isa($bell[$t]->[0], 'ARRAY')) {  # shortening property of the rules
        $s = $skip[$t];

        foreach $trick (split //, $bell[$t]->[0]->[1]) {
            until (defined $tree[$s]->{$trick}) {  # loops only if not in the
                push @{$bell[$t]}, @{$bell[$s]};
                $s = $skip[$s];
            }

        $s = $tree[$s]->{$trick};
    }

    $skip[$t] = $s;
    $bell[$t]->[0] = $bell[$t]->[0]->[0];
}
$s = $skip[$q];
$bell[$t] = [ @{$bell[$q]} ];  # unique reference quite important

until (defined $tree[$s]->{another}) {  # extremely tricky ...
push @{$bell[$t]}, @{$bell[$s]};
$s = $skip[$s];
}

$skip[$t] = $tree[$s]->{another};
}

$tree[$q] = $null_hash unless keys @{$tree[$q]};  # economize with memory

Since the rules did not involve any lookahead ‘cheating’, the reading in the code above was rather straightforward. If we now consider the optional lookahead, the algorithm will take the different routes in the code.

Suppose the rules to compile are now modified, so that the upon recognizing the sequence “ab” as the longest match, the automaton not only takes the appropriate action, but also ‘returns’ the lookahead “b” to the input. More precisely, it moves to exactly the state corresponding to seeing “b” on the input. Explain the cheating!

for ($q = 1; $q < @bell; $q++) {  # optimize the bell function for $q > 0
    if (grep { UNIVERSAL::isa($_, 'CODE') } @{$bell[$q]}) {
        $no_code = 0;
    }
    elsif (defined $option{'join'}) {
        $bell[$q] = join $option{'join'}, @{$bell[$q]};
        next;
    }
    if (@{$bell[$q]} == 1) {
        $bell[$q] = @{$bell[$q]}->[0];
    } else {
        $bell[$q] = $null_list if @{$bell[$q]} == 0;
        $no_list = 0;
    }
}

return bless {
    'current' => 0,
    'tree' => \@tree,
    'bell' => \@bell,
    'skip' => \@skip,
    'null' => { 'list' => $null_list, 'hash' => $null_hash },
The `process` function/method feeds in its parameters into the Mapper, which consumes them and returns the list of search results. It maintains its current state even after the input is over, and, keeps its search hypotheses in progress.

```perl
sub process ($@) { # returns the list of search results performed by Mapper my $obj = shift @_; my (@returns, $phrase, $token, $q);

$q = $obj->{‘current’};

if ($obj->{‘no_list’}) { foreach $phrase (@_) {
    foreach $token (split ‘/’, $phrase) {
        until (defined $obj->{‘tree’}[$q]->{$token}) {
```

Figure 5.2  Tricky Mapper for the rules including lookahead.

```
‘join’ => $option(‘join’),
‘no_code’ => $no_code,
‘no_list’ => $no_list,
}`, $cls;
```
5.1. ENCODE MAPPERS IN PERL

```
push @returns, $obj->{‘bell’}[$q];
$q = $obj->{‘skip’}[$q];
}
$q = $obj->{‘tree’}[$q]->{$token};
}
else {
    foreach $phrase (@_){
        foreach $token (split ‘//’, $phrase) {
            until (defined $obj->{‘tree’}[$q]->{$token}) {
                push @returns, ref $obj->{‘bell’}[$q] eq ‘ARRAY’ ?
                    @{$obj->{‘bell’}[$q]} : $obj->{‘bell’}[$q];
$q = $obj->{‘skip’}[$q];
        }
$q = $obj->{‘tree’}[$q]->{$token};
    }
    $obj->{‘current’} = $q;
}

return @returns;
```

The `recover` function/method returns the ‘in-progress’ search results and resets Mapper to the initial state.

```
sub recover ($;$$) {  # returns the ‘in-progress’ search result and resets Mapper
    my ($obj, $r, $q) = @_;  
    my (@returns);
$q = $obj->{‘current’} unless defined $q;

    if ($obj->{‘no_list’}) {
        until ($q == 0) {
            push @returns, $obj->{‘bell’}[$q];
$q = $obj->{‘skip’}[$q];
        }
    } else {
        until ($q == 0) {
            push @returns, ref $obj->{‘bell’}[$q] eq ‘ARRAY’ ?
                @{$obj->{‘bell’}[$q]} : $obj->{‘bell’}[$q];
$q = $obj->{‘skip’}[$q];
        }
$q = defined $r ? $r : 0;
    return @returns;
```
5.2 Encode Arabic in Perl
I love you when you dance, 
when you freestyle in trance, 
so pure, such an expression.

Alanis Morissette in So Pure (Supposed Former Infatuation Junkie)

Chapter 6

Encode Arabic

6.1 Encode Arabic in Haskell

We present an implementation of a Haskell library for processing the Arabic language in the Arab\TeX X transliteration (Lagally, 2004), a non-trivial and multi-purpose notation for encoding Arabic orthographies and phonetic transcriptions in parallel. Our approach relies on the Pure Functional Parsing library developed in (Ljunglöf, 2002), which we accommodate to our problem and partly extend. In the general view, we describe two alternative algorithms for longest-match deterministic parsing and rewriting, present the monadic-style grammars formalizing the relation of the script and the sound in Arabic, and promote modular design in systems for modeling or processing natural languages.

6.2 Introduction

Arab\TeX X (Lagally, 2004) is an extension to the \TeX /\LaTeX typesetting system that solves the problem of producing documents in the Arabic script, a writing system used with modifications by a variety of languages of the Orient.

One of the highlights of Arab\TeX X is its invention of an artful high-level notation for encoding the possibly multi-lingual text in a way that allows further interpretation by the computer program. In particular, the notation can be typeset in the original orthography of the language or in some kind of transcription, under one rendering convention or another. These options are controlled by setting the interpretation environment, and no change to the data is required.

While the Arab\TeX X notation is extremely useful for instance in lexicography and linguistics, the interpreter for it, the Arab\TeX X’s parser, cannot be directly applied to problems other than \TeX Xing.

With the motivation to overcome this limitation, we describe our implementation in Haskell of an independent and extensible parser converting the Arab\TeX X notation of Arabic into a representation isomorphic to Unicode. The result of our work is a succinct and reusable programming library called Encode Arabic.
6.3. Functional Parsing


The overall parsing process can be divided into layers of simpler parsing processes. Typically, there is one lexing/scanning phase involving deterministic finite-state techniques (Chakravarty, 1999), and one proper parsing phase resorting to context-free or even stronger computation to resolve the language supplied by the lexer.

Most of the libraries, however, implement parsers giving fixed data types for them, and implicitly restrict the parsing method to a single technique. Thus, lexing with Chakravarty’s CTK/Lexers and proper parsing with Leijen’s Parsec would imply ‘different’ programming.

A unifying and theoretically instructive account of parsing in a functional setting was presented by Peter Ljunglöf in his licentiate thesis (Ljunglöf, 2002). Pure Functional Parsing, esp. the parts discussing recursive-descent parsing using parser combinators, seemed the right resource for us to implement the grammars we need. This library in Haskell, called FunParsing in the sequel, abstracts away from the particular representation of the parser’s data type. It provides a programming interface based on type classes and methods, leaving the user the freedom to supply the parser types and the processing functions quite independently of the descriptions in the grammars.

The code in this section is due to (Ljunglöf, 2002). We extract only those snippets that are relevant to our own work.

```haskell
-- The classes of context-free and monadic combinator
-- parsers, from sections 2.4, 2.5 and 2.7 together
-- with the derived combinators from section 2.8

module FunParsing.Parsers.Parser where

infixr 4 <::>
```
A type \( m \) is a parser for a list of type \( s \) capable of computing multiple results of type \( a \), if it instantiates the `parse` method of the `Parser` class declaration.

```haskell
class Parser m s | m -> s where
  parse :: m a -> [s] -> [[(s), a]]
  parseFull :: m a -> [s] -> [a]
  parseFull p inp = [ a | ([], a) <- parse p inp ]
```

Parsers can be decomposed into or combined out of other, more elementary parsers. That is, we would like their types to be `Monoids`, to combine them in parallel, and also to belong to the class `Sequence`, in order to model their sequential composition.

```haskell
class Monoid m where
  zero :: m a
  (<+>) :: m a -> m a -> m a
  anyof :: [m a] -> m a
  anyof = foldr (<+>) zero
```

```haskell
class (Monad m, Functor m) => Sequence m where
  (<*>) :: m (a -> b) -> m a -> m b
  ( *> ) :: m a -> m b -> m b
  p <* q = p >>= \ f -> fmap f q
  p *> q = fmap (\ x y -> y) p <*> q
```

The presence of the parser’s type \( m \) in the `Sequence` class requires the type to be a `Functor`, for we would like to extract results from any parser by using the `fmap` method. It is also convenient to express the sequencing operators via the monadic bind operator, i.e. \( >>= \) of the `Monad` class. The \( <*> \) operator then reduces to the monadic \( >> \) operator, provided that the default implementations apply.

If there is equality \( == \) defined on the type of input symbols \( s \), we can make a parser of type \( m \) a member of the `Symbol` class, and implement the elementary parsers for recognizing or ignoring a single symbol.

```haskell
class Eq s => Symbol m s | m -> s where
  sym :: s -> m s
  sat :: (s -> Bool) -> m s
  skip :: m s
  sym s = sat (s ==)
  skip = sat (\ x -> True)
```

In finite-state parsers, the condition-satisfying `sat` parser can only be realized by combining the `sym` parsers for those symbols for which the condition holds. Then, we must require the input symbol type \( s \) to conform to the `InputSymbol` class.

```haskell
class Ord s => InputSymbol s where
  minSym, maxSym :: s
  symbols :: [s]
```
Within this module, we also define the derived combinators for listing the results of sequenced parsers, for applying a parser many times, and for recognizing a given sequence of symbols.

\[
\text{(<:>) :: } \text{Sequence } m \rightarrow m a \rightarrow m [a] \rightarrow m [a]
\]
\[
p <:> ps = \text{fmap } (:)_0 p <:> ps
\]

\[
\text{many :: } (\text{Monoid } m, \text{Sequence } m) \rightarrow m a \rightarrow m [a]
\]
\[
\text{many } p = ps
\]
\[
\text{where } ps = \text{return } [] \leftrightarrow p <:> ps
\]

\[
\text{syms :: } (\text{Sequence } m, \text{Symbol } m s) \rightarrow [s] \rightarrow m [s]
\]
\[
\text{syms } [] = \text{return } []
\]
\[
\text{syms } (s:ss) = \text{sym } s <:> \text{syms } ss
\]

### 6.4 Encode Mapper

In the Encode Mapper module, we would like to implement a lazy deterministic finite-state transducer. This kind of parser is in the first approximation represented as a trie, i.e. a tree structure built from the lexical specification in the grammar. In the trie, edges correspond to input symbols and nodes to states in which output results are possibly stored. A path from the root of the trie to a particular node encodes the sequence of symbols to be recognized in the input, if the result associated with that node is to be emitted.

Chakravarty (1999) gave an account on building tries with possible repetitions and cycles. The results can be actions or meta-actions—the latter being a device to escape to non-regular capabilities of the parsers, such as recognizing nested expressions or changing the parser dynamically during parsing. The parser does not allow ambiguous results, and parsing is controlled by the principle of the longest match.

Ljunglöf (2002) re-formulates such kind of parsing in terms of his library, and offers further explanation and discussion on the whole issue. While he develops several data representations of tries suited for ambiguous grammars and supporting efficient sharing of subtrees in memory, he leaves the question of longest match aside.

The Encode Mapper implements something in between the two. The nature of our mapper parser is the AmbExTrie parser described in detail in (Ljunglöf, 2002, sec. 4.3). We add to it the abilities to ‘cheat’ by rewriting the input with other symbols while still producing results of the general type \(a\), and to parse ambiguously using a breadth-first or a depth-first longest match algorithm.

```haskell
module Encode.Mapper where

import FunParsing.OrdMap
import FunParsing.Parsers.Parser

data Mapper s a = [Quit s a] :+: Map s (Mapper s a)
```
A node in the Mapper \( s \ a \) trie is the tuple \( \&: \) of a list of results of type \( \text{Quit} \ s \ a \) and a finite map from input symbols to subtries, mimicking the labeled directed edges. For better memory representation, subtries can be wrapped into the \( FMap \) constructor introduced in the original work. For finite maps, we use Ljunglöf’s \text{FunParsing.OrdMap} similar to \text{Data.Map} of the Haskell libraries.

The \( \text{Quit} \ s \ a \) data type is a tuple of a sequence of input symbols to be returned to the input upon a match, and of the result to be reported. Some elementary functions are given to access it.

\[
\text{quit} :: [s] \to \text{Quit} \ s \ a \to ([s], a)
\]
\[
\text{quit} r (s, a) = (s ++ r, a)
\]

\[
\text{returnQuit} :: [s] \to a \to \text{Quit} \ s \ a
\]
\[
\text{returnQuit} s a = (s, a)
\]

\[
\text{justQuit} :: \text{Quit} \ s \ a \to a
\]
\[
\text{justQuit} (s, a) = a
\]

\[
\text{skipQuit} :: \text{Quit} \ s \ a \to [s]
\]
\[
\text{skipQuit} (s, a) = s
\]

\[
\text{fmapQuit} :: (a \to b) \to \text{Quit} \ s \ a \to \text{Quit} \ s \ b
\]
\[
\text{fmapQuit} f (s, a) = (s, f a)
\]

Our notation for expressing grammars will use four new infix operators, the definition of which follows. The \( |+| \) appends an alternative rule. A completely matching input sequence with no cheating is combined with the result by \( |.| \). In case of cheating, i.e. input rewriting, the matching and the cheating sequences are joined with \( |-| \), and that is combined with the result via \( |:| \).

\[
infix 4 |-| \quad -- \text{rule } = \quad "a" \quad |.| 1
\]
\[
infix 3 |:|, |.| \quad -- \quad |+| "a" \quad |.| 2
\]
\[
infixl 2 |+| \quad -- \quad |+| "b" \quad |-| "aa" \quad |:| 3
\]

\[
|:| :: \text{InputSymbol} \ s \Rightarrow (a \to \text{Mapper} \ s \ a) \Rightarrow a
\]
\[
|:| x y = x y
\]

\[
|.-| :: \text{InputSymbol} \ s \Rightarrow [s] \Rightarrow [s] \Rightarrow a \Rightarrow \text{Mapper} \ s \ a
\]
\[
|.-| x y z = \text{sym} x \Rightarrow [\text{returnQuit} y z] \&: \text{emptyMap}
\]

\[
|.| :: \text{InputSymbol} \ s \Rightarrow [s] \Rightarrow a \Rightarrow \text{Mapper} \ s \ a
\]
\[
|.| x y = x |-| [] \&: y
\]

\[
|.|+| :: \text{InputSymbol} \ s \Rightarrow \text{Mapper} \ s \ a \Rightarrow \text{Mapper} \ s \ a
\]
\[
|.|+| = (<+>)
\]
Plus two more convenience functions to use in the grammars.

\[
\text{anySymbol} :: (\text{Monoid } m, \text{ Symbol } m \ a) \Rightarrow [a] \rightarrow m \ a
\]
\[
\text{anySymbol} = \text{anyof \ map \ sym}
\]

\[
\text{some} :: (\text{Monoid } m, \text{ Sequence } m) \Rightarrow m \ a \rightarrow m \ [a]
\]
\[
\text{some } p = p <\!:\!> \text{many } p
\]

The construction of our Mapper $s$ a trie is modified relative to AmbExTrie in order to accommodate the Quit $s$ a type internally, yet to only produce the pure type $a$ on all interfaces.

The combinators in a grammar for Mapper are actually constructors that take care of building the trie gradually with rules. The trick to improve subtrie sharing rests in delaying function application on the results, and instead storing the modification functions inside the FMap value. Note therefore the definitions of fmap and unfold.

\[\text{instance Ord } s \Rightarrow \text{Functor } (\text{Mapper } s) \text{ where}\]
\[\text{fmap } = \text{FMap}\]
\[\text{unfold} :: \text{Ord } s \Rightarrow (a \rightarrow b) \rightarrow \text{Mapper } s \ a \rightarrow \text{Mapper } s \ b\]
\[\text{unfold } f \ (\text{as } :\&: \text{pmap}) = \text{map } (\text{fmapQuit } f) \ as \ :\&:\]
\[\quad \text{mapMap } (\text{FMap } f) \ pmap\]
\[\text{unfold } f \ (\text{FMap } g \ p) = \text{FMap } (f \ . \ g) \ p\]

An empty trie, the zero parser, has no results in the root and no edges to point to any subtries. The $<\!:\!>$ combinator merges the edges and the results of two parsers into a single combined trie. Recognizing a symbol with $\text{sym } s$ creates a trie with one edge $s$, at the end of which there is a single node with result $s$.

\[\text{instance Ord } s \Rightarrow \text{Monoid } (\text{Mapper } s) \text{ where}\]
\[\text{zero} = [\ ] :\&: \text{emptyMap}\]
\[\text{FMap } f \ p \ <\!:\!> \ q = \text{unfold } f \ p <\!:\!> q\]
\[p \ <\!:\!> \text{FMap } f \ q = p <\!:\!> \text{unfold } f \ q\]
\[\text{as } :\&: \text{pmap} <\!:\!> (\text{bs } :\&: \text{qmap}) = (\text{as } ++ \text{bs}) :\&:\]
\[\quad \text{mergeWith } (\text{<\!:\!>}) \ pmap \ qmap\]

\[\text{instance InputSymbol } s \Rightarrow \text{Symbol } (\text{Mapper } s) \ s \text{ where}\]
\[\text{sym } s = [\ ] :\&: (s \mid-> \text{return } s)\]
\[\text{sat } p = \text{anyof } (\text{map } \text{sym } (\text{filter } p \text{ symbols}))\]

Sequencing of tries means applying the continuation trie $k$ on any of the results as associated with the current node $:\&:\$ in the current trie, or continuing parsing with the current trie and binding the continuation trie recursively on the current subtries.

\[\text{instance Ord } s \Rightarrow \text{Sequence } (\text{Mapper } s)\]

\[\text{instance Ord } s \Rightarrow \text{Monad } (\text{Mapper } s) \text{ where}\]
\[\text{return } a = [\text{returnQuit } [\ ] a] :\&: \text{emptyMap}\]
\[\text{FMap } f \ p \ \gg\gg \ k = \text{unfold } f \ p \ \gg\gg \ k\]
\[\text{as } :\&: \text{pmap} \ \gg\gg \ k = \text{foldr } (\text{<\!:\!>})\]
\[\quad ([]) :\&: \text{mapMap } (\gg\gg \ k) \ pmap\]
\[\quad (\text{map } (k \ . \text{justQuit}) \ as)\]
Parsing with tries then amounts to traversing the trie along the edges according to the symbols on input. Results in the trie are still subject to the function accumulated from FMap$s$.

\[
\text{instance } \text{Ord } s \Rightarrow \text{Parser } (\text{Mapper } s) s \text{ where}
\]
\[
\text{parse } p \text{ inp } = \text{parse'} p \text{ inp } id \\
\text{parseFull } p \text{ inp } = \text{parseFull'} p \text{ inp } id
\]

\[
\text{parse'} :: \text{Ord } s \Rightarrow \text{Mapper } s a \rightarrow [s] \rightarrow (a \rightarrow b) \\
\rightarrow ([s], b)]
\]
\[
\text{parse'} ([] \&\& pmap) [] k = [k]
\]
\[
\text{parse'} ([] \&\& pmap) (s:inp) k = \text{case } pmap ? s \text{ of}
\]
\[
\text{Just } p \rightarrow \text{parse'} p \text{ inp } k \\
\text{Nothing } \rightarrow []
\]
\[
\text{parse'} (xs :\&\& pmap) \text{ inp } k =
\]
\[
\text{foldr } (((::) . \text{quit } \text{ inp } . \text{fmapQuit } k) \\
(\text{parse'} ([] \&\& pmap) \text{ inp } k) \text{ xs)
\]

\[
\text{parseFull'} :: \text{Ord } s \Rightarrow \text{Mapper } s a \rightarrow [s] \rightarrow (a \rightarrow b) \\
\rightarrow [b]
\]
\[
\text{parseFull'} (\text{FMap } f \text{ p}) \text{ inp } k = \text{parseFull'} p \text{ inp } (k \cdot f)
\]
\[
\text{parseFull'} p@\{xs \&\& _\} [] k = \text{concat } (\text{map } \text{quitQuit } xs)
\]
\[
\text{where } \text{quitQuit } x = \text{case } \text{quit } [()] (\text{fmapQuit } k x) \text{ of}
\]
\[
([], y) \rightarrow [y] \\
(is, y) \rightarrow y : \text{parseFull'} p \text{ is } k
\]
\[
\text{parseFull'} _\{\_ \&\& pmap\} (s:inp) k = \text{case } pmap ? s \text{ of}
\]
\[
\text{Just } p \rightarrow \text{parseFull'} p \text{ inp } k \\
\text{Nothing } \rightarrow []
\]

We once again refer the reader to (Ljunglöf, 2002) for the proper discussion of this technique. In the rest of this section, we present our own contribution—the two longest-match parsing algorithms.

### 6.4.1 Longest Match Insight

Consider the following rules defining our example trie in Figure 6.1a:

\[
\text{trie } :: \text{Mapper } \text{Char } [\text{Char}]
\]
\[
\text{trie } = (\text{many } \{\text{sym } "c"\}) \&\& \text{return } "x"
\]
\[
\mid + | "b" \mid . | "b" \mid \text{ -- equal to syms } "b"
\]
\[
\mid + | "a" \mid . | "a" \mid 
\]
\[
\mid + | "ab" \mid - | "b" \mid : | "ab" \mid \text{ -- cheating with } "b"
\]
\[
\mid + | "ba" \mid . | "ba" 
\]
\[
\mid + | "abacbd" \mid . | "abacbd"
\]

We can view this trie as a dictionary specifying a language. Its words are composed of the labels of those edges that create a path from the root to some node with a non-empty list of results. Given an arbitrary input string, we would like to use this trie for finding and translating the longest non-overlapping substrings that belong to the dictionary. Yet, cheating and ambiguities will be allowed.
The inspiration for us is the Aho–Corasick algorithm for text search (Aho and Corasick, 1975). The important insight is the idea of a failure function, depicted with dashed lines in Figure 6.1b.

If parsing cannot proceed along the labeled, solid edges in the trie, there is no chance for any longer match. Then, we can report the longest match results we had collected (the node labels in this subfigure). But we also have to keep in mind that the input symbols accepted after the latest match can form a prefix of a new successful match. The failure function serves to move us to this prefix. From that point and with the same input symbol, we iterate this process until we succeed or reach the root in the trie. Then we parse the next input symbol, or conclude by following the failure functions and reporting the collected results until we reach the root.

In our implementation in Haskell, we are not going to construct the failure function in the trie. It would require to traverse the whole data structure before parsing a single symbol. Thus, we would lose the great advantage of lazy construction of the trie. The Mapper would also become restricted to finite tries only, which we cannot easily guarantee given the power of the combinator grammar.

Therefore, the parsing process itself will have to simulate what the failure function provides. We can either develop parallel hypotheses about where the actual match
occurs (breadth-first search), or try the hypotheses in a backtracking manner using an accumulator for the non-matched input (depth-first search).

### 6.4.2 Breadth-First Algorithm

The `ParseWide` data type will maintain all information reflecting the state of computation. The complete trie will be passed to various functions as an argument (see `m` further), but what we store in each `PW` is the current subtrie to parse with. We will store the results not as a list, but as a `show` function to avoid inefficient list operations.

```haskell
data ParseWide s a = forall b . PW
  Int -- length of the current prefix
  ([a] -> [a]) -- accumulator for sure results
  (b -> a) -- transformer for new results
  (Mapper s b) -- current node’s subtrie parser
  ([ParseWide s a]) -- dependent parallel hypotheses
```

```haskell
initPW :: Ord s => Mapper s a -> ([a] -> [a]) -> ParseWide s a
initPW m h = PW 0 h id m []
```

The initial `PW` takes as arguments the complete trie and the history of results, which will be applied as a function on the new results that we will acquire during parsing any future input.

Symbols will be processed one by one, each causing a change in the state of parallel computations. Thus, `parseWide` will take care of iterating over the input and hypotheses, while `parsePW` will develop a single `PW` into a list of advanced alternative `PW` hypotheses.

```haskell
parseWide :: Ord s => Mapper s a -> [ParseWide s a] -> s -> [s] -> [ParseWide s a]
parseWide m = foldl (\w y -> concat [ parsePW m p y | p <- w ])
```

```haskell
parsePW :: Ord s => Mapper s a -> ParseWide s a -> s -> [ParseWide s a]
parsePW m (PW l h f c s) y = let n = l + 1 in case c of
  FMap t q -> parsePW m (PW l h f q c s) y
    where FMap qf qc = stripFMap (f . t) q
  r & k -> case k ? y of
    Just q -> let FMap qf qc = stripFMap f q in
      case qc of
        [] -> case r of
        [] -> case s of
```
Let us explain this part of the parsePW function now, which describes the following situation. We assume to be at a node \( r : &: k \), from which there is an edge with label \( y \) leading through some \( \text{FMap}s \) to the next node \( qc \). If there are some results associated with \( qc \) (the most recent line of code), they will for sure become the match. We therefore delete all previous dependent hypotheses, and are ready to continue parsing with \( qc \).

If there are no results found with \( qc \), we investigate the results \( r \) of the current node. If there are none, we simply let develop all dependent hypotheses \( s \) upon the input symbol \( y \), and move the current node to \( qc \). Otherwise, we take the ambiguous results one by one and create the list of dependent hypotheses. If with \( y \) we are just entering the complete trie, i.e. \( l == 0 \), we start a plain hypothesis. It disregards any symbol cheating and means that \( y \) might not be part of any successful match. Then, \( f \ (\text{justQuit} \ x) \) would be reported, i.e. composed in future with the history \( h \). If \( l > 0 \), we develop fully fledged hypotheses re-parsing the input \( \text{skipQuit} \ x ++ [y] \). The reason for this distinction is non-termination if we allowed repeated empty-word matching.

If there is no edge for \( y \) leading out of \( r : &: k \), we essentially process the results of \( r \) and do what the fictitious failure function would do. We now observe that the dependent hypotheses extended with \( \text{lastPW} \ s \ (\text{initPW} \ m \ \text{id}) \) contain exactly the subtries to which the failure function is pointing transitively.

```haskell
Nothing -> case l of
  0 -> case r of
     [] -> [ PW l h f c s ]
     xs -> [ initPW m (h . (f (justQuit x))) | x <- xs ]
     _ -> case r of
```

### Nothing -> case l of

0 -> case r of

[] -> [ PW l h f c s ]

xs -> [ initPW m (h . (f (justQuit x))) | x <- xs ]
6.4. Encode Mapper

\[
[] \rightarrow [\text{PW rn (h . rh) rf rc rs}] \\
\text{PW rn rh rf rc rs} \leftarrow \text{parseWide m} \\
\text{(lastPW s (initPW m id))} \\
[y] \\
\text{xs} \rightarrow \text{concat [ parseWide m] } \\
\text{[initPW m (h . (f (justQuit x) :))]} \\
\text{(skipQuit x ++ [y])} \\
\text{x \leftarrow xs]}
\]

The `parsePW` function is rather complete, only two more definitions.

\[
\text{stripFMap :: Ord s} \Rightarrow (a \rightarrow c) \rightarrow \text{Mapper s a} \\
\rightarrow \text{Mapper s c} \\
\text{stripFMap k (FMap f p)} = \text{stripFMap (k . f) p} \\
\text{stripFMap k x} = \text{FMap k x}
\]

\[
\text{lastPW :: Ord s} \Rightarrow [\text{ParseWide s a}] \rightarrow \text{ParseWide s a} \\
\rightarrow [\text{ParseWide s a}] \\
\text{lastPW w p} = \text{case w of} [\text{]} \rightarrow [p]; _ \rightarrow w
\]

The decisions about longest match are delayed to respond properly to any next input symbol. No sooner than at the very end of the input, can we interpret the `ParseWide` structures and report the results. We define `unParseWide` and `unParsePW` for that purpose. Some thought goes with the symbol cheating, as we must finish parsing on that input, too. We continue the discussion below.

\[
\text{unParseWide :: Ord s} \Rightarrow \text{Mapper s a} \rightarrow [\text{ParseWide s a}] \\
\rightarrow [[[a]]] \\
\text{unParseWide m} = \text{concat . map (unParsePW m)}
\]

\[
\text{unParsePW :: Ord s} \Rightarrow \text{Mapper s a} \rightarrow \text{ParseWide s a} \\
\rightarrow [[[a]]] \\
\text{unParsePW m (PW l h f c s)} = \text{case c of} \\
\text{FMap t q} \rightarrow \text{unParsePW m (PW l h qf qc s)} \\
\text{where FMap qf qc} = \text{stripFMap (f . t) q} \\
\text{r :@: k} \rightarrow \text{case r of} \\
\text{[]} \rightarrow \text{case s of} \\
\text{[]} \rightarrow [[[h []]]] \\
\text{zs} \rightarrow [\text{h u : v l (u : v) \leftarrow unParseWide m zs}]
\]

\[
\text{xs} \rightarrow \text{case l of} \\
0 \rightarrow [[[h []]]] \\
_{-} \rightarrow \text{concat [ case skipQuit x of} \\
\text{[]} \rightarrow [[[h [f (justQuit x)]]]]
\]
is -> [ h [f {justQuit x}] : u | u <- unParseWide m (parseWide m
(initPW m id) is) ]
| x <- xs ]

Let us see what results we get at this moment for our example trie:

ex :: [Char] -> [[[Char]]]
ex = unParseWide trie . parseWide trie [initPW trie id]

ex "ab"   -> [[["ab"],["b"]]]
ex "aba"   -> [[["ab"],["ba"]]]
ex "abacba" -> [[["ab"","ba","x","ba"]]]
ex "abacbc" -> [[["ab"","ba","x","b","x"]]]
ex "abacccc" -> [[["x","ab","ba","x","b","x","x"]]]

In case of trie with ‘problematic’ rules and ex’ defined likewise:

trie’ = trie |+| "ab" |.|.unParseWide m
|+| "" |.-| "a" |::| "y" |-- undefined
|+| "c" |.-| "abac" |::| "" |-- expansion

ex’ "abab" -> [[["ab","ba","b"]]}, -- cheating difference
[[["ab","ab"];["b"]]
[[["ab","ab"]]

ex’ "cc"   -> [[["x"]]]

ex’ "c"    -> [[["x"]],
[[""],["ab","ba","x"]],
[[""],["ab","ba","x"],["ab","ba","x"]], ...
-- iterating ["ab","ba","x"] infinitely

ex’ "cbd"  -> [[["x","b","x"]],
[["x","b","y"],
[[""],["abacbd"]]]
-- double match for ‘d’
[[""],["abacbd"]]] -- finite ‘c’ rewriting

The expressive power of the rules and the robustness of the algorithm is pleasing. Obvi-
ously, the outermost level of listing is there for ambiguous results. The deeper level
lists separately the very last parses required in unParsePW in case of cheating. The in-
nermost level then enumerates the longest-match results.

We can harness this flexibility into parseLongestWide to return the first solution only
and concatenate all its ‘cheating’ phrases. Or we can provide parseLongestWideWith
with a more standard interface, similar to that of parseFull, yet customizable by the
user.

parseLongestWide :: Ord s => Mapper s a -> [s] -> [a]
parseLongestWide =
                parseLongestWideWith (head . map concat)

parseLongestWideWith :: Ord s => (([[a]]) -> [b])
                        -> Mapper s a -> [s] -> [b]
parseLongestWideWith f m =
    f . unParseWide m . parseWide m [initPW m id]
6.4. ENCODE MAPPER

6.4.3 Depth-First Algorithm

The depth-first algorithm has to collect the non-matched input in an extra record. The analogy with the breadth-first algorithm is strong, so we will be terse and will not show any type signatures.

```hs
data ParseDeep s a = forall b . PD
  Int
  ([a] -> [a])
  (b -> a)
  (Mapper s b)
  [s]
  [ParseDeep s a]

initPD m h = PD 0 h id m [] []
parseDeep m = foldl \( w \ y \rightarrow \text{concat} \)
  [ parsePD m p y | p <- w ]
parsePD m (PD l h f c i s) y = let n = l + 1 in case c of
  FMap t q -> parsePD m (PD l h qf qc i s) y
      where FMap qf qc = stripFMap (f . t) q

r :&: k -> case k ? y of
  Just q -> let FMap qf qc = stripFMap f q in
      case qc of
        ([]) :&: _ -> case r of
          [] -> case s of
            [] -> [ PD n h qf qc [] [] ]
            zs -> [ PD n h qf qc (y : i) zs ]
        xs -> case l of
          0 -> [ PD n h qf qc [] ]
            [ initPD m (f (justQuit x) :) ]
            x <- xs ]
          _ -> [ PD n h qf qc [y] (concat
                  [ parseDeep m
                    [initPD m (f (justQuit x) :)]
                    (skipQuit x) ]
                    x <- xs ] ) ]
        _ -> [ PD n h qf qc [] [] ]

Nothing -> case l of
```
The depth-first algorithm would, by intuition, probably need less work to find the solution. In Hugs, however, it is the breadth-first algorithm which does slightly better in terms of the number of reductions and cells. Perhaps, lazy evaluation of the breadth-
first approach is responsible for this. We have not measured the performance in time, though. In the module, we use the following:

```plaintext
cparselonest = pparselonestWide
```

### 6.5 Encode Extend

In this section, we will describe the Encode Extend module implementing a general recursive-descent parser derived in the standard approach as a state transformer monad (Wadler, 1995). The ‘extension’ is that we decompose the state into the input being processed and the environment supplying any other needed parameters.

The `Extend e parser is based on the Standard parser discussed in (Ljunglöf, 2002, sec. 3.2). Its state is a combination `InE s e of a list of input symbols s and a stack of environment settings e s.

```plaintext
module Encode.Extend where
import FunParsing.OrdMap
import FunParsing.Parsers.Parser
import Control.Monad

newtype Extend e s a = Ext (InE s e -> [(InE s e, a)])

type InE i e = ([i], [e i])
```

No modification of the instance declarations found in the original work is done, except for a minor change in the sat method.

```plaintext
instance Monoid (Extend e s) where
  zero = Ext (\inp -> [])
  Ext p <+> Ext q = Ext (\inp -> p inp ++ q inp)
```

```plaintext
instance Monad (Extend e s) where
  return a = Ext (\inp -> [(inp, a)])
  Ext p >>= k = Ext (\inp -> concat [ q inp' |
                                       (inp', a) <- p inp,
                                       let Ext q = k a ])
```

```plaintext
instance Sequence (Extend e s)
instance Functor (Extend e s) where
fmap f p = do a <- p; return (f a)
```

```plaintext
instance Eq s => Symbol (Extend e s) s where
sat p = Ext sat'
  where sat' ((s : inp), e) | p s = [(inp, e), s)]
    sat' _ = []
```
The `Extend `es parser for symbols `s` is polymorphic in the type of the environment `e`. The methods of the `Parser` class cannot incorporate the environment information directly—they can, however, delegate the task to other functions supplied with some initial environment. Therefore, we need environment types to belong to the `ExtEnv` class to guarantee the instantiation of the `initEnv` method.

```haskell
class ExtEnv e where initEnv :: e i

instance ExtEnv e where initEnv :: e i

parse :: ExtEnv e => e s -> Extend e s a -> [s] -> [[(s), a]]
parse' e (Ext p) i = [ (x, y) | ((x, _), y) <-- p i, e ]
parseFull :: ExtEnv e => e s -> Extend e s a -> [[s]]
parseFull' :: ExtEnv e => e s -> Extend e s a -> [[s]]
```

It is up to the user’s definition of the environment type, what kind of information it will store. Thus, `Extend `es a` can become a flexible generalization of various functional parser implementations that keep track of the current position in the parsed string, for example, via a hard-wired data type or in some other obligatory manner.

How then can we creatively work with the environment? The `Extend `es is a monad capable of computing any result type `a`. It is possible to include environment inspection or setting into the grammars, if we define parsers that access the environment part of the state, just like the usual ones only access the input symbol part.

```haskell
inspectIList :: Extend e s [s]
inspectIList = Ext (\(i, e) -> [((i, e), i)])

returnIList :: [s] -> Extend e s [s]
returnIList i = Ext (\(_, e) -> [((i, e), i)])

inspectEList :: Extend e s [e s]
inspectEList = Ext (\(i, e) -> [((i, e), e)])

returnEList :: [e s] -> Extend e s [e s]
returnEList e = Ext (\(i, _) -> [((i, e), e)])
```

Environments are stacked in a list modeling the hierarchy of conceivable nesting. Still, only the head of the list is of interest in most situations. Inspect it with `inspectEnv`. Setting up a single current environment’s value `v` with `resetEnv` needs also the function `f` defined for reconstructing the concrete environment type `e`.

```haskell
inspectEnv :: Extend e s (e s)
inspectEnv = Ext (\(i, e) -> [((i, e), head e)])

resetEnv :: (a -> e s -> e s) -> a -> Extend e s (e s)
```
resetEnv f v = Ext (\ (i, e : q) ->
    [((i, f v e : q), f v e)])

The biased choice combinator <|> tries the second parser only if the first parser fails (Wadler, 1995). Thus, again will return the longest possible parse only, in terms of iterations—cf. <+> and many.

\[ \text{infixr} \ 2 \ (<|>) \]

\[(<|>) :: \text{Extend e s a} \rightarrow \text{Extend e s a} \rightarrow \text{Extend e s a} \]

\[(<|>) \ p \ q = \text{Ext (\ cs -> let Ext pp = p}
\hspace{1cm} r = pp cs
\hspace{1cm} Ext qq = q
\hspace{1cm} \text{in case r of [] } \rightarrow qq \ cs
\hspace{1cm} (s : _) \rightarrow [s]
\hspace{1cm} _ \rightarrow r) \]

again :: \text{Extend e s a} \rightarrow \text{Extend e s [a]}
again p = ps where ps = p <+> ps <|> return []

In the Encode Extend module, we also design several convenience functions whose usage will be explained in the next sections.

\[ \text{lookupList} :: (\text{OrdMap m, Ord s}) \Rightarrow s \rightarrow [m s a] \rightarrow [a] \]

\[ \text{lookupList} \ x \ l = \text{concat} [\text{maybe} [] (: []) (i ? x) |
\hspace{1cm} i <- l ] \]

\[ \text{oneof} :: (\text{Ord s}, \text{Symbol m s}) \Rightarrow [\text{Map s a}] \rightarrow m s \]

\[ \text{oneof} \ l = \text{sat (\ s -> any (\ i -> maybe False}
\hspace{1cm} (\text{const True}) (i ? s)) l) \]

\[ \text{lower} :: (\text{Ord s}) \Rightarrow [s] \rightarrow [s] \rightarrow \text{Extend e s [s]} \]

\[ \text{lower s c = Ext (\ inp -> [ ((c ++ i, e), r) |
\hspace{1cm} ((i, e), r) <- f inp ]} \]

where Ext f = \text{syms} s

\[ \text{upper} :: (\text{OrdMap m, Ord s}) \Rightarrow [s] \rightarrow [m s [c]] 
\hspace{1cm} -> \text{Extend e d ([c] -> [c])} \]

\[ \text{upper s l = foldM (\ f -> fmap (..) f) . anyof}.
\hspace{1cm} \text{map (return . (++)})
\hspace{1cm} \text{id [ lookupList x l | x <- s ]} \]

\[ \text{upperWith} :: (s -> m -> e d -> [[c]]) -> [s] -> m 
\hspace{1cm} -> \text{Extend e d ([c] -> [c])} \]

\[ \text{upperWith f s l =}
\hspace{1cm} \text{do e <- inspectEnv}
\hspace{1cm} \text{foldM (\ f -> fmap (..) f) . anyof}.
\hspace{1cm} \text{map (return . (++)})
\hspace{1cm} \text{id [ f x l e | x <- s ]} \]
6.6 Encode Arabic

Before applying Encode Mapper and Encode Extend to the notation of ArabTeX, let us reformulate the idea of converting textual data from one encoding scheme into another in the way inspired by the Encode module in Perl (Kogai, 2002–2006).

We introduce the internal representation `UPoint` as the intermediate data type for these conversions. The distinction between this representation and characters `Char` is intentional, as ‘decoded’ and ‘encoded’ data are different entities. Since `UPoint` is an instance of the `Enum` class, the type’s constructor and selector functions are available as `toEnum` and `fromEnum`, respectively.

```haskell
module Encode where

newtype UPoint = UPoint Int deriving (Eq, Ord, Show)

instance Enum UPoint where
    fromEnum (UPoint x) = x
    toEnum = UPoint

class Encoding e where
    encode :: e -> [UPoint] -> [Char]
    decode :: e -> [Char] -> [UPoint]

    encode _ = map (toEnum . fromEnum)
    decode _ = map (toEnum . fromEnum)

```

Encoding schemes are modeled as data types `e` of the `Encoding` class, which defines the two essential methods. Developing a new encoding means to write a new module with a structure similar to `Encode.Arabic.Buckwalter` or `Encode.Unicode`, for instance.

```haskell
module Encode.Arabic.Buckwalter (Buckwalter(..)) where

import Encode
import FunParsing.OrdMap

data Buckwalter = Buckwalter | Tim deriving (Enum, Show)

instance Encoding Buckwalter where
    encode _ = recode (recoder decoded encoded)
    decode _ = recode (recoder encoded decoded)

recode :: (Eq a, Enum a, Enum b, Ord a) => Map a b -> [a] -> [b]
recode xry xs = [ lookupWith ((toEnum . fromEnum) x) xry x | x <- xs ]

recoder :: Ord a => [a] -> [b] -> Map a b
recoder xs ys = makeMapWith const (zip xs ys)

decoded :: [UPoint]
```
The Buckwalter encoding is a lossless romanization of the standard Arabic script, and is a one-to-one mapping between the Unicode code points for Arabic and lower ASCII. In Figure 6.2, we highlight this correspondence by coloring vocalization marks, written only optionally in most documents, in red. Note that the glyphs for the letters in the script are context-dependent, yet they are just variants of a single grapheme (cf. Beesley, 1997, 1998, for terminology).

module Encode.Unicode (Unicode(..)) where

import Encode

data Unicode = Unicode | UCS deriving (Enum, Show)

instance Encoding Unicode

6.6.1 ArabTEX Encoding Concept

The ArabTEX typesetting system (Lagally, 2004) defines its own Arabic script meta-encoding that covers both contemporary and historical orthography in an excellent way. Moreover, the notation is human-readable as well as very natural to learn to write with. The notation itself is quite close to the phonetic transcription, yet extra features are introduced to make the conversion to script/transcription unambiguous. More comments on Figure 6.2 will follow.

Unlike other transliteration concepts based on the one-to-one mapping of graphemes, ArabTEX interprets the input characters in context to get their right meaning. Finding
6.6. Encode Arabic

This module relates the Arab\TeX notation and the Arabic orthography. It provides definitions of the ‘lexicon’ of type \texttt{LowerUp}, which lists the distinct items in the notation and associates them with the information for their translation. Lexical items are identified by their \texttt{Char} representation and their transcription.

<table>
<thead>
<tr>
<th>Script</th>
<th>Buckwalter</th>
<th>Arab\TeX \texttt{[Char]}</th>
<th>Transcription</th>
</tr>
</thead>
<tbody>
<tr>
<td>ببلبل</td>
<td>&quot;bulbulun&quot;</td>
<td>bulbulun</td>
<td>(a)</td>
</tr>
<tr>
<td>بالبلبل</td>
<td>&quot;balAbilu&quot;</td>
<td>balâbilu</td>
<td>(b)</td>
</tr>
<tr>
<td>قرأ،</td>
<td>&quot;qurrA'un&quot;</td>
<td>qirâtan</td>
<td>(c)</td>
</tr>
<tr>
<td>قرأ‘اطن</td>
<td>&quot;qirA'atun&quot;</td>
<td>qirâtan</td>
<td>(d)</td>
</tr>
<tr>
<td>وٍ</td>
<td>&quot;say'IN&quot;</td>
<td>sayin</td>
<td>(e)</td>
</tr>
<tr>
<td>ن</td>
<td>&quot;say'{A}ni&quot;</td>
<td>say'{A}ni</td>
<td>(f)</td>
</tr>
<tr>
<td>لا</td>
<td>&quot;sayi'unya&quot;</td>
<td>sayyi'{u}na</td>
<td>(g)</td>
</tr>
<tr>
<td>لا</td>
<td>&quot;sayi'unya&quot;</td>
<td>sayyi'{u}na</td>
<td>(h)</td>
</tr>
<tr>
<td>لا</td>
<td>&quot;sayi'unya-a&quot;</td>
<td>sayyi'{u}na-a</td>
<td>(i)</td>
</tr>
<tr>
<td>زيارة الزوار</td>
<td>&quot;\cap ziyArat-u&quot;</td>
<td>Ziyārat-u</td>
<td>(k)</td>
</tr>
<tr>
<td>할وانا</td>
<td>&quot;az-zUwAr-i&quot;</td>
<td>'z-zūwar-i</td>
<td>(l)</td>
</tr>
<tr>
<td>ماشه</td>
<td>&quot;m_I'ah\textrm{u}N&quot;</td>
<td>mi'ah</td>
<td>(m)</td>
</tr>
<tr>
<td>بياى</td>
<td>&quot;bi-al-mi'ah&quot;</td>
<td>bi-'l-mi'ah</td>
<td>(n)</td>
</tr>
</tbody>
</table>

Figure 6.2 Interpreting the \texttt{[Char]} input in the Arab\TeX notation.

the glyphs of letters (initial, medial, final, isolated) and their ligatures is not the issue of encoding, but of visualizing only. Nonetheless, definite article assimilation, inference of hamza carriers and silent \textit{alif}s, treatment of auxiliary vowels, optional quoting of diacritics or capitalization, resolution of notational variants, and mode-dependent processing are the challenges for our parsing exercise now.

Arab\TeX’s implementation is documented in (Lagally, 1992), but the parsing algorithm for the notation has not been published. The \TeX code of it is organized into deterministic-parsing macros, yet the complexity of the whole system makes consistent modifications or extensions by other users very difficult, if not impossible.

We are going to describe our own implementation of the interpreter, i.e. we will show how to decode the notation. To encode the Arabic script or its phonetic transcription into the Arab\TeX notation requires some heuristics, if we want to achieve linguistically appropriate results. We leave these for future work.

6.6.2 Encode Arabic Arab\TeX

This module relates the Arab\TeX notation and the Arabic orthography. It provides definitions of the ‘lexicon’ of type \texttt{LowerUp}, which lists the distinct items in the notation and associates them with the information for their translation. Lexical items are identi-
fied in the lexing phase by an instance of Encode Mapper of type Mapping. The proper parsing phase uses Encode Extend parsers of type Parsing.

```haskell
module Encode.Arabic.ArabTeX (ArabTeX(..)) where

import Encode
import Encode.Mapper
import Encode.Extend
import FunParsing.OrdMap

data ArabTeX = ArabTeX | TeX deriving (Enum, Show)

instance Encoding ArabTeX where
  encode _ = error "'encode' is not implemented"
  decode _ = concat . parseFull decoderParsing .
             concat . parseLongest decoderMapping

type Parsing = Extend Env [Char] ([UPoint] -> [UPoint])
type Environ = Env [Char]
type Mapping = Mapper Char [[Char]]
type LowerUp = Map [Char] [UPoint]

The environment Environ is needed to store information bound to the context—otherwise, parsing rules would become complicated and inefficient.

data Env i = Env { envQuote :: Bool, envMode :: Int,
                    envWasla :: Bool, envVerb :: Bool,
                    envEarly :: [i] }

setQuote q (Env _ m w v e) = Env q m w v e
setMode m (Env q _ w v e) = Env q m w v e
setWasla w (Env q _ _ v e) = Env q m w v e
setVerb v (Env q m _ _ e) = Env q m w v e
setEarly e (Env q m _ _ _) = Env q m w v e

instance ExtEnv Env where
  initEnv = Env False 0 False False []

Note that the decode method ignores the encoding parameter now. If our definitions were slightly extended, the ArabTeX data type could be parametrized with Env to allow user’s own setting of the initial parsing environment, passed to Encode.Extend.parseFull’.

Lexicon  The design of the lexicon cannot be simpler—the presented lexicon is nearly complete, but can be easily modified and extended. Lexical items are referred to in the mapping and the parsing phases by the sets they belong to, thus robustness is achieved.

define :: [[[Char], [Int]]] -> LowerUp
define l = makeMapWith const [ (x, map toEnum y) | (x, y) <- l ]

consonant :: LowerUp
consonant = unionMap [sunny, moony, bound]

sunny = define [
  ( "c", [ 0x062A ] ), ( "s", [ 0x0634 ] ),
  ( "t", [ 0x062B ] ), ( "\_s", [ 0x0635 ] ),
  ( "d", [ 0x062F ] ), ( "\_d", [ 0x0636 ] ),
  ( "\_d", [ 0x0630 ] ), ( "\_t", [ 0x0637 ] ),
  ( "\_z", [ 0x0631 ] ), ( "\_z", [ 0x0638 ] ),
  ( "z", [ 0x0632 ] ), ( "\_z", [ 0x0644 ] ),
  ( "s", [ 0x0633 ] ), ( "\_s", [ 0x0646 ] )
]
invis = define [ ( ",", [ ] ) ]
empty = define [ ( "", [ 0x0627 ] ) ]
shadda = define [ "\_s", [ 0x0651 ] ]
silent = define [
  ( "A", [ 0x0627 ] ), ( "N", [ 0x0627 ] )
]
wasla = define [ ( "W", [ 0x0671 ] ) ]
taaaa = define [
  ( "T", [ 0x0629 ] ), ( "\_H", [ 0x0629 ] )
]
bound = define [
  ( "A", [ 0x0622 ] ), ( "\_W", [ 0x0624 ] ),
  ( "\_A", [ 0x0623 ] ), ( "\_y", [ 0x0626 ] ),
  ( "\_i", [ 0x0625 ] ), ( "\_/", [ 0x0621 ] )
]
moony = define [
  ( "\_\", [ 0x0621 ] ), ( "\_f", [ 0x0641 ] ),
  ( "\_b", [ 0x0628 ] ), ( "\_g", [ 0x0642 ] ),
  ( "\_g", [ 0x062C ] ), ( "k", [ 0x0643 ] ),
  ( "\_h", [ 0x062D ] ), ( "\_m", [ 0x0645 ] ),
  ( "\_\_h", [ 0x062E ] ), ( "\_h", [ 0x0647 ] ),
  ( "\_t", [ 0x0639 ] ), ( "\_w", [ 0x0648 ] ),
  ( "\_g", [ 0x063A ] ), ( "y", [ 0x064A ] ),
  ( "\_b", [ 0x0640 ] ), ( "\_c", [ 0x0681 ] ),
  ( "\_c", [ 0x0686 ] ),
  ( "\_p", [ 0x067E ] ), ( "\_c", [ 0x0685 ] ),
  ( "\_v", [ 0x06A4 ] ), ( "\_h", [ 0x06AD ] ),
  ( "\_q", [ 0x06A4 ] ), ( "\_w", [ 0x06B5 ] ),
  ( "\_z", [ 0x0698 ] ), ( "\_r", [ 0x0695 ] )
]
vowel = define [
  ( "a", [ 0x064E ] ), ( "\_a", [ 0x0670 ] ),
  ( "i", [ 0x0650 ] ), ( "\_i", [ 0x0656 ] ),
  ( "u", [ 0x064F ] ), ( "\_u", [ 0x0657 ] )
]
Mapping  The Encode Mapper tokenizes the input string into substrings that are items of the lexicon, or, which is very important, rewrites and normalizes the notation in order to make proper parsing clearer. It guarantees the longest match, no matter what order the rules or the lexicon are specified in.

decoderMapping :: Mapper Char [[Char]]
decoderMapping = defineMapping
  ( pairs [ sunny, moony, invis, empty, taaa, silent, vowel, multi, nuuns, other, sukun, shadda, digit, punct, white ] )
  <> rules
  <> "\n" |.| error "illegal symbol"

rules :: Mapping
rules = "aN_A" |.-| "aNY" |.:| [] |.+|
  "_A" |.-| "y" |.:| [] |

  |.+ ruleVerbalSilentAlif |.+ ruleInternalTaaa
  |.+ ruleLiWithArticle |.+ ruleDefArticle
  |.+ ruleIndefArticle
  |.+ ruleMultiVowel |.+ ruleHyphenedVowel
  |.+ ruleWhitePlusControl |.+ ruleIgnoreCapControl
  |.+ ruleControlSequence |.+ rulePunctuation
In the rules, for instance, care of the silent *alif* after the ending "aN" is taken (Figure 6.2, ex. d–f), or the variants of definite article notation are unified (ex. k–l). So is the notation for long vowels, which offers freedom to the user, yet must strictly conform to the context of the next syllable in orthography (ex. h–k). \TeX's control sequences and spaces are normalized, too.

```latex
ruleIndefArticle =
  anyof [ c + m ++ "aNY" | m + m ++ "aNY" | : [ c ] ] |
  c + m ++ "aNA" | m + m ++ "aNA" | : [ c ] |
  c + m ++ "aN" | m + m ++ "aN" | : [ c ] |
  c <- elems [sunny, moony],
  m <- ["", "-", ",", ",(" ] |
  anyof [ v ++ "\"" ++ m ++ "aN" | |
    m ++ "aN" | [ v, "\", "\"] | + |
  v ++ "\"" ++ m ++ "aN" | |
    m ++ "aN" | [ v, "\"] | + |
  v <- ["A", "a"], m <- ["", "-", ",", ",(" ] ]

ruleDefArticle =
  anyof [ "l" ++ "-" ++ c ++ c | |
    "-" ++ c | : [ c ] |
  c <- elems [sunny, moony] ]

ruleMultiVowel =
  "iy" | |
    "I" | [ ] | + |
  "iy" | |
    "yy" | [ "I" ] | + |
  "uw" | |
    "U" | [ ] | + |
  "uw" | |
    "uw" | [ "U" ] | + |
  "aa" | |
    "A" | [ ] |
  anyof [ "iy" ++ v | |
     "y" ++ v | [ ["\"] | + |
  "uw" ++ v | |
     "w" ++ v | [ [\"] | + |
  v <- elems [vowel, multi, nuuns, other] ++ quote [vowel, multi, nuuns, other, sukun] ]

ruleControlSequence =
  do x <- sym ' \ '</ >
     some (anySymbol (["A".."Z"] ++ ["a".."z"]))
     many whites
     return [x]
```

The list comprehension syntax in Haskell allows us to write rules in form of templates, where we can iterate over the elements of the lexical sets and give them symbolic names. In the last example of a Mapping rule, we combine consonants *c* and short vowels *v*, the latter possibly preceded by a quote "\"".

```latex
ruleLiWithArticle =
  anyof [ "#" ++ v ++ "-a" ++ c ++ "-" ++ c | |
    "#" ++ v ++ c ++ "-" ++ c | [ ] |
  c <- elems [sunny, moony], c /= "#",
```

This rule alleviates a limitation in the original Arab\TeX{}'s coding of the prefixed words "\textit{li}" and "\textit{la}" when followed by a definite article. Due to an exceptional convention in orthography (cf. Lagally, 2004, sec. 4.1), "\textit{li-al-mawzi}" is not acceptable, and one has to write "\textit{lil-mawzi}" instead. Further complication comes with "\textit{lil}", so "\textit{lil-lawzi}" has to be transformed to "\textit{li-llawzi}.

With the rule \texttt{LiWithArticle} rewriting, one need not distinguish these anymore, and can just join words, like in other cases:

\begin{verbatim}
li-*ayni li-*ayna
"li-*la_dayni li-*la_dyna" li-*la_dayni li-*la_dyna
"li-al-*la_dayni li-*alla_dyna" li-*la_dayni li-*la_dyna
bi-*ayni bi-*ayna
"bi-*la_dayni bi-*alla_dyna" bi-*la_dayni bi-*la_dyna
\end{verbatim}

\section{Parsing}

The result of complete parsing is \texttt{[UPoint]}. However, to avoid inefficient list concatenations, the simpler parsers being combined produce `show' functions of type \texttt{([UPoint] -> [UPoint])}, composed sequentially with \texttt{plus}.

\begin{verbatim}
defcoderParsings :: Extend Env [Char] [UPoint]
defcoderParsings = (fmap (foldr ($) [])) . again) $  
parseHyphen     |\> parseHamza
|\> parseDefArticle
|\> parseDoubleCons  |\> parseSingleCons
|\> parseInitVowel
|\> parseWhite      |\> parsePunct
|\> parseDigit
|\> parseQuote      |\> parseControl

infixr 7 `\texttt{plus}'
-- infixr 9 .
-- infixr 5 ++

plus :: (a -> b) -> (c -> a) -> c -> b
plus = (.)
\end{verbatim}

Unlike \texttt{decoderMapping}, ordering of rules in \texttt{decoderParsings} does matter. The \texttt{again} and \texttt{|\>} combinators try the parsers in order, and if one succeeds, they continue again from the very first parser.
parseHyphen = do lower ["-" ] []
  resetEnv setEarly []
  parseNothing

This one is rather simple. The lower parser consumes tokens that are specified in its first argument, and returns to the input the tokens of its second argument. Thus, "-" is removed from the input, the memory of previous input tokens is erased with setEarly, and no new output is produced, i.e. parseNothing = return id.

Parsing an assimilated definite article is perhaps even more intuitive, once ruleDefArticle is in effect. We look for any consonant c followed by a hyphen "-" and the same consonant. If we succeed, we return the two consonants back to the input, as they will form a regular ‘syllable’ eventually. We look up the translation for the letter "ı" in the sunny set, and make it the output of the parser.

parseDefArticle = do c <- oneof [consonant]
  lower ["-", c] [c, c]
  upper ["ı"] [sunny]

The compilation of ‘syllables’ in the Arabic script rests in putting vocalization marks (vowel, sukan, shadda, etc.) onto the ‘consonantal’ letters. Processing of these marks is subject to the settings of the environment, in particular the envQuote and envMode values. We generalize upper to upperWith, to allow this processing.

parseDoubleCons =
  do c <- oneof [consonant, taaa, invis, silent]
    lower [c] []
    x <- upper [c] [consonant, taaa, invis, silent]
    y <- upperWith shaddaControl ["*" ] [shadda]
    parseSyllVowel c (x ‘plus’ y)

parseSyllVowel :: [Char] -> ([UPoint] -> [UPoint])
               -> Parsing

parseSyllVowel c x =
  do v <- parseQuote <|> parseNothing >>
      oneof [vowel, multi, nuuns, other] <|>
      return ""
      y <- upperWith (vowelControl c)
        [v] [vowel, multi, nuuns, other, sukan]
      completeSyllable [c, v] (x ‘plus’ y)

completeSyllable :: [[[Char]]] -> ([UPoint] -> [UPoint])
                   -> Parsing

completeSyllable l u = do resetEnv setQuote False
                          resetEnv setWasla True
                          resetEnv setEarly (reverse l)
                          return u

The definitions of vowelControl and shaddaControl are straightforward. The parseSingleCons and parseInitVowel parsers go in the spirit of their namesakes that we have seen. The
parseControl parser interprets control sequences that affect the parsing environment, including possible localization/nesting of the settings.

The last non-trivial parser is parseHamza. It does not produce any output, but computes the so-called carrier for the hamza consonant. In the \texttt{\setverb} mode, this carrier appears in the input after \texttt{'''}, or \texttt{'-'}, or \texttt{''}. In the complementary \texttt{\setarab} mode, this carrier must be determined according to some rather complex orthographical rules. In either case, the hamza combined with the carrier is distributed back to the input.

\begin{verbatim}
parseHamza = do h <- oneof [hamza]
    e <- inspectEnv
    let combineWithCarrier = if envVerb e
        then parseVerbHamza h
        else parseArabHamza h
    ;
    do lower [h] []
        b <- combineWithCarrier
        lower [] [b, b]
    <|>
    do lower ["-", h] []
        b <- combineWithCarrier
        lower [] [b, "-", b]
    <|>
    do b <- combineWithCarrier
        lower [] [b]
parseNothing

parseVerbHamza :: [Char] -> Extend Env [Char] [Char]
parseVerbHamza h =
    do i <- inspectIList
        case i of
            _ -> returnIList [(h ++ i) : y]
    oneof [bound]
\end{verbatim}

We provide the definition of parseArabHamza in the Appendix, for we believe it has never been published in such a complete and formal way. The algorithm essentially evaluates the position of the hamza in the word, and the context of vowels and consonants.

### 6.6.3 Encode Arabic ArabTeX ZDMG

This module relates the ArabTeX notation and the ZDMG phonetic transcription. The organization of the module is very similar to the previous one.

\begin{verbatim}
module Encode.Arabic.ArabTeX.ZDMG (ZDMG ..)) where

import Encode
import Encode.Mapper
import Encode.Extend
import FunParsing.OrdMap
\end{verbatim}
Let us therefore only show how capitalization is implemented. The lexicon stores dia-
critized lowercase characters used as the standard phonetic transcription. Capitaliza-
tion is possible (Figure 6.2, ex. k), but hamza ‘‘ and ymn ‘‘ are ‘transparent’ to it and
let capitalize the following letter.

```haskell
minor = define [("’", [0x02BE]), ("‘", [0x02BF])]
sunny = define [("t", [0x0074]), ("_t", [0x0074, 0x031]), ("d", [0x0064]),
                ("_d", [0x0064, 0x031]), ("z", [0x0072]), ("z", [0x007A]),
                ("s", [0x0073]), ("s", [0x007A, 0x030C]), (".s", [0x0073, 0x0323]),
                (".d", [0x0064, 0x0323]), (".t", [0x0074, 0x0323]),
                (".z", [0x007A, 0x0323]), ("l", [0x006C]), ("n", [0x006E])]

parseSingleCons =
  do c <- oneof [consonant, extra, invis]
     x <- upperWith consControl [c] [consonant, extra, invis]
     resetEnv setCap False
     parseSyllVowel c x
     <|
  do c <- oneof [minor]
     x <- upper [c] [minor]
     parseSyllVowel c x

consControl :: OrdMap m => [Char] -> [m [Char] [Upoint]] -> Environ -> [[Upoint]]
consControl x l e = if envCap e
    then [capFirst n | n <- noChange ]
    else noChange
  where noChange = lookupList x l
       capFirst [] = []
       capFirst (x:xs) = (toEnum . flip (-) 0x0020 . fromEnum) x : xs
```
6.7 Discussion

Some extensions with respect to the original ArabTEX notation are done in this module, too. Depending on the grammatical status of a word, different pronunciation of the \( t\bar{a} \text{- marb} \bar{u}t\bar{ah} \) ending occurs—either "T", or silent "H"—while the script represents the ending always the same (Figure 6.2, ex. 1).

\[
\begin{align*}
\text{ruleInternalTaaaa} = \\
&\text{anyof} \ [ "H" \ +\ v \ |\ | "T" \ +\ v \ +\ c \ |\ -\ "T" \ +\ v \ +\ c \ |\ :\ |\ ]
\end{align*}
\]

6.7 Discussion

Next to the original ArabTEX parser (Lagally, 1992, 2004), there is an implementation in Perl of the Encode Mapper and Encode Arabic modules (Smrz, 2003–2006) with which the interpreter is built as a multi-layer finite-state automaton. The method used there, however, does not achieve the elegance, clarity nor flexibility as the presented Haskell implementation. Lazy construction of the automaton and the power of functional combinator parsing is simply missing there.

The significance of the ArabTEX notation, devised with modifications also for languages other than Arabic, in lexicography, linguistics, and education is discussed in (Lagally, 1994).

Our motivation for developing this approach is the use of the notation in a computational system for language modeling (Smrz, 2006), inspired by and reusing the Functional Morphology library (Forsberg and Ranta, 2004). Further extensions of our work are expected, and inclusion of the programming library in various information processing systems is a possible next application.

6.8 Conclusion

In the first part of this paper, we presented two functional algorithms for deterministic longest-match parsing, and developed variants of the generalized parsers due to (Ljunglöf, 2002). In the second part, we implemented an independent and highly reusable and extensible interpreter for the ArabTEX notation of Arabic (Lagally, 2004), presented most of the code in Haskell and discussed the key ideas of the modular design.
Appendix

This is the complete parser for determining the carrier of *hamza* from the context, according to the rules of Arabic orthography.

```haskell
parseArabHamza :: [Char] -> Extend Env [Char] [Char]
parseArabHamza h =
  do  e <- inspectEnv
     b <- prospectCarrier
     let carryHamza = case envEarly e of
       []       -> case b of
          "'y"     -> "'i"
          "'i"     -> "'i"
          "'A"     -> "'A"
          _        -> "'a"
          "'a":_   -> "'y"
          "'y":_   -> "'y"
          "'I":_   -> caseofMultiI b
          "'I":_   -> caseofMultiI b
          "'I":_   -> caseofMultiI b
          ["", "y"] -> caseofMultiI b
          "'u":_   -> caseofVowelU b
          "'u":_   -> caseofVowelU b
          "O":_    -> caseofVowelU b
          "'U":_   -> caseofMultiU b
          "'U":_   -> caseofMultiU b
          "O":_    -> caseofMultiU b
          "'U":_   -> caseofMultiU b
          "'a":_   -> caseofVowelA b
          "'a":_   -> caseofVowelA b
          "'A":_   -> caseofMultiA b
          "'A":_   -> caseofMultiA b
          ["", "'A"] -> caseofMultiA b
          "":_     -> case b of
            "'y"     -> "'y"
            "'w"     -> "'w"
            "'A"     -> "'A"
```

6.8. CONCLUSION

case carryHamza of

"A" -> lower ["A"] []
_ -> return []

return carryHamza

where
caseofMultiI b = case b of
  "i" -> "j"
  "j" -> "j"
_ -> "y"

caseofMultiU b = case b of
  "i" -> "j"
  "j" -> "j"
  "y" -> "y"
_ -> "w"

caseofMultiA b = case b of
  "y" -> "y"
  "w" -> "w"
_ -> "i"

caseofVowelU b = case b of
  "y" -> "y"
_ -> "w"

caseofVowelA b = case b of
  "y" -> "y"
  "w" -> "w"
  "i" -> "i"
  "A" -> "A"
_ -> "a"

prospectCarrier = do parseQuote
    b <- lookaheadCarrier
    lower [] ["\\""]
    resetEnv setQuote False
    return b
<|> lookaheadCarrier

lookaheadCarrier =

do v <- oneof [multi, other]
  let carryHamza = case v of
def carryHamza(vowel, nuuns):
    carryHamza = case of
        "i" -> "'y"
        "iN" -> "'y"
        "_i" -> "'y"
        "e" -> "'y"
        "U" -> "'w"
        "uN" -> "'w"
        "_U" -> "'w"
        "O" -> "'w"
        "A" -> "'A"
        _ -> "'a"
    lower [] [v] = return carryHamza
<|>
    do v <- oneof [vowel, nuuns] <|> return ""
c <- oneof [sunny, moony, taaaa,
invis, silent]
    let carryHamza = case v of
        "i" -> "'i"
"iN"      -> "'i"
"_i"      -> "'/i"
"_e"      -> "'/i"
_         -> "'/i"

\begin{verbatim}
case v of ""  -> lower [] []
         _  -> lower [] [v]
\end{verbatim}

\textbf{return} carryHamza

Many thanks to the authors of the listings, pgf, arabetx, and acolor packages for \LaTeX.
Um, ah, you mean, um
I think that’s a sort of surreal masterpiece
It has a semi-autobiographical feel about it
A bit of a mixture, you mean, mish-mash
That’s the word I was looking for

Chapter 7

Functional Morphology

7.1 On Hamza Writing

As we know, the carrier of hamza is determined by her position and adjacency. Such decision-making is up to ArabıpX. But it fully rests with us whether to indicate hamza or not!

Let us formulate the criteria of hamza writing as observed from and present some remarkable examples. This study was inspired by and its earlier versions, although and since they do not keep to it systematically.

Being one of the moon consonants, hamza may occur

- in the root of a word: إحدى أخذ أياضا إهالة متمس بالرأي أو لي قرأ آلالات رئيسية أثر

- as a morphological consonant

  - in noun-like words: أربعة أصغر حمزة أعمى أداب آثار ألبسة ألوان أنظار

  - in verbs and masdars of 4th stem and in every 1st person singular imperfect form of a verb: أقترح أعلم أكتب لإجام لإقامة إنتاج أور أشر أرسل أرود أمه أجلس أستطيع

- in other words: أولئك أيها أو إذا كنت أمًا كنت أين إلا أن إن إلى

On the contrary, the hamza mark does not belong before so called auxiliary vowels (otherwise wasla could not be introduced)

- in definite articles: al-qAhiraTu أَلْفَاهْرَةُ al-qāhiratu, mina al-qAhiraTi مَنْ أَلْفَاهْرَة مَنْ اَلْفاهِرةَ ad-dawā‘i, ‘ani ad-dawā‘i

- in prefixes of 7th to 10th stem: in.tilAquN إِنْفَلَاقَةٌ intilāqun, inta.zara إِنْتَظَرْ لَاقَةٌ intażara, u’tubira أَعْتَبِرَ utubira, i_hTara إِخْتَارَ izdāda, i`starY إِشْتَرَى ištāra, idda’y إِذْعَى iddāa, itta.hada إِنْتَحَلَى ittaḥada,
7.2 Error Messages

Check the justification of the ending -AT-! Such an ending is alright in words like fatATuN fatâtun, .hayATuN hayâtun, .salATuN salâtun having wâw or yâ’ as the final consonant of their root. These semi-vowels usually reappear in plural, e.g. fatayATuN fatayâtun, .hayawATuN hayawâtun, .salawATuN salawâtun.

If we are not dealing with a form of a verb, then you may have made a typing error in the second last character of the word and wrote t instead of T! Confined by its working methods, ArabSpell is unable to say that sakata sakata versus sanaTa sanata, yatalaffatu yatalaffatu versus ma’gallaTu magallatu, fa-darasati al-fatATu ‘l-fatâtu versus li-dirAsaTi al-lu.qATi li-dirâsati ‘l-luğâtî are all correct expressions.

If the word is fully vocalized and if we are not dealing with an apocopate or an imperative, then there should come an indefinite article N instead of the final character n! Longer, good-looking words1 with the exception of 1 akin لَكَن
lakin and ‘i DAN دان (or ‘i DAN دان) generate this tanwin warning automatically. It may bother you in the indicated cases.

Let us have a look at some examples. From sakana سكنا, the apocopate yaskun يسكان and the imperative uskun عسكن can be derived. ArabSpell objects if you replace ماتقنااتقنا by lam yatqin لام يتعن. ‘تقنااتقنا is no better. Similarly, yakun يكنا from كان is the stumbling block, however kun كن is too short for ArabSpell to expose to your scrutiny.

The word might possibly end with a silent ‘alif –UA! The detailed analysis is given in the documentation. Here it comes! In general, occurrence of –U at the very end of the word is scarce. There is a set of words called al-asma al-hamsah taking this shape: ‘abU أبو, ‘اHU أه، .hamU حم، fU ف, _DU ج. In addition, final –U can show up as a reduced form of the masculine plural ending –Una in genitive phrases, e.g. mu’allimU al-madrasaTi معلم المدرسة, _dawU ^sa’nIN داون سان, _dawU al-‘iq.TA’i dawU الـيقطي, _ULU al-‘amRI الايامري. As for verbs, imperfect indicatives with وام as the final root consonant have no reason to take silent ‘alif either: ‘ar^gU أرج, tad’U تد، ya^skU نامك, nam.HU نامه.

The word might possibly end with a silent ‘alif –aWA or –aw! The detailed analysis is given in the documentation. There is always a lot of fun about Arabic irregular roots. For instance, the nominative plural of mustawlin مستولون mustawlin reads mustawlUna مصطلحون which is quite a digestible word, whereas in the passive participle mustawlaNY مصطلحون mustawlan becomes mustawlawna مصطلحون, or mustawlaw mustawlaw in a genitive phrase.

Because of examining rAW زوا raw and all its look-alikes, ‘aw أو, _aw or law لوي fall into the trap, too.

- If you use hyphens in other than suggested functions, the words might not be spelled properly. We will try to fix this in the forthcoming versions.
- On account of morphological ambiguity, hamza correction system cannot be complete. In questionable cases, see the Appendix for reference.
sū'ala saw'ala rī'ula ray'ula sā'ula sā'ila sū'ila rī'a
sū'ala saw'ala rī'ula ray'ula sā'ula sā'ila sū'ila rī'a

\setverb a's-а'samsu wa ru'wisa wa \setarab ru'’isa
aš-samsu wa ru-'wisa wa ru'-isa

м_I’aTuN y’Am’Afi’A lI’AnA мI’aTuN yamafafirā
li’ānā
m_I’aTuN мI’aTuN m_o’aTuN mO’aTuN moatun m_u’aTuN мa mIuatun
mU’alaTuN мI’atun daw’alaTuN dawalatun. daw’aN dawan
muwaa`gQahQQaHuN muwaa`gahahun qiyaam qiya'am
muw|”а|gahaHuN muwagahahun qiyaAnuN qiya‘anun
mu|wa`gahaTuN muwagahatun
katabuWA katabuW katabuA katabA katabā
katabā
ramaWA ramaW ramaW ramaW ramaW ramaW ramaW ramaW
ramaWA ramaW ramaW ramaW ramaW ramaW ramaW ramaW
ramaW”A ramaW”A ramaW”A ramaW”A ramaW”A ramaW”A ramaW”A ramaW”A
al-laylaT линия l-laylaT линия l-laylah линия li-laylah
al-l ylim l-ylim lylim ar-rrif ər-rrif l-rrrif l-rrrif
r-rrrif
Lexical closure means that an anonymous function carries its native environment wherever it goes, just like some tourists I have met.

Mark Jason Dominus, *Higher-Order Perl*

Chapter 8

# Lexicon versus Treebank

## 8.1 Functional Description of Language

Let us give a rough outline of the structure of linguistic description in FGD and motivate our specific concerns about Arabic within PADT.

The theory of Functional Generative Description (Sgall et al., 1986, Sgall, 1967, Páněnová, 1980, Hajičová and Sgall, 2003) stresses the principal difference between the form and the function of a linguistic entity, and recognizes these entities as the building blocks of the respective level of linguistic description—be it underlying or surface syntax, morphemics, phonology or phonetics.

In this theory, a *morpheme* is the least unit representing some linguistic meaning, and is understood as a function of a *morph*, i.e. a composition of phonemes in speech or orthographical symbols in writing, which are in contrast the least units capable of distinguishing meanings.

Similarly, morphemes build up the units of syntactic description, and assume values of abstract categories on which the grammar can operate. In FGD, this very proposition implies extremely complex concepts, introduced with their own terminology and constituting much of the theory. For our purposes here, though, we would only like to reserve the generic term *token* to denote a syntactic unit, and consider any necessary distinctions and offer the relevant and more precise definitions in the later sections.

The highest abstract level for the description of linguistic meaning in FDG is that of underlying syntax. It comprises formal means to capture all communicative aspects of language, including the form of an utterance as well as the communication structure of the discourse.

In the theoretical model, the representation of the underlying syntax is able to generate the other information of interest to linguists, in particular the surface syntactic structure of a sentence and its linear sequence of phonemes or graphemes.

In the series of Prague Dependency Treebanks (Hajič et al., 2001, 2006, Cuřín et al., 2004) it seems important to note that the assignment of function to form is arbitrary, i.e. subject to convention—while Kay (2004) would recall *l’arbitraire du signe* in this context, Hodges (2006, section 2) would draw a parallel to *wād* convention.
2004, Hajic et al., 2004a), this transfer from the level of meaning into the level of realization is inverse and elaborated into a different set of levels, with minor modifications to the theory: the highest representation level is called tectogrammatical, then there is the level of analytical syntax, and the level of morphological analysis.

Linguistic data, i.e. mostly texts in their original written form, are gradually analysed in this system of levels, and their linguistic meaning is thus reconstructed and made explicit.

Morphological annotations identify the textual forms of a discourse lexically and recognize the grammatical categories they assume. Processing on the analytical level describes the superficial syntactic structures present in the discourse, whereas the tectogrammatical level reveals the underlying ones and restores the linguistic meaning.

The levels of Prague Arabic Dependency Treebank (Hajic et al., 2004a,b, 2005, Smrz et al., 2006) are not yet ideal—there are only the morphological level and the analytical level of annotations. The tectogrammatical representation, with the annotation of information structure, valency, anaphora resolution and detailed lexical disambiguation (cf. Šgall et al., 2004), is being prepared.

```
module Lexicons.Lexicon28 where

import LexiPhone

version = revised "$Revision: $"

lexicon = listing "Lexicon properties"

| > "yA" <| [ ]
| > "yA" <| [ ]
| > "yA" <| [ ]
| > "yA'" <| [ ]
| > "yAb" <| [ ]
| > "yAbAn" <| [ ]
```

Revision of
August 7, 2006
8.1. FUNCTIONAL DESCRIPTION OF LANGUAGE

| "yAbAn" <| |
|-------|
literal "yAbAn" <| Iy {- yAb -} {- yAbAniy˜ -} |

| "yAt" <| |
|
| "yAtuwm" <| |
|-------|
literal "yAtuwm" {- yAt -} {- yAtuwm -} |

| "yArd" <| |
|
| "yArd" <| |
|-------|
literal "yArd" <| aT {- yArd -} {- yArodap -} |

| "yArk" <| |
|
| "yArkAs" <| |
|-------|
literal "yArkAs" {- yArk -} {- yArokAs -} |

| "yAzˆg" <| |
|
| "yzˆg" <| |
|-------|
CACiC <| Iy {- yzˆg -} {- yAzijiy˜ -} |

| "yAs" <| |
|
| "ya’is-ai" <| |
|-------|
literal "ya’is-ai" {- yAs -} {- ya}is-ai- -} |

| "y’s" <| |
|-------|
HaCCaC {- y’s -} {- >ayo>as -} ,
I斯塔CaC {- y’s -} {- {isotayo>as -} ,
CaCC {- y’s -} {- ya>os -} ,
CaCAC <| aT {- y’s -} {- ya|sap -} ,
8.2 Analytical Dependency Description


وفي ملف الأدب طرحت المجلة قضية اللغة العربية والأخطار التي تهددها. ويرى القانون على الملف أن ما تعرض له اللغة العربية له أهداف محددة منها إبعاد العرب عن لغتهم ومزاحمة اللغات الغربية لها وهو ما يعني ضعف الصلة بها ومحاولة إزاحة اللغة الفصحى بكل الوسائل وإحلال اللهجات المختلفة في البلاد العربية عمها.

‘In the section on literature, the magazine presented the issue of the Arabic language and the dangers that threaten it. The ones in charge of the section are of the opinion that what the Arabic language is exposed to has its specific goals, including the separation of Arabs from their language and the competition of the Western languages with it, which means weakness of the link to it, and the attempt to remove the literary language by all means and to replace it with the different dialects of the Arab world.’

(Žabokrtský, 2005)
(Lopatková et al., 2005)
(Debusmann et al., 2004a,b, 2005a)
(Bojar, 2004)
It was not easy for him to face the television cameras and the lenses of photographers as he was getting on the bus.
And according to first statistics, ten houses were destroyed completely and fifteen partially.
Figure 8.3  Analysis of clausal structure of the example text (see page 73).
Continuing from Fig. 8.3

Figure 8.4 Analysis of coordinations.
(Debusmann et al., 2005b)
(Kruijff and Duchier, 2002, Nivre, 2005)
(El-Shishiny, 1990, Pedersen et al., 2004)
Chapter 9

MorphoTrees

The classical concept of morphological analysis is, technically, to take as input a substring of some linear representation of a discourse and produce a list of other, different linear strings, each of which delivers one hypothetical reading of the input in terms of the underlying lexical units and morphs enriched with some abstract labels revealing the process of derivation of the input from the lexical units, like in Figure 9.1.

The practice has been, at least in Arabic, that the output information is not organized any further. The divergent analyses are not clustered together according to their common features, and the output strings are linear in structure and need explicit parsing. It is very difficult for a human to interpret the analyses and to discriminate among them. For a machine, it is undefined how to compare the differences of the analyses, as there is no disparity measure other than unequalness.

MorphoTrees (Smrz and Pajas, 2004) is the idea of building effective and intuitive hierarchies over and among the input and output strings of morphological systems. It is especially interesting for Arabic and the Functional Arabic Morphology, yet, it is in no sense limited to either of these. Section 9.3 discusses this further.

9.1 The MorphoTrees Hierarchy

As an inspiration for the design of the hierarchies, let us consider the following analyses of the string $\text{فَحَم}$ $\text{فهم}$. Some readings will interpret it as just one token related to the notion of understanding, but homonymous for several lexical units, each giving many distinct inflected forms written like this. Other readings will decompose the string into two co-occurring tokens, the first one, in its unvocalized form $\text{ف} \text{ه}$, standing for an unambiguous conjunction, and the other one, $\text{هم}$ $\text{هم}$, analyzed as a verb, noun or pronoun, each again ambiguous in its functions.

Clearly, this type of concise and “structured” description does not come ready-made—we have to construct it on top of the overall morphological knowledge. We can take the output solutions of morphological analyzers and process them according to our requirements on tokenization and ‘functionality’ stated above. Then, we can merge the analyses and their elements into a five-level hierarchy similar to that of Fig-
9.1. The MorphoTrees Hierarchy

<table>
<thead>
<tr>
<th>Morphs</th>
<th>Form</th>
<th>Token Tag</th>
<th>Lemma</th>
<th>Glosses per Morph</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{laY+(null)}</td>
<td>\text{alā}</td>
<td>VP\text{-A-3MS}--</td>
<td>\text{alā}</td>
<td>promise/take an oath + he/it</td>
</tr>
<tr>
<td>\text{liy}^-</td>
<td>\text{alty}</td>
<td>A----------</td>
<td>\text{alty}</td>
<td>mechanical/automatic</td>
</tr>
<tr>
<td>\text{liy}^-u</td>
<td>\text{alTy-}</td>
<td>A----------1R</td>
<td>\text{alTy}</td>
<td>mechanical... + [def.nom.]</td>
</tr>
<tr>
<td>\text{liy}^-i</td>
<td>\text{alTy-}</td>
<td>A----------2R</td>
<td>\text{alTy}</td>
<td>mechanical... + [def.gen.]</td>
</tr>
<tr>
<td>\text{liy}^-a</td>
<td>\text{alTy-}</td>
<td>A----------4R</td>
<td>\text{alTy}</td>
<td>mechanical... + [def.acc.]</td>
</tr>
<tr>
<td>\text{liy}^-N</td>
<td>\text{alTy-}</td>
<td>A----------1I</td>
<td>\text{alTy}</td>
<td>mechanical... + [indef.nom.]</td>
</tr>
<tr>
<td>\text{liy}^-K</td>
<td>\text{alTy-}</td>
<td>A----------2I</td>
<td>\text{alTy}</td>
<td>mechanical... + [indef.gen.]</td>
</tr>
<tr>
<td>l +</td>
<td>\text{al}</td>
<td>N----------R</td>
<td>\text{al}</td>
<td>family/clan</td>
</tr>
<tr>
<td>+ iy</td>
<td>- \text{I}</td>
<td>S----1-S2--</td>
<td>\text{I}</td>
<td>my</td>
</tr>
<tr>
<td>Oa+lIy+(null)</td>
<td>a-It</td>
<td>VI\text{A}-1-S--</td>
<td>\text{waliya}</td>
<td>I + follow/come after + [ind.]</td>
</tr>
<tr>
<td>Oa+lIy+a</td>
<td>a-lIy-a</td>
<td>V\text{I}-1-S--</td>
<td>\text{waliya}</td>
<td>I + follow/come after + [sub.]</td>
</tr>
</tbody>
</table>

Figure 9.1 Analyses of \text{AlY} \text{؟} \text{ل} \text{؟} \text{ل} \text{؟} transformed into the MorphoTrees hierarchy.

Figure 9.2, the leaves of which are the tokens and their tags as the atomic units, while its root represents the input string, or generally the input entity (some tree of discourse elements). Rising from the leaves up to the root, there is the level of lemmas of the lexical units, the level of non-vocalized standard orthographical forms of the tokens, and the level of decomposition of the entity into a sequence of such forms, implying the number of tokens and their spelling.

This is the definition of MorphoTrees appearing in PADT. Their instances are created algorithmically from the information originating in Tim Buckwalter’s morphology, like in the example in Figure 9.1, which includes the list of analyses for the string \text{AlY} \text{؟} \text{ل} \text{؟} along with the corresponding hierarchy (with the leaves depicted only schematically).

Let us note that the MorphoTrees hierarchy itself might serve as a framework for evaluating morphological taggers, lemmatizers and stemmers of Arabic, since it allows
Figure 9.2  Annotation of MorphoTrees using restrictions on particular values/categories in the morphological tags.
for resolution of their performance on the different levels, which does matter with respect to various applications.

9.2 MorphoTrees Disambiguation in TrEd

The linguistic structures that get annotated as trees are commonly considered to belong to the domain of syntax. It is due to the exceptional flexibility of TrEd, the general-purpose tree editor being used within many linguistic projects next to the treebanking of Czech and Arabic, that we could extend this environment easily and implement in it an extra annotation mode for MorphoTrees, too.

The annotation of MorphoTrees rests in selecting the applicable sequence of tokens that analyze the entity in the context of the discourse. In a naive setting, an annotator would be left to search the trees by sight, decoding the information for every possible analysis before coming across the right one. If not understood properly, the supplementary levels of the hierarchy would rather tend to be a nuisance . . .

Instead, MorphoTrees in TrEd take great advantage of the hierarchy and offer the option to restrict one’s choice to subtrees and hide those leaves or branches that do not conform to the criteria of the annotation. Furthermore, many restrictions may be applied automatically, and the decisions about the tree controlled in a very rapid and elegant way.

The MorphoTrees of the entity \texttt{fhm} \texttt{فم} in Figure 9.2 are in fact annotated already. The annotator was expecting, from the context, the reading involving a conjunction. By pressing the shortcut \texttt{c} at the root node, he restricted the tree accordingly, and the only one eligible leaf was selected at that moment. Nonetheless, the conjunction is a part of a two-token entity, and some annotation of the second token must also be performed. Automatically, all inherited restrictions were removed from the \texttt{hm} \texttt{هم} subtree (notice the empty tag in the flag over it), and the subtree unfolded again. The annotator moved the node cursor to the lemma for the pronoun, and restricted its readings to the nominative by pressing \texttt{1}. There were no more decisions to make, and the annotation proceeded to the next entity of the discourse.

Alternatively, the annotation could have been achieved merely by typing \texttt{s1}. The restrictions would unambiguously lead to the nominative pronoun, and then, without human intervention, to the other token, the unambiguous conjunction. These automatic decisions need no linguistic model, and yet they are very effective.

Incorporating restrictions or forking preferences sensitive to the surrounding annotations is in principle just as simple, however, the concrete rules of interaction may not be easy to find. Grammatical constraints on multi-token word formation are usually hard-wired inside analyzers and apply per entity, but certain restrictions might be pertinent even to adjacent tokens of successive entities, for instance. Eventually, annotation of MorphoTrees might be assisted with real-time tagging predictions provided by some independent computational module.
9.3 Discussion to MorphoTrees

The levels of MorphoTrees are extensible internally (More decision steps for some languages?) as well as externally in both directions (Analyzed entity becoming a tree of perhaps discontinuous parts of a possible idiom? Leaves replaced with derivational morphosyntactic trees of the morphs of the tokens?) and the concept brings a general view of many issues encompassed by morphological analysis and disambiguation.

In Arabic, whose MorphoTrees get on average 7.9 tokens per entity and 1.4 partitions per entity in some data sets (Hajic et al., 2004a), restrictions as a means of direct access to the solutions improve the speed of annotation significantly.

Hierarchization of the selection task is an improvement as well. In fact, it is the most important contribution of the idea. The particular meaning of the levels of the hierarchy mirrors the strategy for decision-making, and also the linguistic theory, neither of which are universal. The power of trees remains, though—efficient top-down search or bottom-up restrictions, gradual focusing of one’s interest, refinement, inheritance and sharing of information, etc.

Tokenization is to be further discussed. Some researchers pose this problem differently—they do not reconstruct the canonical non-vocalized forms as we do, but only determine token boundaries between the characters of the original string (cf. Diab et al., 2004, Habash and Rambow, 2005). Our point in doing the more difficult job is that (a) we are interested in such level of detail (b) disambiguation operations become more effective if the hierarchy reflects more distinctions (i.e. decisions are specific about alternatives).

The disparity of these tokenizations is illustrated in Figure 9.3. The graph on the left depicts the three ‘sensible’ ways of partitioning the input string $A_1Y$ in the approach of (Diab et al., 2004), where characters are classified to be token-initial or not. In the graph, boundaries between individual characters are represented as the numbered nodes in the graph. Two of the valid tokenizations of the string are obtained by linking the boundaries from 0 to 3 following the solid edges in the directions of the arrows. The third partitioning $A_1Y \varepsilon \varepsilon$ indicates that there is another fictitious boundary at
the end of the string, yielding some ‘empty word’ $\epsilon \in \Sigma$, which together corresponds to
leaping over the string at once and then taking the dashed edge in the graph.

Even though conceptually sound, this kind of partitioning may not be as powerful
and flexible as what MorphoTrees propose, because it rests in classifying the input char-
acters only, and not actually constructing the canonical forms of tokens as an arbitrary
function of the input. It cannot undo the effects of orthographical variation (Buckwalter,
2004b), nor express other useful distinctions.

The tree structure in Figure 9.3 illustrates these differences. The boundary-based
tokenizations are definitely not as detailed as those of MorphoTrees given in Figure
9.1, and might be thought of as another intermediate level in the hierarchy. But as they
are not linguistically motivated, we do not establish the level as such.

In any case, we propose to evaluate tokenizations in terms of the Longest Common
The tokens that are the members of the LCS with some referential tokenization, are
considered correctly recognized. Dividing the length of the LCS by the length of one of
the sequences, we get recall, doing it for the other of the sequences, we get precision.
The harmonic mean of both is $F_{\beta=1}$-measure (cf. e.g. Manning and Schütze, 1999).

```c
$current->{ 'hide' } = $current->{ 'apply_m' } > 0 ? 'hide' : '';
```

```c
else {
    while ($current = $current->{ following }) {
        $current->{ 'hide' } = 'hide if defined $current->{ 'tips' } and $current-
```

```c
else {
    while ($current = $current->{ following }) {
```

```c
    $current->{ 'hide' } = 'hide if
```

```c
    $current->{ 'hide' } = 'hide if
```

```c
    $current->{ 'hide' } = 'hide if
```
$current->{'hide'} = '' if defined $current->{'tips'} and $current->{'tips'};

($)nodes, $current) = $fsfile->nodes($index, $recent, $show_hidden);

@($nodes) = reverse @($nodes) if $main::treeViewOpts->(reverseNodeOrder);

return [[@($nodes), $current];

sub get_value_line_hook {
  my ($fsfile, $index) = @_;  
  my ($nodes, $words);

  my $tree = $fsfile->tree($index);

  if ($tree->{'type'} eq 'paragraph') {
    #bind annotate_morphology_click to Ctrl+space menu Annotate as if by Clicking
    sub annotate_morphology_click {
      annotate_morphology('click');
    }

    #bind switch_either_context Shift+space menu Switch Either Context
    sub switch_either_context {
      $Redraw = 'win' if $_[0] eq __PACKAGE__;

      my $quick = $_[0];
      my @$refs;

      if ($root->{'type'} eq 'paragraph') {
        if ($this->{'type'} eq 'paragraph') {
          GotoTree(['split /[0-9]+/, $root->{'par'}][0]);
        }
        elsif ($this->{'type'} eq 'word_node') {
          GotoTree($this->{'ref'});
        }
        else {
          $refs[0] = $this->{'ref'};
        }
      }
    }
  }
}
if ($this->type == 'lemma_id') {
    GotoTree($this->parent()->ref);
} else {
    GotoTree($this->parent()->parent()->ref);
}

$this = ($root->descendants())[$refs[0] - 1];

else {
    ($refs[0]) = $root->id =~ /([0-9]+)$/;
    $refs[1] = $this->ord unless $quick eq 'quick';
    GotoTree($root->ref);
    $this = ($root->children())[$refs[0] - 1];
    unless ($quick eq 'quick') {
        This is the end.
    }
}
Appendix A

Other Listings

Many thanks to the authors of the listings, pgf, arabtex, and acolor packages for \LaTeX.
Bibliography


Bibliography


Huet, Gérard. 2002. The Zen Computational Linguistics Toolkit. ESSLLI Course Notes, FoLLI, the Association of Logic, Language and Information.


BIBLIOGRAPHY


Kruijff, Geert-Jan M. and Denys Duchier. 2002. Formal and Computational Aspects of Dependency Grammar. ESSLLI Course Notes, FoLLI, the Association of Logic, Language and Information.


My Own Index

bytes, 8
characters, 8
conventions, 8
encoding, 8
orthography, 8
romanization, 8
symbols, 8
transcription, 8
transliteration, 8
Haskell Index

decode, 50, 52, 53, 60
encode, 50, 52, 53, 60
map, 27, 38, 39, 43, 44, 46, 49–51, 53
parse, 35, 39, 48
nolig, 39, 48
Perl Index

sub, 85