Monadic Compositional Parsing with Context Using Maltese as a Case Study

Gordon J. Pace

Department of Computer Science and AI,
University of Malta

Abstract. Combinator-based parsing using functional programming provides an elegant, and compositional approach to parser design and development. By its very nature, sensitivity to context usually fails to be properly addressed in these approaches. We identify two techniques to deal with context compositionally, particularly suited for natural language parsing. As case studies, we present parsers for Maltese definite nouns and conjugated verbs of the first form.

1 Introduction

Combinator-based programming, where a basic set of objects are defined, together with operators to combine them to produce more complex instances has frequently been advocated. This approach essentially embeds a domain-specific language, which allows the used to use a combination of the embedded language and the host language to design a solution. The high-level of abstraction offered by modern functional programming languages such as Haskell and ML make them a perfect host language for such an approach. Various complex domains tackled in this manner can be be found in the literature (for example, hardware [2], stage lighting [**?**] and financial contracts [6]).

One domain in which the combinator approach has been successfully applied is that of parsing. In [15], Wadler presented a number of parser combinators for Haskell [9], using which one can compose complex parsers from simpler ones. Furthermore, the core-combinator code for parsing is itself very simple and easy to follow, making it possible to change and add new basic-parsing combinators as other parsing requirements not originally needed. Various later work (see, for example, [3–5, 7, 13, 14]) is based on this classic paper.

Ljunglöf [8] proposed the use of lazy functional programming languages (in particular, Haskell) to parse natural language texts. The paper uses the parser combinators from [15] and [5] to construct a Montague-style parser from English into logic statements. The parser is expressed very concisely in Haskell in easy-to-follow code, which makes the claim of the paper rather convincing. However, a number of more complex, yet interesting issues are not discussed. One such issue is that of sentence context-awareness, where certain words in a sentence depend on the context in which they appear in the sentence. A typical example is verb conjugation which must match its subject. Although these issues did not arise in the parser presented in [8], they arise immediately when the same approach is applied to a more comprehensive grammar, or to other languages such as Maltese.

Not much work has been done on the automatic parsing of Maltese. Of particular note, is the Maltilex project [11], which aimed at producing a computationally tractable lexicon of the Maltese language. Rosner [10] also treats the problem of parsing the Maltese article in conjuction with prepositions, using finite state automata, which is closely related to the domain of application we adopt for this paper.

In this paper, we give an overview monadic parser combinators (from [5]), and proceed to build the machinery to express grammars which are context dependent. We then propose solutions for problems arising with context-dependent grammars, in particular the case of natural language parsing. We use the parsing of simple Maltese grammatical constructs to illustrate our approach. At this stage, our primary aim has been the extension of parser combinators to deal with context. It remains to be established how useful our approach is on more substantial natural language grammar subsets.

2 Haskell Parser Combinators

Haskell is a strongly typed language, and types can be used, not just to identify certain problems at compile-time, but also to shape and direct a solution. It thus makes sense to start by asking what the type of a parser is to be. A parser in a function which reads an input stream of tokens of a particular type. After consuming part of the input, we would expect it to return back the object parsed, and the unconsumed part of the input stream. Actually, since the parser may match the input in more than one way, it would return a list of such possibilities:

```
newtype Parser a b = Parser ([a] -> [(b,[a])])
```

One may think that in some cases this extremely general parser type is overkill — producing the complete list of matches, when we may sometimes be only interested in a possible solution. However, lazyness comes to the rescue in this case. If we are only interested in any parse of the input, we just take the first element of the output list of the parser, and lazyness will take care of the rest.

Since the tag `Parser` stops us from applying the parser directly, we will define a function to 'peel' this tag away:

```
parse (Parser x) = x
```

We can now proceed to construct a number of simple parsers. The simplest parser is the one which returns a constant value, but leaves the input stream intact:

```
constantParser :: a -> Parser b a
constantParser x = Parser (\is -> [(x,is)])
```

Another simple parser is one which simply fails:

```
failParser :: Parser a b
failParser = Parser (\_ -> [])
```

Using these, we can define a parser which fails when a given boolean condition is false:

```
check :: Bool -> Parser a ()
check True  = constantParser ()
check False = failParser
```

Another useful parser, is one which checks that the input stream has been fully consumed:

```
fin :: Parser a ()
fin = Parser (\is -> case is of { [] -> [((),[])]; _ -> [] })
```

Parsing a single object at the beginning of the input:

```
one :: Parser a a
one = Parser (\is -> case is of { [] -> []; (x:xs) -> [(x,xs)] })
```

Parsers can be easily packaged as monads, which offer a reusable structure for functional programs.

```
instance Monad (Parser a) where
  -- return :: b -> Parser a b
  return = constantParser
  -- (>>=) :: Parser a b -> (b -> Parser a c) -> Parser a c
  parser1 >>= fparser2 =
    Parser $ \is ->
      let results1 = parse parser1 is
      in  concat [ parse (fparser x) is' | (x,is') <- results ]
```

The first offers a means of creating a parser which returns a given value, while the second allows us to run two parsers sequentially.

This now allows us to combine parsers to obtain compound parsers such as:

```
p1 <*> p2 =
  p1 >>= (\x1 -> p2 >>= \x2 -> return (x1, x2))
```

Haskell provides two alternative ways to make monad code look prettier:

```
p1 <*> p2 =
  do
    x1 <- p1
    x2 <- p2
    return (x1,x2)
```

```
p1 <*> p2 =
  do { x1 <- p1; x2 <- p2; return (p1,p2) }
```

Another useful parser is one which reads a symbol off the input stream, succeeding only if it satisfies a given property:

```
parseSat :: (a -> Bool) -> Parser a a
parseSat property = do { x <- one; check (property x); return x }
```

Parsing a particular character or string off the input stream can now be defined in terms of this combinator:

```
parseChar c  = parseSat (c==)

parseString "" = return ""
parseString (c:cs) = do { parseChar c; parseString cs; return (c:cs) }
```

Similarly, one can define parsing a vowel or consonant:

```
isVowel     c = c `elem` ['a','e','i','o','u']
isConsonant c = isAlpha c && not (isVowel c)

parseVowel     = parseSat isVowel
parseConsonant = parseSat isConsonant
```

Another useful combinator is the `ifnot` operator which tries one parser, and if it fails uses another:

```
p1 `ifnot` p2 = Parser $ \is ->
    case (parse p1 is) of
      [] -> parse p2 is
      rs -> rs
```

This last operator resolves potential non-determinism between the parsers by giving priority to the first operand. In fact, although our parser type can handle non-determinism, we have only, as yet, used them in deterministic ways. Non-deterministic choice between parsers can be easily defined in this framework:

```
p <+> q = Parser (\is -> parse p is ++ parse q is)
```

Parsing any word from a list of possibilities can now be expressed recursively as:

```
parseAnyOf []     = mzero
parseAnyOf (p:ps) = p <+> anyOneOf ps
```

Parsing a character or a word from a list of possibilities can now be expressed as:

```
anyCharOf   = parseAnyOf . map parseChar
anyStringOf = parseAnyOf . map parseString
```

One can now also define regular expression-like operators on parsers such as star (to accept any number of repetitions of a parser):

```
star :: Parser a b -> Parser a [b]
star parser =
 do { x <- parser; xs <- star parser; return (x:xs) } <+> return []
```

Parsing word space and space separated text has never been easier:

```
parseWordSpace = plus (parseSat isSpace)
  where
    plus p = p <*> star p

p1 <**> p2 = do { x1 <- p1; parseWordSpace; x2 <- p2; return (x1,x2) }
```

3 Parsing in Context: The Maltese Definite Article

We will now look at the Maltese article to see different approaches to parsing a natural language using the parser combinators presented in the previous section.

In English, parsing a definite noun using these combinators is trivial, since the article and noun are independent of each other.

```
parseArticle = parseString "the"
parseNoun = anyStringOf ["dog", "cat", "mouse", "computer"]
parseDefiniteNoun = parseArticle <**> parseNoun
```

The Maltese article is, however, more complicated.

3.1 The Maltese Article

The rules governing the Maltese article are quite intricate, combining morphological rules to aid pronunciation of the words with grammatical rules.

Indefinite nouns in Maltese have no extra signifier. The basic form of the definite article is constructed by adding *il–* before the noun. Hence, *kelb* (a dog), becomes *il–kelb* (the dog). Unfortunately, the matter is not so simple. Nouns starting with one of a number of letters (the *xemxin*[1] consonants: ċ, d, n, r, s, t, x, ż) have the initial letter of the noun assimilated into the article, replacing the *l* to accomodate pronunciation-based rules. *Serp* (a snake) becomes *is–serp* (the snake). Furthermore, nouns starting with a vowel sound

[1] Literally *sun letters.* Singular *xemxija.*

(possibly preceded by a silent letter) drop the initial *i* in the article. *Orfni* (an orphan) becomes *l–orfni* (the orphan). Finally, words starting with two or three consonants, the first being an *s* or an *x*, take *l–*, but also precede the noun by an initial *i*. For example, *spazju* (a space) becomes *l–ispazju* (the space).

To make matters worse, if the word preceding the article ends in a vowel, the article loses the initial *i* (should it start with one). For example, *kiel il–kelb* (he ate the dog), but *qela l–kelb* (he fried the dog)[2]

As these intricate rules indicate, parsing definite noun in Maltese is not as straightforward as it is in English. The logical flow of consequence flows from the right (the noun) to the left (the article). In a limited sense, this is also the case in English when parsing an indefinite noun, and finding *a* or *an*. However, whereas finding *an* in English, one can look for a noun starting with a vowel, in Maltese the situation is more complex since without matching against a lexicon, it is impossible to deduce whether *l–iskola* (the school) is the definite form of *skola* or *iskola*.

Furthermore, Maltese definite nouns reveal another important issue. Usually, we would like to have compositional parsers, that is, if I write a parser for the article, and a parser for nouns, I should be able to simply combine the two to parse a noun with an article. This is what was done in the case of the English definite noun. In the case of Maltese, this is not possible, since one would have to add constraints ensuring that the article and noun match. A simpler situation would arise when parsing the indefinite article (*a* or *an*) in English.

We propose a simple enrichment of the Haskell parser combinators enabling us to define such constructs compositionally, by refering to the context of a parsed object.

3.2 Compositional Context Parsing

A parser with look-ahead can differentiate between equivalent input by looking at input still to arrive. In the environment of an imperative language, one can use the state of the system to remember relevant information about the past context. This leads to a problem, since Haskell provides no global state. Although we can augment the Parser monad to encapsulate an updatable state (and we do so later to deal with another problem), it still lacks the symmetry of looking at context via state instead of using the parser combinators themselves.

3.2.1 Looking into the future With the current `Parser` structure, we can already handle looking ahead into the future input context. We can generalise the look-ahead parser into one which fails if the former failed, but succeeds returning no information (via the unit type `()`) and leaves the input untouched:

```
lookAhead :: Parser a b -> Parser a ()
lookAhead p = Parser $ \is ->
    case (parse p is) of
      [] -> return []
      _  -> return [((), is)]
```

This can be used to write a compositional parser for the indefinite English article, or the Maltese definite article:

```
articleEnglish = parseString "an" <*> lookAhead startsWithVowel
  where
    startsWithVowel = parseWordSpace <*> parseVowel

articleMalteseXemxin = do
```

[2] An alternative grammatical view is that the article consists of *l–*, which acquires an initial *i* if it is not preceded by a vowel, and the noun it is attached to starts with a consonant.

```
parseChar 'i'
c <- parseSat isXemxija
parseChar '-'
lookAhead (parseChar c)
return ('i':c:"-")
```

Note that the Maltese article parser works for nouns starting with one of the *xemxin* consonants ċ, d, n, r, s, t, x and ż. It can be extended to deal with the other cases in a similar manner.

Using this technique, we can describe future context in terms of the same combinators resulting in a composable parser. Parsing a definite Maltese noun can be expressed as: `articleMaltese <**> nounMaltese`, and the matching of the article with the noun is transparently taken care of.

3.2.2 Looking into the past Although the technique described in the previous section enables us to describe all the different cases of the Maltese definite noun, one nuance still has not been catered for — the initial *i* of the article is dropped if the preceding word ends in a vowel. Ideally, we would like to have a past context parser transformer to be able to specify the condition as:

```
weak_i = lookBack (parseVowel <*> parseWordSpace) `ifnot` parseChar 'i'
```

All occurrences of matching the initial *i* in Maltese definite article would then be replaced by the above parser, solving the problem. However, the parser type we have been using, does not hold any past information, and needs to be enriched to enable us to refer to past context:

```
type Stream a = ([a],[a])
newtype Parser a b = Parser (Stream a -> [(b,Stream a)])
```

We have basically just changed our notion of a stream from a sequence of unconsumed symbols to a pair of sequences of symbols — enabling us to recall not only the unconsumed symbols, but also the consumed ones. This does not change the use of any of the composed parsers. However, it may involve slight changes in the foundational parser combinators. For example, parser `one` would need to be redefined as follows[3]:

```
one :: Parser a a
one = Parser $ \(ws,is) ->
         case is of {[] -> []; (i':is') -> [(i',(ws++[i'],is'))] }
```

We can now define `lookBack` using this additional information:

```
lookBack :: Parser a b -> Parser a ()
lookBack p = Parser $ \(ws,is) ->
    case (parse (star one <*> p <*> fin) ([],ws)) of
      [] -> return []
      _  -> return [((), (ws,is))]
```

Note that, since we would like our parser to match the final part of the consumed input stream, we compose it after `star one` which consumes the first part of the stream. Clearly, this raises major efficiency issues.

However, our definition of `star` ensures that the order of the possible parses is longest first. Since most natural language look-backs would only look at a short part of the stream, and we are only interested whether there is at least one match, lazyness ensures that the match is quickly found, and the search stopped. Part of the overhead of using `star one` (the past stream is recomposed through the repeated calls to `one` by `star`) can be avoided by splitting the input and passing it directly to the parser, for some performance improvement.

[3] Interestingly, the definitions of the monad operators $\gg=$ and `return` remain unchanged, despite the fact that the variables now refer to different objects!

Another solution to improve efficiency can be used if the parser applied to the consumed input is sufficiently simple to be able to express its reverse of the parser (which matches the same strings but reversed). Is such cases, we can simply apply the reversed parser to the reversed past, with no other parsers composition. If the programmer would be required to provide the reversed parser, the solution would lead to unreadable and confusing code. One alternative would be to enrich the parser library to be able to identify a subset of parsers which can be automatically reversed, and apply the reverse of the parser before 'looking back'.

4 Further Context in Parsing: Maltese Verbs

The approach presented in the previous section allows us to express parsers with conditions on the context of the input stream where they are applied. The approach works well in the case of very local context, and is thus ideal to apply when one in word morphology based on phonological rules which depend on the immediate context. However, the approach is not so well suited for certain applications, where the context may be set by other complex rules based on text residing far from the location being parsed or information gathered as a result the parsing itself. Furthermore, any context parsing means duplicating the parsing effort since the information may be parsed multiple times, as a context and as the main text.

Another approach to enable adapting to the context, is to modify the global state in order to pass information which will be used later. The unstructured use of such an approach lead to non-reusable code. However, Haskell being a pure functional language with no side-effects, means that we have to use a monad (or equivalent) to pass the state around. This enables us to tailor the use of the state to a limited repertoire of operators, which ensure that the functionality is available without the possibility of abuse.

4.1 Maltese Verbs of the First Form in the Present Tense

The equivalent of the infinitive of Maltese verbs is the past third person singular conjugated form of the verb. Hence, the equivalent of *to ride* would be *rikeb* (he rode). Maltese verbs of semitic origin fall into one of ten forms (types). In the first and main form, one finds verbs based on three consonants (*trilitteri*) as in the case of *rikeb*. The other forms are derived from this first form. In this paper we will only be dealing with regular verbs of the first form.

The conjugation of regular verbs of the first form is rather straightforward, and follows the following pattern:

	Example (rikeb)	General case[4] ($c_1v_1c_2v_2c_3$)
First person singular	nirkeb	$n\ v_1\ c_1\ c_2\ v_2\ c_3$
Second person singular	tirkeb	$t\ v_1\ c_1\ c_2\ v_2\ c_3$
Third person singular male	jirkeb	$j\ v_1\ c_1\ c_2\ v_2\ c_3$
Third person singular female	tirkeb	$t\ v_1\ c_1\ c_2\ v_2\ c_3$
First person plural	nirkbu	$n\ v_1\ c_1\ c_2\ c_3\ u$
Second person plural	tirkbu	$t\ v_1\ c_1\ c_2\ c_3\ u$
Third person plural	jirkbu	$j\ v_1\ c_1\ c_2\ c_3\ u$

We make two remarks on the table above (i) the infinitive of the verb can be reconstructed from all the singular conjugations, but we lack the second vowel in the plural forms; and (ii) encountering *tirkeb* leaves an ambiguity whether we are referring to the second person singular, or the female form of the third person singular.

Based on these rules, one can easily check a given verb infinitive against a list of known verbs:

```
verbKnown  verb = verb `elem` ["rikeb", "kiteb" ...]
```

[4] The letters c_i indicate the consonants of the verb, while v_i indicate the vowels. Other symbols represent actual letters occurring in the verb.

4.2 Attributes

One recurring non-local dependency in natural language parsing, is that of matching gender, tense or case between different parts of the same sentence. To enable such matching, all we need is a means of setting attributes with particular values. An attribute may be set multiple times, but we would have to ensure that the value is always the same. In this manner, both parsing the noun and the verb may set the gender attribute of the subject of the sentence. However, unless the gender set by the two matches, the parsing of the phrase will fail. In the case of attributes which may take one of a number of possibilities, we use non-determinism to try all the cases. For example, when encountering the Maltese verb *tikteb*, we can deduce that the subject is either second person singular, or female third person singular. By having a non-deterministic parse, yielding either of the two, we ensure that any context deciding the gender or person of the subject to one of the possibilities will not fail, but simply reduce the non-determinism.

The other operations we would like to perform on attributes are reading and renaming. Reading is necessary to enable us to act according to a value set elsewhere. Renaming of an attribute may be necessary to be able to adapt the attributes as we acquire more information. For example, parsing a noun would set the attributes of the noun, while a simple sentence would set the attributes of the subject, verb and object. Therefore, once we discover that a particular noun phrase is the subject phrase, we would be able to rename the `NounGender` attribute to `SubjectGender`. We merge the state monad into the parser monad to handle attributes:

```
type State = [(AttributeName, AttributeValue)]
newtype Parser a b = (State, Stream a) -> [(b, (State, Stream a))]
```

As in the previous case, we would need to redefine some of the basic parser combinators, but the derived combinators would still work unmodified. Once again, the code defining the operators in the classes `Monad` remains unchanged.

We also define a number of functions which update and read the attribute list:

```
setAttribute :: (AttributeName, AttributeValue) -> Parser a ()
setAttribute attribute =
  Parser $ \(memory, is) ->
    if noClash attribute memory
      then [((), (updateState attribute memory, is))]
      else []

getAttribute :: AttributeName -> Parser a String
getAttribute attribute_name =
  Parser $ \(memory, is) ->
    if isDefined attribute_name memory
       then [(lookup attribute_name memory, (memory, is))]
       else []
```

Using similar definitions, we also define `setAttributes` (which updates a list of attributes), `renameAttribute` (which renames an attribute) and `renameAttributes`.

Before we define a parser for Maltese verbs, we must decide which attributes we will be setting. From the verb, we can only derive information about the subject. There are three pieces of information that can be gathered: the grammatical person, gender and number. We thus set three attributes, `SubjectPerson`, `SubjectGender` and `SubjectNumber` (respectively).

Parsing the first letter of the verb indicates the person. To simplify the presentation, we split the cases into two — singular and plural. The following in the code for the singular case. The plural case is similar.

```
parsePersonSingular =
  do setAttribute ("SubjectNumber", "Singular")
     c <- anyCharOf ['n','t','j']
```

```
    case c of
       'n' -> setAttribute   ("SubjectPerson" "First"
       'j' -> setAttributes [("SubjectPerson","Third"), ("SubjectGender","Male")]
       't' -> setAttribute   ("SubjectPerson","Second") <+>
              setAttributes [("SubjectPerson","Third"), ("SubjectGender","Female")])
```

Using these functions, we can now express how to parse a conjugated verb. As in the previous case, we separate parsing verbs conjugated in the singular and the plural form. The following is the singular form:

```
parseVerbSingular =
   do  parsePersonSingular
       v1 <- parseVowel
       c1 <- parseConsonant
       c2 <- parseConsonant
       v2 <- parseVowel
       c3 <- parseConsonant
       check (verbKnown [c1,v1,c2,v2,c3])
```

Parsing a verb is now simply the non-deterministic choice between the parsing a singular or plural verb:

```
parseVerb = parseVerbSingular <+> parseVerbPlural
```

Now that we have the parsing of a simple verb in place, and a parser for nouns (setting attributes `Gender`, `Person`, `Number`), we can construct a simple sentence parser for transitive verbs as follows:

```
parseSubject =
  do n <- parseNoun
     renameAttributes [("Noun"++x, "Subject"++x) | x <- ["Gender", "Person", "Number"]]
parseObject =
  do n <- parseNoun
     renameAttributes [("Noun"++x, "Object"++x) | x <- ["Gender", "Person", "Number"]]
parseSentence = parseSubject <**> parseVerb <**> parseObject
```

Although in the example given here we use strings both for the tag names and values stored within, it may be clearer to use a structured type allowing us to group together certain attributes (in this case, all three attributes refer to the subject — one would also need identical attributes for the object of the phrase). We use strings only for ease of presentation.

5 Conclusions

We have presented two approaches which can be used to introduce context-sensitivity for parsing natural language compositionally using parser combinators. The first allows us to refer to the context using parsers, while the second is an instantiation of the state monad with operators refined for natural language parsing.

The combinator-based approach to programming is extremely versatile and introduces a level of abstraction which enables program re-interpretation. Although in this paper we store each parser a function which actually performs the parsing (its behaviour), by changing the basic combinators we can store its structure (for example, the fact that the parser for the Maltese definite noun is built as the 'sequential' composition of two other parsers). No information is lost, since we can reconstruct the parsing function from the description. Having access to the structure, however, would enable us to use analyse and perform operations of the grammar which would have otherwise been impossible to perform[5]. The following are three avenues we would like to explore using this approach:

[5] This distinction between structure and behaviour is very important in other domains such as hardware description. If a combinator library for hardware description is only used to simulate the circuit, then just having a circuit described as a function from inputs to outputs will be sufficient. If, however, we would like to count the number of gates in the circuit, or measure the longest combinational path, clearly we require access to the underlying structure.

Reversing a parser: We have already discussed the efficiency issue when looking back at the consumed input stream. One possible improvement we mentioned in passing is by reversing a parser and applying it to the reversed past stream. Clearly, however, not all parsers can be reversed. With access to the structure, we can define a partial parser reverse function. The `lookBack` function can then use the reversed parser when possible, resulting in more efficient simple look-backs.

Optimisation: Other optimisations also become possible when we have access to the structure. An obvious example is analysing the given grammar structure, and construct a minimised automaton if it turns out to be a regular language.

Language generation: If we have access to the structure of the grammar, we can also use it to generate, or complete phrases, as opposed to parsing. We have already constructed an experimental phrase generator based on parsing combinators. Combining the two together would expose further the duality of grammars as means of checking sentence correctness and as tools to structure and generate sentences.

The work presented in this paper is mainly concerned with dealing compositionally with context, with particular emphasis on natural language processing. We feel that the examples presented in this paper are encouraging and plan to apply the techniques on a non-trivial portion of Maltese grammar to assess their real-world effectiveness.

References

1. L–Akkademja tal-Malti, *Tagħrif fuq il–Kitba tal–Malti II*, Klabb Kotba Maltin, 2004.
2. K. Claessen, M. Sheeran and S. Singh, *The Design and Verification of a Sorter Core,* in Correct Hardware Design and Verification Methods 2001, Lecture Notes in Computer Science 2144, 2001.
3. J. van Eijck, *Parser Combinators for Extraction*, 14th Amsterdam Colloquium, 2003.
4. G. Hutton and E. Meijer, *Monadic Parser Combinators*, J. of Functional Programming, 2(3):323–343, 1992.
5. G. Hutton, *Higher-Order Functions for Parsing*, J. of Functional Programming, 8(4):437–444, 1998.
6. S. Peyton Jones and J.-M. Eber and J. Seward, *Composing contracts: an adventure in financial engineering*, ACM SIG-PLAN Notices, 35(9):280–292, 2000.
7. Peter Ljunglöf, *Pure Functional Parsing*, Licentiate thesis, Technical Report no. 6L, Department of Computer Science and Engineering, Chalmers University of Technology, 2002.
8. Peter Ljunglöf, *Functional Programming and NLP*, Department of Computer Science and Engineering, Chalmers University of Technology, 2002.
9. S. Peyton Jones, J. Hughes et al., *Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language,*, Available from `http://www.haskell.org`, 1999.
10. M. Rosner, *Finite State Analysis of Prepositional Phrases in Maltese.* In G. Pace and J. Cordina, editors, University of Malta Computer Science Annual Workshop 2003 (CSAW '03), 2003.
11. M. Rosner, J. Caruana, and R. Fabri. *Maltilex: A Computational Lexicon for Maltese.* In M. Rosner, editor, Computational Approaches to Semitic Languages: Proceedings of the Workshop held at COLING-ACL98, Université de Montréal, Canada, pages 97–105, 1998.
12. M. Sperber, *Developing a stage lighting system from scratch,* in Proceedings of the sixth ACM SIGPLAN international conference on Functional programming, 2001.
13. D. Swierstra and L. Duponcheel, *Deterministic, error-correcting combinator parsers.* In Advanced Functional Programming, volume 1129 of LNCS, 1996.
14. S. Doaitse Swiestra, *Combinator Parsers: From Toys to Tools.* In Graham Hutton, editor, Haskell Workshop, pages 46–57, 2000.
15. P. Wadler, *How to Replace a Failure by a List of Successes.* In Jean-Pierre Jouannaud, editor, FPCA, volume 201, pages 113–128, 1985.