Implementation of the Arabic Numerals and their Syntax in GF

Ali Dada
SAP Research CEC
Blumenbergplatz 9
9000 St. Gallen, Switzerland
ali.dada@sap.com

Abstract

The numeral system of Arabic is rich in its morphosyntactic variety yet suffers from the lack of a good computational resource that describes it in a reusable way. This implies that applications that require the use of rules of the Arabic numeral system have to either reimplement them each time, which implies wasted resources, or use simplified, imprecise rules that result in low quality applications. A solution has been devised with Grammatical Framework (GF) to use language constructs and grammars as libraries that can be written once and reused in various applications. In this paper, we describe our implementation of the Arabic numeral system, as an example of a bigger implementation of a grammar library for Arabic. We show that users can reuse our system by accessing a simple language-independent API rule.

1 Introduction

1.1 Problem

Language technology and software localization consume a significant share of many companies' time and work. Translating an operating system or an application to different languages involves, in the traditional approach, translating out-of-context strings into different languages. This requires a language expert for each new language, and will still involve language-related problems because of the difficulty in translating out-of-context strings and taking care of morphological and syntactic variations at the same time. We illustrate this with an example. A mail reader application wants to display messages like

```
You have 1 new message
You have 2 new messages
You have 3 new messages
You have 100 new messages
```

If these are to be translated into Arabic, special morphological and syntactic considerations should be made, which include inflecting "message" in number:

`1 message`	رِسَالَةٌ	*risālatun*
`2 messages`	رِسَالَتَانِ	*risālatāni*
`(3-10) messages`	رَسَائِلَ	*rasāʾila*
`(11-99) messages`	رِسَالَةً	*risālatan*
`x100 messages`	رِسَالَةٍ	*risālatin*

So the word "messages" is translated into different words in Arabic, depending on the numeral counting it. Counted nouns are an extreme example of how varied case inflection can be: The case of the singular and the dual is determined by their syntactic function (nominative in the example above). This is not the case for plurals, which assume the genitive case from three to ten (رَسَائِلَ is diptote, thus the فَتحَة marker), then accusative (singular) from eleven to nighty-nine, and genitive again for plurals that are multiples of hundred. This is not to mention noun-adjective agreement which should be taken care of when translating "new messages" into Arabic.

The aforementioned details should not be the responsibility of the application programmer, and hav-

Proceedings of the 5th Workshop on Important Unresolved Matters, pages 9–16,
Prague, Czech Republic, June 2007.

ing translators do this work over and over again for each application can be costly and lead to repeated work and/or poor results.

1.2 Solution and Contributions

We reviewed in other works (Dada and Ranta, 2007) an approach that addresses problems in language technology similar but not limited to the above. We applied this approach to Arabic, thus developing a resource grammar for Arabic in which we implement rules that cover the orthography, morphology, and syntax. In short, this approach is based on developing libraries of natural language constructs and rules, which can be used by an application programmer who is not knowledgeable in a specific language. The core programming language is Grammatical Framework (GF) (Ranta, 2004). The language library, called a resource grammar (Khegai and Ranta, 2004) and comprising the linguistic rules, can be reused in applications through an *Application Programming Interface* (API) by programmers that are unaware of the details of the specific natural language. Such a programmer uses a resource grammar assuming it will take care of morphological and syntactic rules. So far, we have implemented significant parts of the Arabic morphology, syntax, orthographic rules, and provided a sample lexicon of 300 words based on the Swadesh list (Hymes, 1960).

In this paper, we only describe part of the work, namely the numeral system of Arabic and its syntax. In the next section we elaborate on the approach, the programming language that implements it, and on Resource Grammars.

2 GF and the Resource Library

GF is a special-purpose functional programming language for defining grammars of (formal or natural) languages. A common API and resource grammars for various natural languages accompany GF with a purpose similar to that of libraries in general programming languages: implementing pieces of code that can be reused by the application programmer.

GF makes a distinction between abstract and concrete syntaxes. The common API specifies a set of syntactic rules that are language independent (abstract syntax), and the resource grammar implements each rule according to the particular rules of the language (concrete syntax). This latter involves word order, agreement, case inflection, etc. This distinction can abstract over language-dependent features and enables an application programmer to write sentences in a language only by describing their abstract syntax trees or by translating them from another language, preferably in a limited domain. The abstract representation would then act as interlingua.

3 The Numerals

We give here an explanation of our treatment of the Arabic number system, namely the numerals and their counted nouns. Our implementation is based on the work done by Hammarström and Ranta (2004) in defining the cardinal numerals in GF. We will gradually give the governing grammar rules along with their our formal description in GF.

The numbers from one to nineteen in Arabic have two forms, a masculine form and a feminine one, so in general we will take gender to be one of the inflecting attributes of numbers. Which of these two forms to use depends on the counted noun and the counting number:

- The numerals 1 and 2 show gender agreement with the counted noun (their grammatical role is an adjective modifying this noun).
- Numerals 3-10 show gender polarity with the counted noun, so a masculine noun is counted with a number in its feminine form and vice versa, e.g. ثَلاثَة رِجَال (three [+FEM] men [+MASC]) but ثَلاث نِسَاء (three [+MASC] women [+FEM]).
- Numbers 11 and 12 have two constituents which show gender agreement with each other and with the counted noun, e.g. أَحد عَشَر رجلاً (eleven [+MASC] men [+MASC]).
- Numbers 13-19 show gender polarity between their first constituent and the counted noun.
- Numbers further on, except those ending in 01 and 02, show no gender distinction.

Numerals dictate the number of the counted noun is a way different to what is the case in other languages:

- Numeral One: The noun is in the singular form.

- Numeral Two: The noun is in the dual form.
- Numerals 3-10: The noun is in the plural form, e.g. ثلاثة رجال (three men [+PLUR]).
- Numerals $\geq$ 10: The noun is in singular form again, e.g. ثلاثون رجلاً (thirty men [+SING]).

The numbers inflect also in case, so in the general case the number can have different forms for the three cases: nominative, accusative, and genitive. But again, as with gender, this will depend on the particular range of numerals:

- Numeral 1: full case distinction (it is an adjective)
- Number 2: usually the noun in dual is used alone, and if the number 2 is specified then it is usually only for emphasis. In this case it's an adjective in the dual form, thus it has two cases: nominative and oblique, e.g. ولدان اثنان [+NOM] and ولدين اثنين [+OBL].
- Numerals 3-10 : full case distinction for the numbers; the counted noun is always genitive, e.g. خمسةُ كتبٍ (five [+NOM] books [+GEN]), خمسةً كتبٍ (five [+ACC] books [+GEN]), خمسةِ كتبٍ (five [+GEN] books [+GEN]).
- Numerals 11 and 13-19: only accusative, same as their counted noun, e.g. أربعةَ عشرَ قلمًا (fourteen [+ACC] pens [+ACC]).
- 12: same as 2, but the counted noun is always accusative
- The tens (20, 30, ... 90): nominative and oblique cases, the counted noun is accusative
- multiples of 100 or 1000: the counted noun is genitive.
- composites: the case distinction of the number is the same as each of its constituent parts, and the case of the counted noun is determined by the rule of the last part of the compound construction. For example, 23: the three follows the rule of 3-10, the 20 follows the rule of the tens, and the counted noun is accusative as in the rule of the tens, the last part of the construction twenty three (three and twenty in Arabic).

The rules above only treat the indefinite state of the numerals, since the numerals in the definite state will be an adjective modifying the noun. The case of such a noun will not then follow the rules above but will assume the case dictated by its syntactic role in the sentence. We do however give below the type of the numerals inflection table including all the attributes that a number can inflect in: gender, state, and case.

```
lincat Numeral = {
  s : Gender => State => Case => Str ;
  n : Size
} ;

param Size =
  One | Two | ThreeTen | Teen
| NonTeen | Hundreds | None ;

param
   Gender  = Masc | Fem ;
   State   = Def | Indef | Const ;
   Case    = Nom | Acc | Gen ;
```

The `lincat` (`lin`earize `cat`egory) statement defines the type of a numeral in Arabic. It states that in GF, an Arabic numeral is a record that comprises two fields. The first is a string `s` which is in this case an inflection table specifying that a numeral is inflected in gender, state, and case. The => operator is the table operator in GF, so having three inputs to the table means that a `Numeral` is inflected in these three attributes. The three inflectional attributes are defined as `param`eters that take one of predefined values: gender can be masculine or feminine, case can be nominative, accusative, or genitive, and state can be definite with *al*, definite with a genitive construction (إضافة) or indefinite. The second field is `n` of type `Size`, which is also defined as a `param`eter with several possible values. These values specify which range of numbers does the numeral belong to. This is needed to be able to apply the rules above properly at all stages, including the formation of the number and the formation of the noun phrase from the number and the counted noun.

As mentioned earlier, GF differentiates between abstract and concrete syntaxes, and this differentiation also applies for the numeral system. So first an abstract syntax defines how numbers are formed in a language-independent way. The numbers are defined in a way that draws similarities found across languages in the formation of compound numbers. We linearize the rules into Arabic thus making use of this division but making distinctions because of the special rules that govern numerals in Arabic. A typical example of such numbers is the special treatment that numbers ending in 2 have in Arabic due to the notion of the dual.

We give here the rules for the first division of numbers and show how we implement them for Arabic. The API specifies the following categories and rules for numbers less than ten:

```
cat
  Digit ;                          -- 2..9
  Sub10 ;                          -- 1..9

fun
  n2, n3, n4, n5, n6, n7, n8, n9 : Digit ;

  pot01 : Sub10 ;                  -- 1
  pot0 : Digit -> Sub10 ;          -- d * 1
```

So the number 1 is treated separately from the remaining digits. We want to preserve a difference in our Arabic implementation between `n2` and the remaining digits because of the different way the digit 2 combines in compound numbers later on. This is the motivation between the division seen in `Size` between `Two` and `ThreeTen`.

Following is the type of the categories above in Arabic (the concrete syntax):

```
lincat Digit = {
s : DForm => Gender => State => Case => Str;
n : Size
} ;

lincat Sub10 = {
s : DForm => Gender => State => Case => Str;
n : Size
} ;

param DForm = unit | ten ;
```

The inflection table shows what we discussed earlier, that Arabic numbers get in the general case inflected in gender, state, and case. The `DForm` is used to calculate both the digit and its multiple of ten.

We write functions that form the inflection tables of the digits: one function for numeral 2 (`num2`, not shown here) and one function for the rest of the digits, including 1 (`num1_10`, shown below). [1]

```
oper num1_10 : Str -> { s : DForm => Gender
  => State => Case => Str } = \xams ->
let xamsa = xams + "ap" in {
 s= table {
    unit => table {
     Masc => \\s,c => (sing xams) ! s ! c;
     Fem => \\s,c => Al ! s + xamsa
                      + dec1sg ! s ! c
    };
    ten => \\_,s,c => Al ! s + xams +
                      m_pl ! Indef ! c
  }
};
```

Note the following GF syntax notations: The keyword `oper` defines a GF function. An `oper` judgment includes the name of the defined operation (e.g. `num1_10` in the example above), its type (e.g. `Str -> { s : DForm => Gender => State => Case => Str }`), and an expression defining it (everything after the = operator). As for the syntax of the defining expression, notice the lambda abstraction form `\x -> t` of the function. Inflection tables are either specified by the `table` keyword or using the shorthand `\\... =>` notation. Finally, `+` is the character concatenation operator and `!` is the table selection operator.

The `num1_10` function takes a string which can be any of the stems of the numerals from one to ten excluding two, e.g. *ẖams*. From this stem, and using helping functions from the nominal morphology modules, we build the inflection table of the numeral. For example, for the case where `DForm` is `unit` and the `Gender` is feminine (e.g. *ẖamsah*), the actual numeral string would be the concatenation of a possible definite marker (*al*), the stem, and a suffix determined by the state and the case of the numeral, `s` and `c` respectively. The helping function that determines if the definite marker is needed is the following:

```
Al : State => Str =
  table {
    Def => "Al";
    _   => ""
  };
```

The second helping function defines the suffixes that attach to singular or broken plurals of the first (strong) declension of Arabic nominal words (Retsö, 1984). It calculates, given the state of the word and its case, what its suffix will be. Note that `N`, `F`, and `K` are the nominative, accusative, and genitive nunation diacritics.

```
dec1sg : State => Case => Str =
  table {
    Indef =>
      table {
        Nom => "N";
        Acc => "F";
```

[1] Our grammar files are in unicode, but the example codes shown here are written using the Buckwalter (2003) transliteration with a few changes that suit our needs. We note our use of 'c' to denote the *ʿayn*.

```
      Gen => "K"
    };
  _ =>
    table {
      Nom => "u";
      Acc => "a";
      Gen => "i"
    }
};
```

As expected, only words with indefinite state take double diacritics (nunation), where as the rest (*al*-definite or construct-definite words) take simple diacritics. The remaining helping functions will not be all explained here as they follow similar logic.

The `num1_10` and `num2` produce only the inflection tables (the `s` field of the digit record). We simply add the correct `Size` parameter to each digit as follows:

```
oper num3_10 : Str -> { s : DForm => Gender
 => State => Case => Str ; n : Size } =
   \xams ->
   num1_10 xams ** { n = ThreeTen } ;

lin n2 = num2 ** {n = Two };

lin n3 = num3_10   "valAv";
lin n4 = num3_10  ">arbac";
lin n5 = num3_10    "xams";
lin n6 = num3_10    "sit~";
lin n7 = num3_10    "sabc";
lin n8 = num3_10  "vamAnI";
lin n9 = num3_10    "tisc";

lin pot01 = num1_10 "wAHid" ** { n = One } ;

lin pot0 d = d ;
```

The last function in the linearization shown above, `pot0`, is used to promote a `Digit` into a `Sub10` in order to use it later on as any numeral less that ten. This is the way the API specifies different numerals, dividing them into categories based on the decimal system. We give here the rest of the API categories and their linearization in Arabic:

```
cat
  Sub100 ;       -- 1..99
  Sub1000 ;      -- 1..999
  Sub1000000 ;  -- 1..999999

lincat Sub100 = {
 s : Gender => State => Case => Str ;
 n : Size
} ;
```

We will now show only a few implementation examples of the rules that specify the formation of the `Sub100` category. The rest of the rules for this and other categories don't show any different logic and will not be detailed here. The first rule we give is for the special cases of numeral 11:

```
fun
  pot111 : Sub100 ;

lin pot111 = {
 s = \\g,d,_ =>
  case g of {
   Masc => Al ! d + ">aHada" ++ teen ! Masc;
   Fem => Al ! d + "<iHdaY" ++ teen ! Fem
  };
 n = NonTeen
};

oper teen : Gender => Str =
  table {
    Masc => "ca$ara";
    Fem  => "ca$rapa"
  };
```

The implementation shows how the qualitative rules stated at the beginning are described formally. The inflection table doesn't give different forms for the three cases, and the accusative is used whatever the context case is. Both parts of the construction show gender agreement.

The numbers 12-19 have a common rule in the API but we should differentiate in the Arabic linearization between 12 and 13-19 because of the special status of the dual in Arabic and the different rules that these numbers assume in Arabic (see rules above).

```
fun
  pot1to19 : Digit -> Sub100 ;  -- 10 + d

lin pot1to19 dig = {
 s = \\g,d,c =>
  case dig.n of {
   Two => Al ! d + num2.s ! unit ! g
            ! Const ! c ++ teen ! g  ;
   _   => dig.s ! unit ! g ! Const ! Acc
          ++ teen ! (genPolarity ! g)
  };
 n =
  case dig.n of {
   Two => NonTeen;
   _ => Teen
  }
 };

oper
  genPolarity : Gender => Gender =
    table {
      Masc => Fem;
      Fem => Masc
    };
```

The `pot1to19` function takes a `Digit` as argument. In our implementation we take cases for the

Size of the digit. When the Size is Two, i.e. the number will be 12, we apply the rules for number 12 as given in the beginning: gender agreement between the two constituents, the first constituent is inflected in case (it is basically number 2 in the Const state). Otherwise (when the digit size is ThreeTen), we apply the rules of numbers 13 - 19: gender polarity between the two constituents and the first constituent is the digit inflected for the construct state and accusative case. The second constituent for all the numbers 11-19 is always accusative as shown in the teen helping function before.

The rest of the rules for forming numbers will not be detailed here. Instead we will explain how all these numbers will combine with nouns to form noun phrases. The different number ranges as defined by the Size parameter will be now used extensively in applying the proper rules. Following is the rule that takes that takes a Determiner (which can, among others, be a numeral) and a common noun to give a noun phrase.

```
fun
    DetCN   : Det -> CN -> NP ;
```

The rule above has the same type in all languages since it's part of the language-independent API (abstract syntax). The advantage of this is that a user of our system can access the Arabic numerals at this high level of abstraction, without being knowledgeable about the details of our implementation.

When determiners combine with common nouns in the general case, it will make a difference whether or not the determiner was a numeral, and if it were then the range of the numeral will probably determine the case of the noun in the resulting NP. Thus the type of the determiner category should include a Size field which is taken directly from the size of the number if that determiner is a numeral:

```
lincat Det = {
  s : Species => Gender => Case => Str ;
  d : State;
  n : Size
} ;

param Species = NoHum | Hum ;
```

If the determiner is not a numeral, then this will be denoted by n = None.

The first determiner-noun modification we will introduce is the determiner's gender. If we don't consider numerals, then a determiner's gender is directly deduced from that of the noun. But, as we saw in the rules for Arabic counted nouns, if the numeral was in the range 3-10 or 13-19 (Size is ThreeTen or Teen), then the numeral will show gender polarity instead of agreement. The rest of the cases continue to show agreement. This is described in detGender:

```
oper
  detGender : Gender -> Size -> Gender =
    \g,s ->
    case s of {
      ThreeTen | Teen => genPolarity ! g;
      _ => g
    };
```

The arguments are the gender of the noun and the size of the determiner. The correct gender of the determiner is calculated after taking cases of the Size.

Again, if we were not to consider numerals, the number in which we should inflect the common noun (singular, dual, or plural) would be directly determined by the number of the determiner. Now with the consideration of numerals and their special rules that dictate the number of the counted noun, we have to specify a correcting function:

```
oper sizeToNumber : Size -> Number = \s ->
  case s of {
    ThreeTen | None => Pl;
    Two => Dl;
    _ => Sg
  } ;

param Number = Sg | Dl | Pl;
```

This function converts from the Size of the determiner to a number in which the noun should be inflected in. As the rules of Arabic numerals specify, only the 3-10 numeral range dictate a noun in the plural form. Apart from the dual, the remaining numeral ranges take a singular noun.

The last way that a numeral will affect the noun it counts is by specifying its case as we have already seen in the rules. Without considering numerals, the case of the noun would always be determined by its grammatical role in the sentence. Again, this changes with the introduction of numerals. We write now a function that takes the case from the sentence, along with the size and state of the determiner, and modifies the case if required:

```
oper
  nounCase : Case -> Size -> State -> Case =
    \c,size,s ->
```

```
    case <size,s> of {
      <Teen,_> => Acc;
      <NonTeen,_> => Acc;
      <ThreeTen,_> => Gen;
      <Hundreds,_> => Gen;
      <_,Const> => Gen;
      _      => c
    };
```

Numbers from 11 to 99 dictate the accusative case on the nouns they count, numbers from 3 to 10 and multiples of hundred dictate the genitive case of the nouns they count, and the remaining numbers (1 and 2) don't change the case determined by the context. The remaining case of `State = Const` takes care of the *iḍāfah* genitive constructions.

Thus, after applying all the "correction" functions above, we get the following implementation of the noun determination rule:

```
lin DetCN det cn =
let number = sizeToNumber det.n in {
 s = \\c =>
  det.s ! cn.h ! (detGender cn.g det.n) ! c
  ++ cn.s ! number ! (nounState det.d number)
                ! (nounCase c det.n det.d);
 a = agrP3 cn.h cn.g number
};

oper agrP3 : Species -> Gender -> Number
             -> PerGenNum=
  \h,g,n ->
    case <h,n> of {
      <NoHum,Pl> => Per3 Fem Sg;
      _          => Per3 g n
    };
```

The `agrP3` helping function tests for the case when the species and number are nonhuman and plural. This case is treated in agreement as the feminine singular.

4 Related Work

A large-scale implementation of the Arabic morphological system is the Xerox Arabic Morphological Analyzer and Generator (Beesley and Karttunen, 2000; Beesley, 2001). This system is developed using only the Xerox Finite State Technology tools (Beesley and Karttunen, 2003) from which an Arabic Finite State Lexical Transducer is written. A research version is available for online testing, and an expanded and updated version can be obtained with a commercial license. Another notable computational model of the Arabic morphology is Tim Buckwalter's Arabic Morphological Analyzer (Buckwalter, 2004b,a). Buckwalter's analyzer parses Arabic words and gives all their possible morphological interpretations, each solution having a unique lemma ID, different word constituents, the part-of-speech, and English glosses.

Other works that also use functional languages for the treatment of Arabic include a morphology system by Smrž (in prep.). This work is based on Functional Morphology (Forsberg and Ranta, 2004), a methodology for building morphological systems in the Haskell programming language. Our treatment of Arabic shares similarities with that of Functional Morphology. Both approaches use typed languages, making use of finite algebraic datatypes to define linguistic categories. Both languages are functional, so the approaches use functions to realize linguistic abstractions. A large-scale implementation of this approach, in which a typed functional programming language is used to build a morphology, is Huet's Sanskrit dictionary and morphological system (Huet, 2006) upon which the Zen computational linguistics toolkit is based (Huet, 2005).

Of the available works in Arabic syntax, we mention El-Shishiny (1990) who developed a formal description of Arabic syntax in Definite Clause Grammar. We also make note of the work in Othman et al. (2003), where the authors describe a parser they wrote in Prolog to parse and disambiguate the Arabic sentence. Shaalan (2005) builds on this work to develop a syntax-based grammar checker for Arabic called Arabic GramCheck.

5 Discussion

Our implementation of the Arabic numerals covers all natural numbers in the range 1-999,999. This was accomplished by implementing only a few functions, thanks to the repetitive way in which numerals are composed to form larger numerals. As for performance, Arabic grammars are slower to compile than comparable GF grammars of other languages, partly because of the additional complexity of Arabic and partly because of the general way in which our lexicon is specified. Our implementation stresses more on elegance and generality rather than efficiency, thus more work needs to be done on the latter.

6 Conclusion

We discussed in this paper the details of implementing the Arabic numeral system in GF. We motivated our work by taking an example that shows the value of having the necessary language rules implemented in a reusable fashion. We built up our implementation towards a single language-independent rule that a user can call to access our system. We show how the grammar formalism we use in our implementation parallels the way linguists think.

Acknowledgments

Most of the work was done at Chalmers University of Technology. Thanks to Prof. Aarne Ranta for supervising this work and providing constant help. Also thanks to Björn Bringert, Harald Hammarström, and Otakar Smrž for giving valuable comments.

References

Kenneth Beesley. Finite-State Morphological Analysis and Generation of Arabic at Xerox Research: Status and Plans in 2001. In *Workshop Proceedings on Arabic Language Processing: Status and Prospects*, pages 1–8, Toulouse, 2001. ACL.

Kenneth Beesley and Lauri Karttunen. Finite-state non-concatenative morphotactics. In *Proceedings of the Fifth Workshop of the ACL SIG in Computational Phonology*, pages 1–12, 2000.

Kenneth R. Beesley and Lauri Karttunen. *Finite State Morphology*. CSLI Studies in Computational Linguistics. CSLI Publications, Stanford, California, 2003.

Tim Buckwalter. Arabic transliteration, 2003. `http://www.qamus.org/transliteration.htm`.

Tim Buckwalter. Issues in Arabic Orthography and Morphology Analysis. In *Proceedings of the COLING 2004 Workshop on Computational Approaches to Arabic Script-based Languages*, pages 31–34, 2004a.

Tim Buckwalter. Buckwalter Arabic Morphological Analyzer Version 2.0. LDC catalog number LDC2004L02, ISBN 1-58563-324-0, 2004b.

Ali Dada and Aarne Ranta. Implementing an Open Source Arabic Resource Grammar in GF. In Mustafa Mughazy, editor, *Perspectives on Arabic Linguistics*, volume XX. John Benjamins, 2007.

Hisham El-Shishiny. A formal description of Arabic syntax in definite clause grammar. In *Proceedings of the 13th Conference on Computational Linguistics*, pages 345–347. ACL, 1990.

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

Harald Hammarström and Aarne Ranta. Cardinal Numerals Revisited in GF. In *Workshop on Numerals in the World's Languages*, Leipzig, Germany, 2004. Dept. of Linguistics Max Planck Institute for Evolutionary Anthropology.

Gérard Huet. A Functional Toolkit for Morphological and Phonological Processing, Application to a Sanskrit Tagger. *Journal of Functional Programming*, 15:573–614, 2005.

Gérard Huet. Sanskrit Site, 2006. `http://sanskrit.inria.fr/`.

D. H. Hymes. Lexicostatistics so far. *Current Anthropology*, 1:3–44, 1960.

Janna Khegai and Aarne Ranta. Building and Using a Russian Resource Grammar in GF. In *Intelligent Text Processing and Computational Linguistics (CICLing-2004)*, pages 38–41, Korea, 2004.

E. Othman, K. Shaalan, and A. Rafea. A Chart Parser for Analyzing Modern Standard Arabic Sentence. In *Proceedings of the MT Summit IX Workshop on Machine Translation for Semitic Languages*, pages 37–44, 2003.

Aarne Ranta. Grammatical Framework: A Type-theoretical Grammar Formalism. *Journal of Functional Programming*, 14:145–189, 2004.

Jan Retsö. State, Determination and Definiteness in Arabic: A Reconsideration. *Orientalia Suecana*, 33–35:341–346, 1984.

Khaled F. Shaalan. Arabic GramCheck: a grammar checker for Arabic: Research Articles. *Software - Pracice and Experience*, 35(7):643–665, 2005.

Otakar Smrž. *Functional Arabic Morphology. Formal System and Implementation*. PhD thesis, Charles University in Prague, in prep.