Encode Arabic: Exercise in Functional Parsing

Otakar Smrž

Institute of Formal and Applied Linguistics
Faculty of Mathematics and Physics
Charles University in Prague
otakar.smrz@mff.cuni.cz

Abstract

We present an implementation of a Haskell library for processing the Arabic language in the ArabTEX 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.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming

General Terms Functional Parsing, Natural Language Processing

Keywords Rewrite Rules, Longest Match Algorithms, Lexical Mapping, Parser Combinators, Encoding Scheme, ArabTEX

1. Introduction

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.

While the ArabTEX notation is extremely useful for instance in lexicography and linguistics, the interpreter for it, the ArabTEX's parser, cannot be directly applied to problems other than TEXing.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP '06 Haskell Workshop September 17, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM [submission version]. . . $5.00.

In the next section, we introduce the core constructs of the Pure Functional Parsing library (Ljunglöf, 2002), from which we depart. The third section addresses lazy deterministic parsing with possible input symbol 'rewriting' in Encode Mapper, for which we develop a breadth-first and a depth-first search algorithms to find the longest match. The fourth section provides Encode Extend, an implementation of the standard functional parser extended with environment setting during parsing. In the fifth section, we put these pieces together and instantiate these parsers to eventually form the Encode Arabic suite. After that we conclude our paper.

2. Functional Parsing

Parsing is the process of recovering structured information from a linear sequence of symbols. The formulation of the problem in terms of functional programming is well-known, and both excellent literature and powerful computational tools are available (Wadler, 1985, 1995, Hutton and Meijer, 1996, 1998, Swierstra and Duponcheel, 1996, Swierstra, 2001, Leijen, 2001, Leijen and Meijer, 2001, Marlow, 2001, 2003) .

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.

```
-- 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  <:>
infixl 3  <*>, *>
infixl 2  <+>
```

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.

```
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 `Monoid`s, to combine them in parallel, and also to belong to the class `Sequence`, in order to model their sequential composition.

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

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.

```
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.

```
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.

```
(<:>) :: Sequence m => m a -> m [a] -> m [a]
p <:> ps = fmap (:) p <*> ps

many :: (Monoid m, Sequence m) => m a -> m [a]
many p = ps
  where ps = return [] <+> p <:> ps

syms :: (Sequence m, Symbol m s) => [s] -> m [s]
syms []     = return []
syms (s:ss) = sym s <:> syms ss
```

3. 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 subtries 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.

```
module Encode.Mapper where

import FunParsing.OrdMap
import FunParsing.Parsers.Parser

data Mapper s a = [Quit s a] :&: Map s (Mapper s a)
                | forall b . FMap (b -> a) (Mapper s b)
type Quit s a = ([s], a)
```

A node in the `Mapper s a` trie is the tuple `:&:` of a list of results of type `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 `FunParsing.OrdMap` similar to `Data.Map` of the Haskell libraries.

The `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.

```
quit :: [s] -> Quit s a -> ([s], a)
quit r (s, a) = (s ++ r, a)

returnQuit :: [s] -> a -> Quit s a
returnQuit s a = (s, a)

justQuit :: Quit s a -> a
justQuit (s, a) = a

skipQuit :: Quit s a -> [s]
skipQuit (s, a) = s

fmapQuit :: (a -> b) -> Quit s a -> Quit s b
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 |-|            -- rule =     "a"          |.| 1
infix  3 |:|, |.|       --         |+| "a"          |.| 2
infixl 2 |+|            --         |+| "b" |-| "aa" |:| 3

(|:|) :: InputSymbol s => (a -> Mapper s a) -> a
                                            -> Mapper s a
(|:|) x y = x y

(|-|) :: InputSymbol s => [s] -> [s] -> a -> Mapper s a
(|-|) x y z = syms x >> [returnQuit y z] :&: emptyMap

(|.|) :: InputSymbol s => [s] -> a -> Mapper s a
(|.|) x y = x |-| [] |:| y
```

```
(|+|) :: InputSymbol s => Mapper s a -> Mapper s a
                                     -> Mapper s a
(|+|) = (<+>)
```

Plus two more convenience functions to use in the grammars.

```
anySymbol :: (Monoid m, Symbol m a) => [a] -> m a
anySymbol = anyof . map sym

some :: (Monoid m, Sequence m) => m a -> m [a]
some p = p <:> 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`.

```
instance Ord s => Functor (Mapper s) where
  fmap = FMap

unfold :: Ord s => (a -> b) -> Mapper s a -> Mapper s b
unfold f (as :&: pmap) = map (fmapQuit f) as :&:
                         mapMap (FMap f) pmap
unfold f (FMap g p)    = 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 `sym s` creates a trie with one edge `s`, at the end of which there is a single node with result `s`.

```
instance Ord s => Monoid (Mapper s) where
  zero                          = [] :&: emptyMap
  FMap f p      <+> q           = unfold f p <+> q
  p             <+> FMap f q    = p <+> unfold f q
  (as :&: pmap) <+> (bs :&: qmap) = (as ++ bs) :&:
                          mergeWith (<+>) pmap qmap

instance InputSymbol s => Symbol (Mapper s) s where
  sym s = [] :&: (s |-> return s)
  sat p = anyof (map sym (filter p 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.

```
instance Ord s => Sequence (Mapper s)

instance Ord s => Monad (Mapper s) where
  return a            = [returnQuit [] a] :&: emptyMap
  FMap f p      >>= k = unfold f p >>= k
  (as :&: pmap) >>= k = foldr (<+>)
                          ([] :&: mapMap (>>= k) pmap)
                          (map (k . 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.

```
instance Ord s => Parser (Mapper s) s where
  parse     p inp = parse'     p inp id
  parseFull p inp = parseFull' p inp id

parse' :: Ord s => Mapper s a -> [s] -> (a -> b)
                                     -> [([s], b)]
parse' (FMap f p)    inp     k = parse' p inp (k . f)
parse' ([] :&: pmap) []      k = []
parse' ([] :&: pmap) (s:inp) k = case pmap ? s of
                                   Just p  -> parse' p inp k
                                   Nothing -> []
parse' (xs :&: pmap) inp     k =
                 foldr ((:) . quit inp . fmapQuit k)
                       (parse' ([] :&: pmap) inp k) xs
```

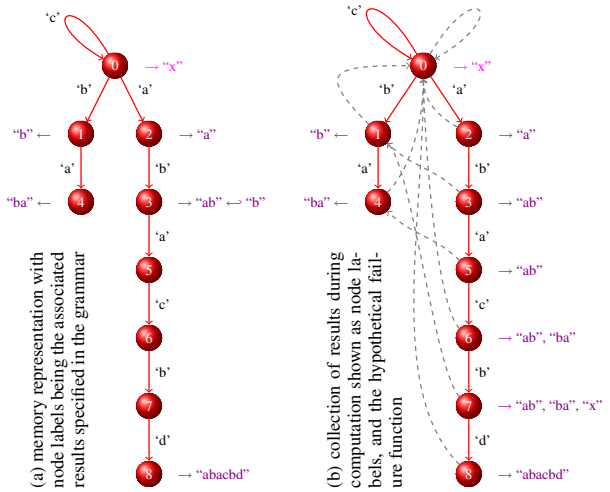

Figure 1. Trie structures illustrating the longest match algorithms.

```
parseFull' :: Ord s => Mapper s a -> [s] -> (a -> b)
                                         -> [b]
parseFull' (FMap f p)    inp k = parseFull' p inp (k . f)
parseFull' p@(xs :&: _) []  k = concat (map quitQuit xs)
      where quitQuit x = case quit [] (fmapQuit k x) of
                           ([], y) -> [y]
                           (is, y) -> y : parseFull' p is k
parseFull' (_ :&: pmap) (s:inp) k = case pmap ? s of
                                      Just p  -> parseFull' p inp k
                                      Nothing -> []
```

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.

3.1 Longest Match Insight

Consider the following rules defining our example trie in Figure 1a:

```
trie :: Mapper Char [Char]
trie = (many (syms "c") >> return "x")  -- loop over "c"
       |+| "b"          |.| "b"      -- equal to syms "b"
       |+| "a"          |.| "a"
       |+| "ab" |-| "b" |:| "ab"     -- cheating with "b"
       |+| "ba"         |.| "ba"
       |+| "abacbd"     |.| "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 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).

3.2 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.

```
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

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.

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

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 qf qc 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

          []  -> [ PW n h qf qc [] ]
          zs  -> [ PW n h qf qc ( concat
                       [ parsePW m z y | z <- zs ] ) ]

        xs      -> case l of

          0   -> [ PW n h qf qc
                       [ initPW m (f (justQuit x) :) |
                         x <- xs ] ]
          _   -> [ PW n h qf qc ( concat
                       [ parseWide m
                           [initPW m (f (justQuit x) :)]
                           (skipQuit x ++ [y]) |
                         x <- xs ] ) ]

      _          -> [ PW n h qf qc [] ]
```

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 `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 (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 `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 `lastPW s (initPW m id)` contain exactly the subtries to which the failure function is pointing transitively.

```
    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

        []      -> [ PW rn (h . rh) rf rc rs |
                     PW rn rh rf rc rs <- parseWide m
                                 (lastPW s (initPW m id))
                                 [y] ]
        xs      -> concat [ parseWide m
                     [initPW m (h . (f (justQuit x) :))]
                     (skipQuit x ++ [y]) |
                           x <- xs ]
```

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

```
stripFMap :: Ord s => (a -> c) -> Mapper s a
                                       -> Mapper s c
stripFMap k (FMap f p) = stripFMap (k . f) p
stripFMap k x          = FMap k x

lastPW :: Ord s => [ParseWide s a] -> ParseWide s a
                                   -> [ParseWide s a]
lastPW w p = case w of [] -> [p]; _ -> 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.

```
unParseWide :: Ord s => Mapper s a -> [ParseWide s a]
                                       -> [[[a]]]
unParseWide m = concat . map (unParsePW m)

unParsePW :: Ord s => Mapper s a -> ParseWide s a
                                       -> [[[a]]]
unParsePW m (PW l h f c s) = case c of

  FMap t q -> unParsePW m (PW l h qf qc s)
                where FMap qf qc = stripFMap (f . t) q

  r :&: k  -> case r of

    []      -> case s of
```

```
      []    -> [[ h [] ]]
      zs    -> [ h u : v | (u : v) <- unParseWide m zs ]

    xs      -> case l of

      0   -> [[ h [] ]]

      _   -> concat [ case skipQuit x of

               [] -> [[ 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 "abacbccc"   →  [[["ab","ba","x","b","x"]]]
ex "eabacbee"   →  [[["x","ab","ba","x","b","x","x"]]]
```

In case of `trie'` with 'problematic' rules and `ex'` defined likewise:

```
trie' = trie |+| "ab" |.| "ab"               -- ambiguity
             |+| ""   |-| "a"    |:| "y"     -- undefined
             |+| "c"  |-| "abac" |:| ""      -- expansion

ex' "abab" → [[["ab","ba","b"]],  -- cheating difference
              [["ab","ab"],["b"]],
              [["ab","ab"]]]
ex' "cc"   → [[["x"]]]            -- unique longer match
ex' "c"    → [[["x"]],
              [[""],["ab","ba","x"]],
              [[""],["ab","ba",""],["ab","ba","x"]], ...
                -- iterating ["ab","ba",""] infinitely
ex' "cbd"  → [[["x","b","x"]],
              [["x","b","y"]],    -- double match for 'd'
              [["","abacbd"]]]    -- finite 'c' rewriting
```

The expressive power of the rules and the robustness of the algorithm is pleasing. Obviously, 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 innermost 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]
```

3.3 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.

```
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 -> 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

      0   -> case r of

        []      -> [ PD l h f c i s ]
        xs      -> [ initPD m (h . (f (justQuit x) :)) |
                     x <- xs ]

      _   -> case r of

        []      -> [ PD rn (h . rh) rf rc ri rs |
                     PD rn rh rf rc ri rs <- parseDeep m
                                 (lastPD s (initPD m id))
                                 (reverse (y : i)) ]
        xs      -> concat [ parseDeep m
                     [initPD m (h . (f (justQuit x) :))]
                     (skipQuit x ++ [y]) |
                            x <- xs ]

lastPD w p = case w of [] -> [p]; _ -> w

unParseDeep m = concat . map (unParsePD m)

unParsePD m (PD l h f c i s) = case c of

  FMap t q -> unParsePD m (PD l h qf qc i s)
              where FMap qf qc = stripFMap (f . t) q

  r :&: k  -> case r of

    []      -> case s of

      []   -> [[ h [] ]]
      zs   -> [ h u : v | (u : v) <- unParseDeep m
                          (parseDeep m zs (reverse i)) ]

    xs      -> case l of

      0   -> [[ h [] ]]

      _   -> concat [ case skipQuit x of

               [] -> [[ h [f (justQuit x)] ]]
               is -> [ h [f (justQuit x)] : u |
                       u <- unParseDeep m (parseDeep m
                                    [initPD m id] is) ]

             | x <- xs ]
```

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:

```
parseLongest = parseLongestWide

parseLongestWith = parseLongestWideWith
```

4. 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`.

```
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.

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

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 ])
instance Sequence (Extend e s)

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

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 e s` 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.

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

instance ExtEnv e => Parser (Extend e s) s where
  parse     = parse'     initEnv
  parseFull = parseFull' initEnv

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]
                                              -> [a]
parseFull' e (Ext p) i = [ y | (([],_), y) <- p (i,[e]) ]
```

It is up to the user's definition of the environment type, what kind of information it will store. Thus, `Extend e s 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 e s` 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.

```
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`.

```
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`.

```
infixr 2 <|>

(<|>) :: Extend e s a -> Extend e s a -> Extend e s a
(<|>) p q = Ext (\ cs -> let Ext pp = p
                             r = pp cs
                             Ext qq = q
                         in case r of []      -> qq cs
 -- single solution only                      (s : _) -> [s]
                                              _       -> r)

again :: Extend e s a -> 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.

```
lookupList :: (OrdMap m, Ord s) => s -> [m s a] -> [a]
lookupList x l = concat [ maybe [] (: []) (i ? x) |
                          i <- l ]

oneof :: (Ord s, Symbol m s) => [Map s a] -> m s
oneof l = sat (\ s -> any (\ i -> maybe False
                                  (const True) (i ? s)) l)

lower :: (Ord s) => [s] -> [s] -> Extend e s [s]
lower s c = Ext (\ inp -> [ ((c ++ i, e), r) |
                            ((i, e), r) <- f inp ])
                where Ext f = syms s

upper :: (OrdMap m, Ord s) => [s] -> [m s [c]]
                                  -> Extend e d ([c] -> [c])
upper s l = foldM (\ f -> fmap ((.) f) . anyof .
                          map (return . (++)))
                  id [ lookupList x l | x <- s ]

upperWith :: (s -> m -> e d -> [[c]]) -> [s] -> m
                                  -> Extend e d ([c] -> [c])
upperWith f s l =
        do  e <- inspectEnv
            foldM (\ f -> fmap ((.) f) . anyof .
                          map (return . (++)))
                  id [ f x l e | x <- s ]
```

5. 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.

```
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.

```
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]
decoded = map toEnum $ []

    ++ [0x0640] ++ [0x0623, 0x0624, 0x0625]
    ++ [0x060C, 0x061B, 0x061F]
    ++ [0x0621, 0x0622] ++ [0x0626 .. 0x063A]
    ++ [0x0641 .. 0x064A]
    ++ [0x067E, 0x0686, 0x06A4, 0x06AF]
    ++ [0x0660 .. 0x0669]
    ++ [0x0671] ++ [0x0651]
    ++ [0x064B .. 0x0650] ++ [0x0670] ++ [0x0652]

encoded :: [Char]
encoded = map id $ []

    ++ "_" ++ "OWI"
    ++ ",;?"
    ++ "'|" ++ "}Abptvj Hxd*rzs$SDTZEg"
    ++ "fqklmnhwYy"
    ++ "PJVG"
    ++ ['0' .. '9']
    ++ "{" ++ "~"
    ++ "FNKaui" ++ "`" ++ "o"
```

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 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).

Script	Buckwalter	ArabTEX `[Char]`	Transcription	
بُلْبُلٌ	`bulobulN`	`"bulbuluN"`	*bulbulun*	(a)
بَلَابِلُ	`balaAbilu`	`"balAbilu"`	*balābilu*	(b)
قُرَّاءٌ	`qur~aA'N`	`"qurrA'uN"`	*qurrāʾun*	(c)
قِرَاءَةً	`qiraA'apF`	`"qirA'aTaN"`	*qirāʾatan*	(d)
شَيْءٍ	`$ayo'K`	`"^say'iN"`	*šayʾin*	(e)
شَيْئًا	`$y}A`	`"^s\"ay\"'\"aN"`	*šayʾan*	(f)
شَيْئَانِ	`$ayo}aAni`	`"^say'Ani"`	*šayʾāni*	(g)
سَيِّئُونَ	`say~i}uwna`	`"sayyi'Una"`	*sayyiʾūna*	(h)
سَيِّئُونَ	`say~i}uwna`	`"sayyi'uwna"`	*sayyiʾūna*	(i)
سَيِّئُونَ	`say~i}uwna`	`"sayyi'Un-a"`	*sayyiʾūn-a*	(j)
زِيَارَةُ ٱلزُّوَّارِ	`ziyaAr apu` `{lz~uw~aAri`	`"\\cap ziyAraT-u"` `"az-zUwAr-i"`	*Ziyārat-u* *'z-zūwār-i*	(k)
مِائَةٌ بِٱلْمِئَةِ	`miA}apN` `bi{lomi}api`	`"m_I'aHuN"` `"bi-al-mi'aHi"`	*miʾah* *bi-'l-miʾah*	(l)

Figure 2. Interpreting the `[Char]` input in the ArabTEX notation.

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

import Encode

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

instance Encoding Unicode
```

5.1 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 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 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 *ʾalif*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.

ArabTEX'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 ArabTEX notation requires some heuristics, if we want to achieve linguistically appropriate results. We leave these for future work.

5.2 Encode Arabic ArabTEX

This module relates the ArabTEX notation and the Arabic orthography. It provides definitions of the 'lexicon' of type `LowerUp`, which lists the distinct items in the notation and associates them with the information for their translation. Lexical items are identified in the lexing phase by an instance of Encode Mapper of type `Mapping`. The proper parsing phase uses Encode Extend parsers of type `Parsing`.

```
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 m _ v e) = Env q m w v e
setVerb  v (Env q m w _ e) = Env q m w v e
setEarly e (Env q m w v _) = 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 [
    ( "t",  [ 0x062A ] ),   ( "^s", [ 0x0634 ] ),
    ( "_t", [ 0x0638 ] ),   ( ".s", [ 0x0635 ] ),
    ( "d",  [ 0x062F ] ),   ( ".d", [ 0x0636 ] ),
    ( "_d", [ 0x0630 ] ),   ( ".t", [ 0x0637 ] ),
    ( "r",  [ 0x0631 ] ),   ( ".z", [ 0x0638 ] ),
    ( "z",  [ 0x0632 ] ),   ( "l",  [ 0x0644 ] ),
    ( "s",  [ 0x0633 ] ),   ( "n",  [ 0x0646 ] ) ]

invis = define [ ( "|",             [        ] ) ]

empty = define [ ( "",              [ 0x0627 ] ) ]

shadda = define [ ( "*",            [ 0x0651 ] ) ]

silent = define [
    ( "A",  [ 0x0627 ] ),   ( "W",  [ 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 ] ),   ( "q",  [ 0x0642 ] ),
    ( "^g", [ 0x062C ] ),   ( "k",  [ 0x0643 ] ),
    ( ".h", [ 0x062D ] ),   ( "m",  [ 0x0645 ] ),
    ( "_h", [ 0x062E ] ),   ( "h",  [ 0x0647 ] ),
    ( "`",  [ 0x0639 ] ),   ( "w",  [ 0x0648 ] ),
    ( ".g", [ 0x063A ] ),   ( "y",  [ 0x064A ] ),

    ( "B",  [ 0x0640 ] ),   ( "c",  [ 0x0681 ] ),
                            ( "^c", [ 0x0686 ] ),
    ( "p",  [ 0x067E ] ),   ( ",c", [ 0x0685 ] ),
    ( "v",  [ 0x06A4 ] ),   ( "^n", [ 0x06AD ] ),
    ( "g",  [ 0x06AF ] ),   ( "^l", [ 0x06B5 ] ),
    ( "^z", [ 0x0698 ] ),   ( ".r", [ 0x0695 ] ) ]

vowel = define [
    ( "a",  [ 0x064E ] ),   ( "_a", [ 0x0670 ] ),
    ( "i",  [ 0x0650 ] ),   ( "_i", [ 0x0656 ] ),
    ( "u",  [ 0x064F ] ),   ( "_u", [ 0x0657 ] ),
    ( "e",  [ 0x0650 ] ),   ( "o",  [ 0x064F ] ) ]

multi = define [ ( "A",     [ 0x064E, 0x0627 ] ),
                 ( "I",     [ 0x0650, 0x064A ] ),
                 ( "U",     [ 0x064F, 0x0648 ] ),
                 ( "Y",     [ 0x064E, 0x0649 ] ),
                 ( "E",     [ 0x0650, 0x064A ] ),
                 ( "O",     [ 0x064F, 0x0648 ] ),
                 ( "_I",    [ 0x0650, 0x0627 ] ),
                 ( "_U",    [ 0x064F, 0x0648 ] ),
                 ( "aNY",   [ 0x064B, 0x0649 ] ),
                 ( "aNA",   [ 0x064B, 0x0627 ] ) ]

nuuns = define [    ( "aN",         [ 0x064B ] ),
                    ( "iN",         [ 0x064D ] ),
                    ( "uN",         [ 0x064C ] ) ]

other = define [ ( "_aY",   [ 0x0670, 0x0649 ] ),
                 ( "_aU",   [ 0x0670, 0x0648 ] ),
                 ( "_aI",   [ 0x0670, 0x064A ] ),

          ( "^A",   [ 0x064F, 0x0627, 0x0653 ] ),
          ( "^I",   [ 0x0650, 0x064A, 0x0653 ] ),
          ( "^U",   [ 0x064F, 0x0648, 0x0653 ] ) ]
```

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, taaaa, silent,
              vowel, multi, nuuns, other, sukun, shadda,
              digit, punct, white ] )
    <+>  rules
    <+>  "" |.| error "Illegal symbol"

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

    |+|  ruleVerbalSilentAlif |+|  ruleInternalTaaaa
    |+|  ruleLiWithArticle    |+|  ruleDefArticle
    |+|  ruleIndefArticle
    |+|  ruleMultiVowel       |+|  ruleHyphenedVowel
    |+|  ruleWhitePlusControl |+|  ruleIgnoreCapControl
    |+|  ruleControlSequence  |+|  rulePunctuation
```

In the rules, for instance, care of the silent *ʾalif* after the ending `"aN"` is taken (Figure 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.

```
ruleIndefArticle =
    anyof [ c ++ m ++ "aNY"  |-| m ++ "aNY"  |:| [c]  |+|
            c ++ m ++ "aNA"  |-| m ++ "aNA"  |:| [c]  |+|
            c ++ m ++ "aN"   |-| m ++ "aNA"  |:| [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"    |-|    "ww"    |:|    ["u"]   |+|
    "aa"    |-|    "A"     |:|    []
    |+| anyof [
            "iy" ++ v  |-|  "y" ++ v      |:| ["i"]   |+|
            "uw" ++ v  |-|  "w" ++ v      |:| ["u"]
       | 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 `"\""`.

```
ruleLiWithArticle =
    anyof [ "l" ++ v ++ "-a" ++ c ++ "-" ++ c  |-|
            "l" ++ v ++          c ++ "-" ++ c  |:| []
    |  c <- elems [sunny, moony], c /= "l",
       v <- elems [vowel, sukun] ++ quote [vowel, sukun]]
    |+| anyof [
        "l" ++ v ++ "-a" ++ c ++ "-" ++ c      |-|
        "l" ++ v ++ "|-" ++ c ++ c             |:| [] |+|
        "l" ++ v ++ "-a" ++ c ++ "-" ++ c ++ c |-|
        "l" ++ v ++ "|-" ++ c ++ c             |:| [] |+|
        "l" ++ v ++ "-a" ++ c ++ "-"           |-|
        "l" ++ v ++         c ++ "-"           |:| [] |+|
        "l" ++ v ++ "-a" ++ c ++ c             |-|
        "l" ++ v ++ "|-" ++ c ++ c             |:| []
    |  c <- elems [sunny, moony], c == "l",
       v <- elems [vowel, sukun] ++ quote [vowel, sukun]]
```

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

With the `ruleLiWithArticle` rewriting, one need not distinguish these anymore, and can just join words, like in other cases:

لِلَّذَيْنِ لِلَّذِينَ `lil~a*ayoni lil~a*iyna`

`"li-lla_dayni li-lla_dIna"` *li-llaḍayni li-llaḍīna*

`"li-al-la_dayni li-alla_dIna"` *li-'l-laḍayni li-'llaḍīna*

بِٱللَّذَيْنِ بِٱلَّذِينَ `bi{ll~a*ayoni bi{l~a*iyna`

`"bi-al-la_dayni bi-alla_dIna"` *bi-'l-laḍayni bi-'llaḍīna*

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

```
decoderParsing :: Extend Env [Char] [UPoint]
decoderParsing = (fmap (foldr ($) []) . again) $
            parseHyphen          <|>     parseHamza
    <|>     parseDefArticle
    <|>     parseDoubleCons      <|>     parseSingleCons
    <|>     parseInitVowel
    <|>     parseWhite           <|>     parsePunct
    <|>     parseDigit
    <|>     parseQuote           <|>     parseControl

infixr 7 `plus`                          -- infixr 9 .
                                         -- infixr 5 ++
plus :: (a -> b) -> (c -> a) -> c -> b
plus = (.)
```

Unlike `decoderMapping`, ordering of rules in `decoderParsing` does matter. The `again` and `<|>` 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 `"l"` in the `sunny` set, and make it the output of the parser.

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

The compilation of 'syllables' in the Arabic script rests in putting vocalization marks (`vowel`, `sukun`, `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, taaaa, invis, silent]
      lower [c] []
      x <- upper [c] [consonant, taaaa, 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, sukun]
      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 `"\setverb"` mode, this carrier appears in the input after `"'`' `"` or `"'-'"` or `"'"`. In the complementary `"\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.

```
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
                  x : y   -> returnIList ((h ++ x) : y)
                  _       -> returnIList [h]
              oneof [bound]
```

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.

5.3 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.

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

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

data ZDMG = ZDMG | Trans deriving (Enum, Show)

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

Let us therefore only show how capitalization is implemented. The lexicon stores diacritized lowercase characters used as the standard phonetic transcription. Capitalization is possible (Figure 2, ex. k), but *hamza* `"'"` and *ʿayn* `"`"` are 'transparent' to it and let capitalize the following letter.

```
minor = define [
    ( "'",  [ 0x02BE ] ),   ( "`",  [ 0x02BF ] ) ]

sunny = define [ ( "t",     [ 0x0074 ] ),
                 ( "_t",    [ 0x0074, 0x0331 ] ),
                 ( "d",     [ 0x0064 ] ),
                 ( "_d",    [ 0x0064, 0x0331 ] ),
                 ( "r",     [ 0x0072 ] ),
                 ( "z",     [ 0x007A ] ),
                 ( "s",     [ 0x0073 ] ),
                 ( "^s",    [ 0x0073, 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
```

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 *tāʾ marbūṭah* ending occurs—either `"T"`, or silent `"H"`—while the script represents the ending always the same (Figure 2, ex. l).

```
ruleInternalTaaaa =
    anyof [ "H" ++ v        |-|  "H"             |:| []
          | v <- elems [vowel] ++ quote [vowel, sukun] ]
    |+| anyof [
            "T" ++ v ++ c   |-|  "t" ++ v ++ c   |:| [] |+|
            "H" ++ v ++ c   |-|  "t" ++ v ++ c   |:| []
    | c <- elems [sunny, moony, minor, invis],
      v <- elems [vowel, sukun] ++ quote [vowel, sukun] ]
```

6. Discussion

Next to the original ArabTEX parser (Lagally, 1992, 2004), there is an implementation in Perl of the Encode Mapper and Encode Arabic modules (Smrž, 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 (Smrž, 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.

7. 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.

```
parseArabHamza :: [Char] -> Extend Env [Char] [Char]
parseArabHamza h =
        do  e <- inspectEnv
            b <- prospectCarrier
            let carryHamza = case envEarly e of

                    []          -> case b of

                                     "ʼy"    ->  "ʼi"
                                     "ʼi"    ->  "ʼi"
                                     "ʼA"    ->  "ʼA"
                                     _       ->  "ʼa"

                    "i"  : _    ->              "ʼy"
                    "_i" : _    ->              "ʼy"
                    "e"  : _    ->              "ʼy"

                    "I"  : _    -> caseofMultiI b
                    "_I" : _    -> caseofMultiI b
                    "E"  : _    -> 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

                    ["", "ʼA"]  -> caseofMultiA b

                    ""   : _    -> case b of

                                     "ʼy"    ->  "ʼy"
                                     "ʼw"    ->  "ʼw"
                                     "ʼA"    ->  "ʼA"
                                     _       ->  "ʼ|"

                    _           -> error "Other context"

            case carryHamza of

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

            return carryHamza

      where caseofMultiI b = case b of
                                     "ʼi"    ->  "ʼ|"
                                     "ʼ|"    ->  "ʼ|"
                                     _       ->  "ʼy"

            caseofMultiU b = case b of
                                     "ʼi"    ->  "ʼ|"
                                     "ʼa"    ->  "ʼ|"
                                     "ʼ|"    ->  "ʼ|"
                                     "ʼy"    ->  "ʼy"
                                     _       ->  "ʼw"

            caseofMultiA b = case b of
                                     "ʼy"    ->  "ʼy"
                                     "ʼw"    ->  "ʼw"
                                     _       ->  "ʼ|"

            caseofVowelU b = case b of
                                     "ʼy"    ->  "ʼy"
                                     _       ->  "ʼw"

            caseofVowelA b = case b of
                                     "ʼy"    ->  "ʼy"
                                     "ʼw"    ->  "ʼw"
                                     "ʼi"    ->  "ʼi"
                                     "ʼA"    ->  "ʼA"
                                     _       ->  "ʼa"

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

lookaheadCarrier =

    do  v <- oneof [multi, other]
        let carryHamza = case v of

                    "I"     ->  "ʼy"
                    "_I"    ->  "ʼy"
                    "^I"    ->  "ʼy"
                    "E"     ->  "ʼy"

                    "U"     ->  "ʼw"
                    "_U"    ->  "ʼw"
                    "^U"    ->  "ʼw"
                    "O"     ->  "ʼw"

                    "A"     ->  "ʼA"

                    _       ->  "ʼa"

        lower [] [v]
        return carryHamza
    <|>

    do  v <- oneof [vowel, nuuns] <|> return ""
        c <- oneof [sunny, moony, taaaa,
                                  invis, silent]
        let carryHamza = case v of

                    "i"     ->  "ʼy"
                    "iN"    ->  "ʼy"
                    "_i"    ->  "ʼy"
                    "e"     ->  "ʼy"

                    "u"     ->  "ʼw"
                    "uN"    ->  "ʼw"
                    "_u"    ->  "ʼw"
                    "o"     ->  "ʼw"

                    "a"     ->  "ʼa"
                    "aN"    ->  "ʼa"
                    "_a"    ->  "ʼa"

                    _       ->  "ʼ|"

        case v of   ""      -> lower [] [c]
                    _       -> lower [] [v, c]

        return carryHamza
    <|>

    do  v <- oneof [vowel, nuuns] <|> return ""
        let carryHamza = case v of

                    "i"     ->  "ʼi"
                    "iN"    ->  "ʼi"
                    "_i"    ->  "ʼi"
                    "e"     ->  "ʼi"

                    _       ->  "ʼ|"

        case v of   ""      -> lower [] []
                    _       -> lower [] [v]

        return carryHamza
```

Acknowledgments

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

References

Aho, Alfred V. and Margaret J. Corasick. 1975. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM* 18(6):333–340.

Beesley, Kenneth R. 1997–1998. Romanization, Transcription and Transliteration. `http://www.xrce.xerox.com/competencies/content-analysis/arabic/info/romanization.html`.

Chakravarty, Manuel M. T. 1999. Lazy Lexing is Fast. In *FLOPS '99: Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, vol. 1722 of *Lecture Notes in Computer Science*, pages 68–84. London, UK: Springer-Verlag. ISBN 3-540-66677-X.

Forsberg, Markus and Aarne Ranta. 2004. Functional Morphology. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004*, pages 213–223. ACM Press.

Hutton, Graham and Erik Meijer. 1996. Monadic Parser Combinators. Tech. Rep. NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham.

Hutton, Graham and Erik Meijer. 1998. Monadic Parsing in Haskell. *Journal of Functional Programming* 8(4).

Kogai, Dan. 2002–2006. Encode. Character encodings module in Perl, `http://perldoc.perl.org/Encode.html`.

Lagally, Klaus. 1992. ArabTeX: Typesetting Arabic with Vowels and Ligatures. In *EuroTeX 92*, page 20. Prague, Czechoslovakia.

Lagally, Klaus. 1994. Using TeX as a Tool in the Production of a Multi-Lingual Dictionary. Tech. Rep. 1994/15, Fakultät Informatik, Universität Stuttgart.

Lagally, Klaus. 2004. ArabTeX: Typesetting Arabic and Hebrew, User Manual Version 4.00. Tech. Rep. 2004/03, Fakultät Informatik, Universität Stuttgart.

Leijen, Daan. 2001. Parsec, a Fast Combinator Parser. `http://www.cs.uu.nl/~daan/parsec.html`.

Leijen, Daan and Erik Meijer. 2001. Parsec: A Practical Parser Library. `http://research.microsoft.com/~emeijer/`.

Ljunglöf, Peter. 2002. *Pure Functional Parsing. An Advanced Tutorial*. Licenciate thesis, Göteborg University & Chalmers University of Technology.

Marlow, Simon. 2001. Happy: The Parser Generator for Haskell. `http://www.haskell.org/happy/`.

Marlow, Simon. 2003. Alex: A Lexical Analyser Generator for Haskell. `http://www.haskell.org/alex/`.

Smrž, Otakar. 2003–2006. Encode::Arabic. Programming module registered in the Comprehensive Perl Archive Network, `http://search.cpan.org/dist/Encode-Arabic/`.

Smrž, Otakar. 2006. *Functional Arabic Morphology. Formal System and Implementation*. Ph.D. thesis, Charles University in Prague. In preparation.

Swierstra, S. Doaitse. 2001. Combinator Parsers: From Toys to Tools. In G. Hutton, ed., *Electronic Notes in Theoretical Computer Science*, vol. 41. Elsevier Science Publishers.

Swierstra, S. Doaitse and Luc Duponcheel. 1996. Deterministic, Error-Correcting Combinator Parsers. In J. Launchbury, E. Meijer, and T. Sheard, eds., *Advanced Functional Programming*, vol. 1129 of *LNCS-Tutorial*, pages 184–207. Springer-Verlag.

Wadler, Philip. 1985. How to Replace Failure by a List of Successes. In *Proceedings of a Conference on Functional Programming Languages and Computer Architecture*, vol. 201 of *Lecture Notes in Computer Science*, pages 113–128. New York, NY, USA: Springer-Verlag. ISBN 3-387-15975-4.

Wadler, Philip. 1995. Monads for Functional Programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*, vol. 925 of *Lecture Notes in Computer Science*, pages 24–52. London, UK: Springer-Verlag. ISBN 3-540-59451-5.