An Open Source Persian Computational Grammar

Shafqat Mumtaz Virk[a], Elnaz Abolahrar[b]
[a]Department of Applied IT,
University of Gothenburg, Sweden
[a]Rachna College of Eng. & Tech.,
University of Eng. & Tech. Lahore, Pakistan
[b]Department of Computer Science & Eng.
Chalmers University of Technology, Sweden
E-mail: virk@chalmers.se, elnaz.abolahrar @gmail.com

Abstract

In this paper, we describe a multilingual open-source computational grammar of Persian, developed in Grammatical Framework (GF) – A type-theoretical grammar formalism. We discuss in detail the structure of different syntactic (i.e. noun phrases, verb phrases, adjectival phrases, etc.) categories of Persian. First, we show how to structure and construct these categories individually. Then we describe how they are glued together to make well-formed sentences in Persian, while maintaining the grammatical features such as agreement, word order, etc. We also show how some of the distinctive features of Persian, such as the *ezafe* construction, are implemented in GF. In order to evaluate the grammar's correctness, and to demonstrate its usefulness, we have added support for Persian in a multilingual application grammar (the Tourist Phrasebook) using the reported resource grammar.

Keywords: Grammatical Framework, Abstract syntax, Concrete syntax.

1. Introduction

The idea of providing assistance to programmers in the form of software libraries is not new. It can be tracked back to 1959, when JOVIAL gave the concept of COMPOOL (Communication Pool). In this approach, the code and data that provide independent services are made available in the form of software libraries. Software libraries are now at the heart of modern software engineering, and many programming languages (e.g. C, C++, Java, Haskell, etc.) come with built-in libraries. However, the idea of providing natural language grammars as software libraries is relatively new. It was first introduced in CLE (Core Language Engine: Alshawi, 1992; Rayner, 2000). GF (Grammatical Framework: Ranta, 2004) is another example that provides natural language grammars in the form of libraries. GF is a special purpose programming language designed for developing natural language processing applications. Historically, GF and its libraries have been used to write a number of application grammars including GF-Key[1] (authoring and translation of software specifications), TALK[2] (a multilingual and multimodal spoken dialogue system), and WebALT[3] (multilingual generation of mathematical exercises). Moreover, GF has support for an increasing number of natural languages. Currently, it supports 23 languages (see the status of GF resource library[4] for more details).

GF provides libraries in the form of resource grammars – one of the two types of programs that can be written in GF. A resource grammar is a general-purpose grammar (Ranta, 2009a) that encodes the syntactic constructions of a natural language. For example modification of a noun by an adjective is a syntactic construction, and it is developed as a part of resource grammar development. A collection of such syntactic constructions is called a resource grammar. A resource grammar is supposed to be written by linguists, who have sufficient grammatical knowledge (i.e. knowledge about word order, agreement features, etc.) of the target natural language. The other type of grammar that one can write in GF is an application grammar. It is a domain specific grammar that encodes semantic constructions. This is supposed to be written by domain experts, who have a better understanding of the domain specific terms. An application grammar may use a resource grammar as a supporting library (Ranta, 2009b) through a common resource grammar API[5].

Furthermore, every grammar in GF has two levels: *abstract syntax* and *concrete syntax*, which are based on Haskell Curry's distinction of *tectogrammatical* and *phenogrammatical* structures (Curry, 1963). The abstract syntax is independent of any language and contains a list of categories (cat), and a set of tree-defining rules (fun). The concrete syntax contains rules telling how the abstract syntax categories and trees are linearized in a particular language. Since the abstract syntax is common to a set of languages – languages that are part of the GF resource library – it is possible to have multiple parallel concrete syntaxes for one abstract syntax. This makes the GF resource library multilingual. Development of a resource grammar means writing linearization rules (lincat and lin) of the abstract syntax trees for a given natural language. This is a challenging task, as it requires comprehensive knowledge of the target natural

[1]http://www.key-project.org/
[2]http://www.talk-project.org/
[3]http://webalt.math.helsinki.fi/content/index_eng.html
[4]http://www.grammaticalframework.org/lib/doc/status.html

[5] http://www.grammaticalframework.org/lib/doc/synopsis.html GF resource grammar API

language as well as a practical programming experience of GF. In this paper we describe the development of the Persian resource grammar.

Persian is an Iranian language within the Indo-Iranian branch of the Indo-European family of languages. It is widely spoken in Iran, Afghanistan, Tajikistan, and Uzbekistan. In Iran it is also called Farsi, and the total number of Farsi speakers is about 60 million (Bahrani, 2011). It has a suffix predominant morphology, though there are a small number of prefixes as well (Megerdoomian, 2000). Persian tense system is structured around tense, aspect and mood. Verbs agree with their subject in number and person, and there is no grammatical gender (Mahootiyan, 1997). Persian has a relatively free word order (Müller, 2010), but declarative sentences are mostly structured as "(S) (O) (PP) V". Optional subject (S) is followed by an optional object (O), which is followed by an optional propositional phrase (PP). All these optional components precede the verb (V).

In Sections 2 and 3, we talk about morphology and syntax (two necessary components of a grammar) followed by an example in Section 4. Coverage and evaluation is discussed in Section 5, while related and future work follows in Sections 6.

2. Morphology

Every GF resource grammar has a test lexicon of almost 450 words. These words belong to different lexical categories (both open and closed), and have been randomly selected for test purposes. Different inflectional forms of these words are built through special functions called lexical paradigms. These lexical paradigms take the canonical form of a word and build finite inflection tables. However, the morphological details are beyond the scope of this paper which concentrates on the syntactical details.

3. Syntax

While morphology is about principles and rules of making individual words, syntax is about how these words are grouped together to make well-formed sentences in a particular language. In this section, we talk about the syntax of Persian. First, in the following subsections we discuss different syntactic categories (i.e. noun phrases, verb phrases, adjectival phrases, etc.) individually. Then we show how they are glued together to make clauses and sentences in sections 3.5 and 3.6 respectively.

3.1 Noun Phrase

A noun phrase is a single word or a group of grammatically related words that function as a noun. It consists of a head noun, which is constructed at the morphological level, and one or more optional modifiers. In Persian modifiers mostly follow the noun they modify, even though in limited cases they can precede it. Below, we show the structure of a noun phrase (NP) in our implementation, followed by its construction.

Structure: A NP has the following structure:

```
cat NP ;
lincat NP:Type = {s        : NPForm=>Str;
                  a        : AgrPes ;
                  animacy : Animacy };
```

Where

```
param NPForm  = NPC Ezafe ;
param Ezafe   = bEzafe | aEzafe | enClic;
param AgrPes  = AgPes Number PPerson;
param Number  = Sg | Pl;
param PPerson = PPers1 | PPers2| PPers3;
param Animacy = Animate | Inanimate ;
```

This means that a NP is a record (indicated by curly brackets) of three fields. The purpose of different fields of a NP is explained below.

- 's' defined as 's:NPForm=>Str' is interpreted as: 's' is an object of the type 'NPForm=>Str', where the type 'NPForm=>Str' is a table type structure. In GF, we use such table type structures to formulate inflection tables. In brief 's' stores different forms of a noun phrase corresponding to the parameters 'bEzafe' (a form without the ezafe[6] suffix), 'aEzafe' (a form with the ezafe suffix) and 'enClic' (a form with the enclitic particle). For example consider the following table for the noun 'house'.

  s . NPC bEzafe => خانه -- Xɒ:næh
  s . NPC aEzafe => خانه ی -- Xɒ:næh i:
  s . NPC enClic =>خانه ای -- Xɒ:næh ɒ:i:
  a . AgPes Sg PPers3
  animacy . Animate

  These forms are then used in the construction of clauses and/or other categories. For example in Persian the 'aEzafe' form is used in modifications like adding an adjective e.g. " خانه ی بزرگ, Xɒ:næh bzrg, big house", and in showing possession e.g. " خانه ی من, Xɒ:næh mn, my house". The 'enClic' form is used in constructions where the noun is followed by a relative clause e.g. "خانه ای که آنجا است, Xɒ:næh kh A:njɒ: ɒ:st, the house which is there".
- 'a' is the agreement parameter and stores information about number and person of a noun phrase. This information is used for agreement with other categories.
- 'animacy' keeps the information about whether the noun phrase is animate or inanimate. This information is useful in the subject-verb agreement at clause level.

[6] Ezafe construction is a special grammatical feature of Persian, which is used to link the words in phrases (Samvelian, 2007). It is inherited from Arabic and is commonly used to express noun-adjective linking.

Construction: The head noun corresponds to the morphological category noun (N). The morphological category N is first converted to an intermediate category common noun (CN), through the following function:

```
fun UseN : N -> CN ;  --  خانه , Xɒ:næh, house
```

Where a common noun has the following structure:

```
lincat CN = {s : Ezafe=>Number=>Str ; animacy : Animacy};
```

It deals with modification of a noun by different modifiers including but not limited to adjectives, quantifiers, determiners, etc. We have different functions for these modifications. Consider the following function that is used for adjectival modification:

```
fun AdjCN   : AP -> CN  -> CN ;
--  خانه ی بزرگ , Xɒ:næhi: bzrg , big house
```

And its linearization rule for Persian is given below:

```
lin AdjCN ap cn = {
 s = table { bEzafe => table {
      Sg => cn.s ! aEzafe ! Sg ++ ap.s ! bEzafe;
      Pl => cn.s ! aEzafe ! Pl ++ ap.s ! bEzafe
                                  };
            aEzafe => table {
      Sg => cn.s ! aEzafe ! Sg ++ ap.s ! aEzafe;
      Pl => cn.s ! aEzafe ! Pl ++ ap.s ! aEzafe
                                  };
                  enClic => table {
      Sg => cn.s ! aEzafe ! Sg ++ap.s ! enClic;
      Pl => cn.s ! aEzafe ! Pl ++ ap.s ! enClic
                                    };
           };
animacy = cn.animacy
                      };
```

The above linearization rule takes an adjectival phrase and a common noun and builds a modified common noun. As explained previously ‘s’ in the above given code is an inflection table from ‘Ezafe to Number to String’, and stores different inflectional forms of a modified common noun. Since Persian adjectives do not inflect for number, we use the same form of an adjective, both for ‘Sg’ and ‘Pl’ parameters of the common noun. However, adjectives have three forms corresponding to ‘bEzafe’, ‘aEzafe’ and ‘enClic’ (see Section 3.3). As it is clear in the above code, whenever a common noun is modified by an adjective, the ‘aEzafe’ form of the common noun is used. Moreover, the modifier follows the common noun to ensure the proper word order.

GF provides a syntactic sugar for writing the above table concisely. For example the above given code can be replaced by the following simplified version.

```
lin AdjCN ap cn = {
   s      = \\ez,n =>  cn.s ! aEzafe ! n ++ ap.s ! ez;
   animacy = cn.animacy
    } ;
```

Note how the ‘\\’ operator is used as a syntactic sugar with parameter variables ‘ez’ and ‘n’ to compress the branches of a table together. Also note that ‘!’ is used as selection operator to select different values from the inflection table and ‘++’ is used as a concatenation operator.

The resulting common noun is then converted to a noun phrase (NP) through different functions depending on the constituents of the NP. In the simplest case a common noun without any article can be used as a mass noun phrase. It is constructed through the following function:

```
fun MassNP    : CN -> NP ; -- آب , A:b, water
```

And its linearization rule is:

```
lin MassNP cn = {s         = \\ez => cn.s ! ez ! Sg
                 a         = AgPes PPers3 Sg ;
                 animacy = cn.animacy
                } ;
```

This function takes a common noun and converts it to a NP.

Few others functions for the construction of a NP are:

```
fun DetCN  : Det -> CN -> NP ;
  -- مرد, mrd, man
fun AdvNP  : NP -> Adv -> NP ;
  -- امروز پاریس  ,pi:rs ɒ:mrɒ:z, Paris today
fun DetNP  : Det -> NP ;
  -- این پنج  , ɒ:n pnj, these five
```

3.2 Verb Phrase

A verb phrase normally consists of a verb and one or more optional complements. It is the most complicated category in our constructions. First, we explain the structure of the Persian verb phrase in detail, and then we continue with the description of its construction.

Structure: In our construction a verb phrase (VP) has the following structure:

```
cat VP ;
lincat VP : Type = {
  s          : VPHForm => {inf : Str} ;
  obj        :  Str ;
  comp       : AgrPes  => Str;
  vComp      : AgrPes  => Str;
  embComp  : Str ;
  inf        : Str;
  adv        : Str;
  } ;
```

Where

```
param VPHForm = VPTense Polarity VPPTense AgrPes
                | VPImp Polarity Number
                | VVForm AgrPes
                | VPStem1
```

```
                | VPStem2 ;
param VPPTense = VPPres Anteriority
                |VPPast Anteriority
                |VPFutr Anteriority
                |VPCond Anteriority ;
param Anteriority =  Simul | Anter ;
```

A brief explanation of different fields and their purpose is given below:

- As explained previously 's' is an inflection table, and here, it stores the actual verb form. We make different forms of a verb at verb phrase level. The parameter 'VPHForm' in the above code stores these different forms. A brief overview of these forms and their usage is given below:
  - 'VPTense' is a constructor with context parameters 'Polarity', 'VPPTense' and 'AgrPes'. It stores different forms of a verb inflecting for 'polarity', 'tense' and 'AgrPes' (where AgrPes = AgPes Number PPerson). These forms are used to make nominal declarative sentences at the clause level.
  - 'VPImp' stores the imperative form of a verb inflecting for polarity and number.
  - 'VVForm' stores the form of a verb, which is used when a verb takes the role of a complement of another verb (i.e. in the construction "want to run", 'to run' is used as a complement of the auxiliary verb 'want'. In English the infinitive of the second verb ('to run') is used as the complement of the auxiliary verb ('want'), but in Persian in most cases the present subjunctive form of the second verb is used as the complement of the auxiliary verb. We name this form the 'VVForm'. It inflects for number and person.
  - Finally 'VPStem1' and 'VPStem2' store the present and past roots of the verb, which have different forms in Persian.
- 'obj' is a string type field, which stores the direct object of a verb.
- 'comp' is an inflection table which stores the complements of a verb; those other than a direct object. The complement needs to be in agreement with the subject both in number and person. Therefore, we keep all the inflectional forms (for number and person) of a complement. This parameter is used to store indirect objects of di-transitive verbs.
- 'vComp' is another inflection table inflecting for number and person. When a verb is used as a complement of an auxiliary verb, we store it in this field. Unlike 'comp' or 'obj', this type of complement follows the auxiliary verb. For example in the sentence "او می خواهد بخوابد, ɒ:u: mi: xu:ɒ:hd bxu:ɒ:bd, she wants to sleep", the verb 'خوابیدن, xu:ɒ:bi:dn, to sleep' is the complement of the auxiliary verb 'خواستن, xvɒ:stn, want', therefore it will follow the auxiliary verb.
- 'embComp' is a simple string and is used when a declarative or interrogative sentence is used as a complement of a verb. For example in this sentence "او می گوید که من دارم می خوابم, ɒ:u: mi: gwi:d kh mn dɒ:rm mi: xu:ɒ:hm, she says that I am sleeping", the sentence "من دارم می خوابم , mn dɒ:rm mi: xu:ɒ:hm , I am sleeping" is the complement of the verb 'گفتن , gftn, to say'. This type of complement comes at the very end of a clause. The reason behind storing different types of complements in different fields is that in Persian these different complements take different positions within a clause (see section 3.4 for more details).
- 'inf' simply stores the infinitive form of the verb.
- 'adv' is a string field and stores an adverb.

Construction: The verb phrase (VP) is constructed from the morphological category verb (V) by providing its complements. In the simplest case a single verb without any complements can be used as a verb phrase. We create this verb phrase through the following function:

```
fun UseV     : V   -> VP ;
```
خوابیدن ، xu:ɒ:bi:dn, sleep

And its linearization rule is:

```
lin UseV v = predV v ;
```

Where

```
oper predV : Verb -> VPH = \verb -> {
 s = \\vh =>
   case vh of {
    VPTense pol (VPPres Simul) (AgPes n p) =>
      { inf = verb.s ! VF pol (PPresent PrImperf) p n };
    VPTense pol (VPPres Anter) (AgPes n p) =>
      { inf = verb.s ! VF pol (PPresent PrPerf) p n } ;
    VPTense pol (VPPast Simul) (AgPes n p) =>
      { inf =verb.s ! VF  pol (PPast PstAorist) p n } ;
      VPTense pol (VPPast Anter) (AgPes n p) =>
      { inf =verb.s ! VF pol (PPast PstPerf) p n } ;
    VPTense pol (VPFutr Simul) (AgPes n p) =>
      { inf =verb.s ! VF pol (PFut FtAorist) p n } ;
   VPTense pol (VPFutr Anter) (AgPes n p) =>
      { inf = verb.s ! VF pol (PPresent PrPerf) p n } ;
   VPTense pol (VPCond Simul) (AgPes n p) =>
      { inf = verb.s ! VF pol (PPast PstImperf)  p n } ;
   VPTense pol (VPCond Anter) (AgPes n p) =>
      { inf = verb.s ! VF pol (PPast PstImperf)  p n } ;
   VPImp pol n => { inf = verb.s ! Imp pol n}   ;
   VVForm (AgPes n p) =>
      {inf = verb.s ! Vvform (AgPes n p)} ;
   VPStem1 => { inf =  verb.s ! Root1};
   VPStem2 => { inf =  verb.s ! Root2}
```

```
        };
 obj = {s = [] ; a = defaultAgrPes} ;
 comp = \\_ => [];
 vComp = \\_ => [] ;
 embComp = [];
 inf = verb.s ! Inf;
 adv = [];
} ;
```

This operation (indicated by keyword 'oper' in the above code) converts a verb (a morphological category) to a verb phrase (a syntactic category). At the morphological level, Persian verbs inflect for tense (present/past/future), aspect (perfective/imperfective/aorist), polarity (positive/negative), person ($1^{st}/2^{nd}/3^{rd}$), and number (Sg/Pl). All these morphological forms are stored in an inflection table at the morphological level, and are used in this operation to make different forms at the verb phrase level. For example, the boldfaced line in the above code builds a part of the inflection table 's'. This part stores the forms of the verb that correspond to the (Present, Simul) combination of tense and anteriority, and all possible combinations of polarity and agreement (represented by variables 'pol' for polarity and 'AgPes n p' for agreement). All the complement fields of this verb phrase are left blank or initialized to default values. These complements are provided through other verb phrase construction functions including but not limited to the followings:

```
fun ComplVV  : VV  -> VP -> VP ;
  می خواهد بدود (ا و), (ɒ:u:) mi:  xu:ɒ:hd bdu:d, want to run
fun ComplVS  : VS  -> S  -> VP ;
می گوید او می دود (ا و) ,(ɒ:u:) mi:  gu:i:d ɒ:u:  mi:  du:d,
say that she runs
fun ComplVQ  : VQ  -> QS -> VP ;
  در تعجب است چه کسی می دود (ا و) ,(ɒ:u:) dr tʔjb ɒ:st tʃh
ksi: mi:dvd , wonder who runs
```

These functions enrich the verb phrase by providing complements. The resulting verb phrase is then used in making clauses, which is discussed in section 3.5.

3.3 Adjectival Phrase

In our construction an adjectival phrase has the following structure:

```
lincat AP =  {s : Ezafe => Str ; adv : Str} ;
```

Again 's' stores different forms corresponding to the parameters: 'bEzafe' (before Ezafe), 'aEzafe' (after Ezafe), and 'enClic' (Enclitic). 'adv' is a string field which stores the corresponding form, which is used when an adjective is used as an adverb.
Adjectival phrases are constructed from the morphological category adjective (A) through different construction functions. The simplest one is:

```
fun PositA  : A  -> AP ;       -- گرم , grm , warm
```

This function simply converts the morphological category adjective (A) to the syntactic category adjectival phrase (AP). Its linearization rule for Persian is very simple because an adjective and an adjectival phrase have the same structure. This is as simple as given below:

```
lin PositA a = a ;
```

It is possible to construct adjectival phrases from other categories. We have one function for each corresponding construction including the followings:

```
fun ComparA : A  -> NP -> AP ;
        گرم تر از من , grm tr ɒ:z mn , warmer than I
fun AdjOrd  : Ord -> AP ;
        گرم ترین  , grm tri:n, warmest
fun CAdvAP  : CAdv -> AP -> NP -> AP ;
         به جالبی جان , bh jɒ:lbi: jɒ:n, as cool as John
fun AdAP    : AdA -> AP -> AP ;
        خیلی گرم  , Xi:li: grm, very warm
```

3.4 Adverbs and other Closed Categories

Adverbs are made at morphological level, but it is also possible to construct them at syntactic level form other categories, for example from adjectives. We have separate construction functions for adverbs and other closed categories e.g. pronouns, quantifiers, etc. A few of them are listed here:

```
fun PositAdvAdj : A -> Adv ;
        به گرمی , bh grmi: , warmly
fun PossPron : Pron -> Quant ;
      من (خانه ی ) , (Xɒ:næh i:) mn,  my (house)
fun AdvIP    : IP -> Adv -> IP ;
        چه کسی در پاریس  , tʃh ksi: dr pɒ:ri:s,  who in Paris
```

3.5 Clauses

While a phrase is a single word or a group of grammatically related words, a clause is a single phrase or a group of phrases. Another difference is that a clause may have both a subject and a predicate of its own, while a phrase cannot have both at the same time. Though, sometimes it is possible that a clause does not have any subject at all, and is only composed of a verb phrase.

Structure: In our construction a clause has the following structure:

```
lincat Clause : Type = {s : VPHTense => Polarity =>
                                    Order => Str} ;
```

Where

```
Param VPHTense = VPres |VPas |VFut |VPerfPres
          |VPerfPast |VPerfFut|VCondSimul
          |VCondAnter ;
```

This shows that a clause is a record with only one field

labeled as ‘s’. It stores clauses with variable tense, polarity and order (declarative/interrogative), which are fixed at sentence level. The GF resource grammar API tense system covers only 8 possibilities through the combination of four tenses (present, past, future and conditional) and two anteriorities (anter/simul). The common API tense system is not adequate for Persian tense system - which is structured around tense, aspect, and mood. However, in our current implementation we stick to the common API tense system, and thus cover only eight possibilities. A better approach is to implement the full tense system of Persian and then map it to the common resource API tense system. This approach has been applied in the implementation of Urdu (Shafqat et. al 2010) and Punjabi (Shafqat et. al 2011) tense systems.

Construction: A clause is constructed through different clause construction functions depending on the constituents of the clause. The most important construction is from a noun phrase (NP) and a verb phrase (VP) through the following function:

```
fun PredVP    : NP -> VP -> Cl ;
      جان راه می رود  , jɒ:n rɒ:h mi:ɾu:d, John walks
```

And its linearization rule for Persian is:

```
lin PredVP np vp = mkClause np vp ;
```

Where

```
oper mkClause : NP -> VPH -> Clause = \np,vp -> {
       s = \\vt,pol,ord =>
       let
       subj = np.s ! NPC bEzafe;
       agr  = np.a ;
       vps  = case <pol,vt> of {
<Pos,VPres>  =>
            vp.s ! VPTense Pos (VPPres Simul) agr ;
<Neg,VPres>  =>
            vp.s ! VPTense Neg (VPPres Simul) agr ;
<Pos,VPerfPres>=>
            vp.s ! VPTense Pos (VPPres Anter) agr;
<Neg,VPerfPres> =>
            vp.s ! VPTense Neg (VPPres Anter) agr;
<Pos,VPast>  =>
            vp.s !  VPTense Pos (VPPast Simul) agr ;
<Neg,VPast>  =>
            vp.s !  VPTense Neg (VPPast Simul) agr ;
<Pos,VPerfPast>=>
            vp.s !  VPTense Pos (VPPast Anter) agr;
<Neg,VPerfPast>=>
            vp.s !VPTense Neg (VPPast Anter) agr;
<Pos,VFut>  => case vp.wish of {
        True  => vp.s ! VPTense Pos (VPPres Simul) agr ;
        False => vp.s ! VPTense Pos (VPFutr Simul) agr};
<Neg,VFut>  => case vp.wish of {
      True  => vp.s ! VPTense Neg (VPPres Simul) agr;
      False => vp.s ! VPTense Neg (VPFutrSimul) agr};
<Pos,VPerfFut> => case vp.wish of {
      True  => vp.s ! VPTense Pos (VPPres Anter) agr ;
      False => vp.s ! VPTense Pos (VPFutr Anter) agr};
<Neg,VPerfFut> => case vp.wish of {
      True => vp.s ! VPTense Neg (VPPres Anter) agr ;
      False => vp.s ! VPTense Neg (VPFutr Anter) agr};
<Pos,VCondSimul> =>
            vp.s ! VPTense Pos (VPCond Simul) agr;
<Neg,VCondSimul> =>
            vp.s ! VPTense Neg (VPCond Simul) agr;
<Pos,VCondAnter> =>
            vp.s ! VPTense Pos (VPCond Anter) agr;
<Neg,VCondAnter> =>
            vp.s ! VPTense Neg (VPCond Anter) agr };
quest = case ord of
                { ODir => [];  OQuest => "A:yA" };
in
quest ++ subj ++ vp.adv ++ vp.comp ! np.a ++
vp.obj.s ++ vps.inf ++ vp.vComp ! np.a ++
vp.embComp
      };
```

This operation takes a noun phrase (NP) and a verb phrase (VP) and constructs a clause with variable tense, polarity and order. Note how agreement information of the noun phrase (i.e. ‘np.a’ in the above code) is used to select the appropriate form of the verb phrase. This is done to ensure the subject-verb agreement. The ‘let’ statement stores different constituents of a verb phrase in different variables. Once we have all these constituents, they can be combined with the subject noun phrase in order to make a clause (see boldfaced code segment). Also note that in the declarative clauses the ‘bEzafe’ (before Ezafe) form of the subject noun phrase (i.e. ‘subj’ in the above code) is used. As an example, if the noun phrase (John) and the verb phrase (walk) were inputs to the above function, the output would be the following clause (only a portion of the full clause is shown):

```
s . VPres => Pos => ODir => جان راه می رود
       -- jɒ:n rɒ:h mi: ɾu:d , John walks
s . VPres => Pos => OQuest =>  آیا جان راه می رود
       -- A:i:ɒ: jɒ:n rɒ:h mi: ɾu:d, Does John walk?
s . VPres => Neg => ODir => جان راه نمی رود
      -- jɒ:n rɒ:h nmi: ɾu:d, John does not walk.
s . VPres => Neg => OQuest => آیا جان راه نمی رود
       -- A:i:ɒ: jɒ:n rɒ:h nmi: ɾu:d, Does John not walk?
s . VPast => Pos => ODir => جان راه رفت
       -- jɒ:n rɒ:h  rft,  John walked.
s . VPast => Pos => OQuest => آیاجان راه رفت
       -- A:i:ɒ: jɒ:n ,rɒ:h rft, Did John walk?
s . VPast => Neg => ODir => جان راه نرفت
       -- jɒ:n rɒ:h nrft, John did not walk.
s . VPast => Neg => OQuest =>     آیا جان راه نرفت
       -- A:i:ɒ: jɒ:n rɒ:h nrft, Did John not walk?
s . VFut => Pos => ODir => جان راه خواهد رفت
       -- jɒ:n rɒ:h Xu:ɒ:hd rft , John will walk.
s . VFut => Pos => OQuest =آیا جان راه خواهد رفت
      A:i:ɒ: jɒ:n rɒ:h Xu:ɒ:hd rft , Will John walk?
s . VFut => Neg => ODir => جان راه نخواهد رفت
       -- jɒ:n rɒ:h nXu:ɒ:hd rft , John will walk.
s . VFut => Neg => OQuest =>  آیا جان راه نخواهد رفت
      --A:i:ɒ: jɒ:n rɒ:h nXu:ɒ:hd rft,Will John not walk?
s . VPerfPres => Pos => ODir => جان راه رفت است
       -- jɒ:n rɒ:h rft ɒ:st , John has walked.
s . VPerfPres => Pos => OQuest => آیا جان راه رفت است
```

-- A:i:ɒ: jɒ:n rɒ:h rft ɒ:st , Has John walked?
s . VPerfPres => Neg => ODir => جان راه نرفت است
-- jɒ:n rɒ:h nrft ɒ:st , John has not walked.
s . VPerfPres => Neg => OQuest =>آیا جان راه نرفت است
-- A:i:ɒ: jɒ:n rɒ:h nrft ɒ:st , Has John walked?
s . VPerfPast => Pos => ODir =>جان راه رفت بود
-- jɒ:n rɒ:h rft bu:d , John had walked.
s . VPerfPast => Pos => OQuest =>آیا جان راه رفت بود
-- A:i:ɒ: jɒ:n rɒ:h rft bu:d , Had John walked?
s . VPerfPast => Neg => ODir => جان راه نرفت بود
-- jɒ:n rɒ:h nrft bu:d , John had not walked.
s . VPerfPast => Neg => OQuest => جان راه نرفت بود
-- A:i:ɒ: jɒ:n rɒ:h nrft bu:d , Had John not walked?
s . VPerfFut => Pos => ODir =>جان راه رفت است
-- jɒ:n rɒ:h rft ɒ:st , John will has walked.
s . VPerfFut => Pos => OQuest =>آیا جان راه رفت است
-- A:i:ɒ: jɒ:n rɒ:h rft ɒ:st ,Will John has walked?

This covers only one way of making clauses, there exist others as well, for example:

fun PredSCVP : SC -> VP -> Cl ;
این که او می رود خوب است , ɒ:i:n kh mi: ru:d xu:b ɒ:st,it is good that she goes.

3.6 Sentences

As mentioned and shown previously, a clause has variable tense, polarity, and order. Fixing these parameters results in declarative sentences. This is done through different functions, where the most important one is as follows:

fun UseCl : Temp -> Pol -> Cl -> S ;

Where

The parameter 'Temp' is a combination of two parameters: one for tense and the other for anteriority. Thus, the function 'UseCl' takes tense, anteriority, polarity and a clause as its input and produces a sentence as output. Therefore, if we fix the variable features of the example clause given in the 'Clause' section, we will get the following sentence - where tense is fixed to simple present, anteriority to simul, and polarity to positive.

s . جان راه می رود , jɒ:n rɒ:h mi: ru:d , John walks

This shows how we can make declarative sentences. Other types of sentences, i.e. interrogative sentences and relative sentences are built through the following functions respectively:

UseQCl : Temp -> Pol -> QCl -> QS ;
UseRCl : Temp -> Pol -> RCl -> RS ;

4. An Example

Here we give an example to demonstrate how our Persian resource grammar works at morphology and syntax levels. Consider the translation of the following sentence from English to Persian.
"He lives in my house"
Figure 1 (below) shows the automatically generated parse tree of the above sentence.

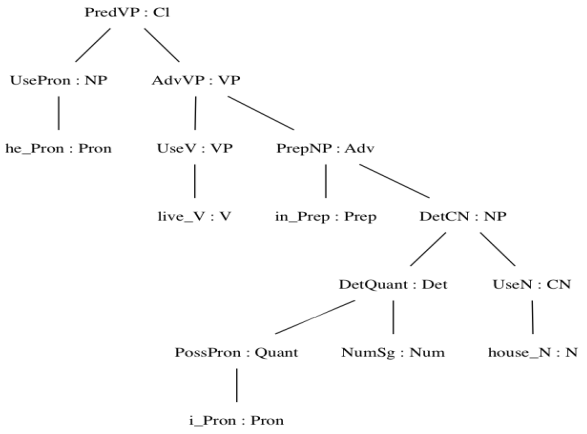

Figure 1: Parse Tree

At the lowest level we have the lexical entries. These lexical entries are used to construct different syntactic categories. These constructions are made according to the grammatical rules, which are declared at the abstract-level. For example the category noun phrase (NP) can be built from a Det (determiner) and a CN (common noun). In the abstract syntax we have the following rule for this construction:

fun DetCN : Det -> CN -> NP ;

Our goal, as a resource grammar developer, is to provide the correct linearization rule for this abstract tree-building function in Persian. This is achieved through implementation of the concrete syntax (described in the syntax section) for Persian. The morphological part ensures that the correct forms of words are created, while the syntactical part handles other grammatical features such as agreement, word order, etc.
The following diagram shows the automatically generated word alignments for the example sentence: "he lives in my house". The language pair is (English, Persian).

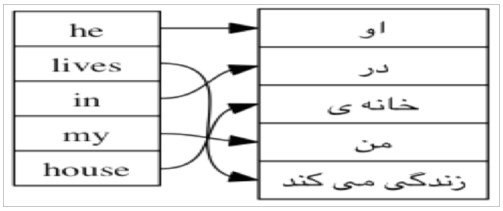

Figure 2: Word Alignments

5. Coverage and Evaluation

Our Persian resource grammar has 44 different categories and 190 syntax functions to cover different syntactic constructions. This covers a fair enough portion of the language but not everything. The reason for not being able to cover the whole language is the chosen approach of a common abstract syntax for a set of

languages in the resource grammar library. In principle, this approach makes it impossible to cover every aspect of every language. An example missing construction for Persian is the causative construction. Such missing constructions are supposed to be implemented in an extra language specific module, which is one direction for future work.

Testing a resource grammar is different from testing NLP applications in general, where testing is done against some text corpus. Testing resource grammars is much like testing software libraries (Ranta, 2009b). In this type of testing, a library is tested by developing some application grammars on top of the resource grammars. Phrasebook is a multilingual application grammar that was developed as part of the MOLTO-Project[7]. This application grammar has support for 15 languages. In order to evaluate our resource grammar we have added support for Persian to it. We achieved satisfactory results when a test case of 250 examples was generated. The application is open to test the accuracy and quality of translations, and is available on the MOLTO homepage.

Another possible way of testing is to generate a set of trees, linearize them, and observe their correctness. This approach has been applied to generate the synopsis[8] document, which contains a set of translated examples. The grammar was released when we reached a satisfactory performance level, with some known issues reported in the library documentation.

6. Related and Future Work

A Persian computational grammar was reported in (Bahrani 2011). This grammar is based on Generalized Phrase Structure Grammar (GPSG) model. Considering nouns, verbs, adjectives, etc. as basic structures, X-bar theory is used to define noun phrases, verb phrases, adjectival phrases, etc. This grammar is monolingual and can be used in applications, which need a syntactic analysis of the language. On the contrary, the grammar we developed is multilingual and can be used to develop different kinds of application grammars, ranging from text-translators to language generation applications, dialogue systems, etc.

(Müller, 2010) reported a Persian grammar implemented in TRALE system (Meurers, 2002). The grammar is based on the Head-driven Phrase Structure Grammar (HPSG) and is still under construction. Its coverage is limited due to the missing lexical items (i.e. verbs, numerals, clitic forms of a copula, etc.)

As mentioned above, the reported grammar does not cover all aspects of Persian. One direction for future work is to explore missing constructions and implement them in a separate language specific module.

Another possible direction for future work is the development of more application grammars on top of the reported resource grammar.

[7]MOLTO home page http://www.molto-project.eu/

[8]http://www.grammaticalframework.org/lib/doc/synopsis.html

7. References

Alshwai H., 1992. *The Core Language Engine. A set of parallel grammars written in Prolog and used as library.* MIT Press, Cambridge.

Bahrani M., Hossein Sameti, Mehdi Hafezi Manshadi, *A computational grammar for Persian based on GPSG*, Lang Resources & Evaluation DOI 10.1007/s10579-011-9144-1

Curry H.B., 1963. *Some logical aspects of grammatical structure*. In R. Jakobson (Dd.), structure of Language and its Mathmatical Aspects: Proceedings of the Twelfth Symposium in Applied Mathmatis, pp. 55-68. American Mathmatical Society, 1961.

Mahootiyan S., 1997, *Persian*. Routledge.

Megerdoomian K., 2000. *Persian computational morphology: A unification-based approach* Memoranda in Computer and Cognitive Science: MCCS-00-320. pp. 1 (http://www.zoorna.org/papers/MCCS320.pdf) (Last accessed February 2012).

Meurers W. D., G. Penn, and F. Richter, *A web-based instructional platform for constraint-based grammar formalisms and parsing,* in Proceedings of the Effective Tools and Methodologies for Teaching NLP and CL, 2002, pp. 18–25.

Müller S., Ghayoomi M., 2010. *PerGram: A TRALE Implementation of an HPSG Fragment of Persian,* Proceedings of the International Multiconference on Computer Science and Information Technology pp. 461–467 ISBN 978-83-60810-22-4,ISSN 1896-7094

Ranta A. 2004. Grammatical Framework: *A Type-Theoretical Grammar Formalism*. The Journal of Functional Programming 14(2) (2004) 145–189.

Ranta A., 2009a. LiLT Volume 2, Issue 2. *The GF Resource Grammar Library.* Copyright © 2009, CSLI Publications.

Ranta A., 2009b. *Grammars as Software Libraries. From Semantics to Computer Science*. Cambridge University Press, Cambridge, pp. 281-308.

Ranta A., 2011. *Grammatical Framework: Programming with Multilingual Grammars*, CSLI Publications, Stanford, 2011, 340 pp., ISBN-10: 1-57586-626-9 (Paper), 1-57586-627-7 (Cloth).

Rayner M. D., 2000. Carter P. Bouillon, Digalakis V., and Wiren M., *The spoken Language Translator*. Cambridge University Press.

Samvelian P., 2007. *A (phrasal) affix analysis of the Persian Ezafe,* Journal of Linguistics, vol. 43, pp. 605–645.

Shafqat M. Virk, M. Humayoun, A. Ranta, 2010. *An Open Source Urdu Resource Grammar.* Proceedings of the 8th Workshop on Asian Language Resources. In conjunction with Coling 2010.

Shafqat M. Virk, M. Humayoun, A. Ranta, 2011 *An Open Source Punjabi Resource Grammar*. Proceedings of Recent Advances in Natural Language Processing (RANLP), pages 70–76, Hissar, Bulgaria, 12-14 September 2011.