Automatic static class generation from informal BON class charts to formal textual BON notation.

MSc. Thesis Report

Aidan Morrissey

A thesis submitted in part fulfilment of the degree of MSc Advanced Software Engineering in Computer Science with the supervision of Dr.Joseph Kiniry.

School of Computer Science and Informatics

University College Dublin

11 May 2010

Abstract

This dissertation describes the process used to build the ESC/IBON (Extended Static Checker/Informal BON). The tool will be used in the translation of Informal BON (Business Object Notation) ,written in natural language to Formal BON notation.

The BON method is described focusing primarily on the BON static model.

Grammatical Framework; the tool used to complete the translation, is described in detail. The Gf files created as part of the project are included.

Finally a description of the Java application and API used to excess the Gf code is included.

Acknowledgments

The author gratefully acknowledges the invaluable help and support of the following people:

My dissertation supervisor, Dr Joe Kiniry, and his Phd student Fintan Fairmicheal, for their support and patience throughout the process.

Professor Aarne Ranta and Krasimir Angelov, for their speedy responses to questions posted on the Grammatical Framework Google Greoup.

My wife Emma, for taking on extra shifts looking after our two young daughters Morgan-May and Chloe and ensuring I had the time to complete this dissertation.

Table of Contents

1 Introduction

Design by contract is a cornerstone in the practise of Dependable or Hardcore Software Engineering. Dependable Software Engineering focuses on correctness, knowing what is going to be built and actually building it.

The term “Design by Contract” was first introduced by Bertrand Meyer in connection with his design of the Eiffel programming language and first described in various articles starting in 1986. Using this method formal reasoning and analysis are used to prove the objective of the project mathematically. Verification of the system can then be made post implementation.

Dependable Software Engineering practises include writing abstract specifications, realizing the classifiers of the system with classes, capturing system constraints with invariants, the use JML (Java Modelling Language) models and defining pre-conditions over post-conditions of the system. The steps mentioned are carried out before any line of code is written.

BON (Business Object Notation) is a high-level specification language used in the Dependable Software Engineering to capture the previously mentioned information. It is used to describe the system components and their relationships. BON creates the contracts from which Systems are built.

Tool support greatly enriches the Dependable Software Engineering process. The idea of writing large detailed specification turns most software engineers away from the process. XP (Extreme programming) fans would point to a system development process where the code acts as the specification. Tools can ease the burden of writing specifications.

The ESC/IBON (Extended Static Checker/Informal BON) , built as part of this project, tries to ease the “burden” of writing specifications while providing an extra layer of reversibility between the specifications of a system and its code. The tool was written primarily using Gf (Grammatical Framework); a type-theoretic grammar formalism based on Martin-Löf type theory and is used in the translation of natural language to BON notation. Before discussing the actual tool and Gf we must first expand on BON and what the ESC/IBON tool is actually translating.

1.1 BON (Business Object Language)

BON (Business Object Notation) was introduced circa 1994 by Kim Waldén and Jean-Marc Nerson. BON is a language used to describe primarily Object Orientated Systems but can be used to describe various other structured systems such as websites or databases. BON describes both the static (i.e. classes, relations) and dynamic (i.e. events) aspects of an object orientated system. BON is not a programming language but does have semantics and a concrete syntax. It attempts to capture the essence of O-O programming without specifically relating to one platform.

BON tries to be simple and well defined allowing users the ability to master the techniques as quickly as possible. It has been described as meaningful UML (Unified Modelling language), or UML without the complexity. *The language summary for UML (version 1.3) is 161 pages; where as the summary for BON is just a few pages.* [3]

BON does not use entity relationship diagrams and state charts used by languages such as UML. Its creators believed these types of modelling concepts to be incompatible with what is actually implemented in the system. They emphasised seamlessness, reversibility and software contracting as the key values of a software modelling language.

The BON method was developed to support the whole life cycle of a system, such that the same models and notations can be used to complete both forward and reverse engineering (seamlessness and reversibility). Methods within BON are contract based; they include assertions described in a mathematical fashion to define a class or relationship. BON ensures system changes are reflected in the design documentation. The design documents remain the main source of information throughout the life cycle of a system and not a description of what the system was meant to look at the start of development.

BON relies on the power of the pure object-oriented modelling concepts: a system as a set of classes (abstractions) which are related by inheritance and client dependencies. The O-O concepts are combined with a clustering mechanism to group related abstractions, and with precise component specification based on strong typing and software contracts. [2]

As previously mentioned the ESC/IBON tool will add to the BON tool suite adding a further layer of seamlessness to the process and taking reversibility a step further.

1.2 The BON Model

BON is broken down into two models, the static model and the dynamic model, both covering different elements of a system. The static model focuses primarily on the classes within a system, their interfaces, how they are related to each other and how they are grouped into clusters. The Dynamic model focuses on the incoming internal events, outgoing external events, and interesting system usage the system. The static model provides both the informal and formal class charts, the focus of the ESC/IBON tool.

1.3 The Static Model

The static model best describes what a system "is" as opposed to how a system "does what it needs to do". There are two parts to the Static model, the first is a set of untyped modelling charts, known as Informal BON and the second is a structured description containing fully typed class interfaces and a formal description of software contracts, this is known as Formal BON.

1.3.1 Informal BON

Three types of static charts are used within informal static model,

- System Chart: Listing the topmost clusters of the system
- Cluster Charts: Listing the classes and other clusters held within a cluster.
- Class Charts: Describing the interface of a class

SYSTEM	**CAR_RENTAL_SYSTEM**	**Part:1/1**
TYPE OF SYETEM System keeping track of vehicles and rental agreements in a car rental company.		**INDEXING** **keywords:** vehicle,rentel

Cluster	**Description**
Contract_Elements	Concepts that have to do with rental agreements such as contracts, clients, means of payment, rentals.
RENTAL_PROPERTIES	Properties of individual rentals such as vehicle, rate, extra options, insurance policy, certified drivers.
VEHICLE_PROPERTIES	Properties of rentable vehicles such as availability, location, models.

Figure 1. System Class Chart

CLUSTER	**ORGANIZATION**	**Part:1/1**
TYPE OF CLUSTER Handles all major events occurring during the organization and completion of a conference		**INDEXING** **keywords:** Organization,Staff

Class/(Cluster)	**Description**
CONFERENCE	The root class of the conference system.
PROGRAM	Information about the final conference program and its preparation..
(COMMITTEES)	The committees engaged in the conference organization to take care of the technical and administrative parts.

Figure 2. Cluster Class Chart

CLASS	**CITIZEN**	**Part:1/1**
TYPE OF OBJECT Person Born or living in a country	**INDEXING** **cluster:** CIVIL_STATUS	
Queries	What is your name? What is your Sex? What is your age? Do you have a Spouse? Who are your Parents? Impediment to marriage? Who are you children?	
Commands	Marry! Divorce!	
Constraints	Each Citizen has 2 parents . At Most 1 spouse allowed May not marry children or parents or persons of the samee sex. Spouse's spouse must be this person ~~All children, if any, must have this person among their parents~~	

Figure 3. Class Chart

All three charts are used at the very early stages of development and give a high level design of the system and its components. Examples of the charts are given in figures 1 - 3 .The header of all three charts are separated from its body with a double line The first row describes the chart type (System/Cluster/Class), the name of the chart and sequencing. The second row provides the class purpose which is a brief description of the class and indexing which includes information as the authors name and potential keywords used in the project. These keywords are decided on at the beginning of a project. The system chart (figure 1) lists the name and the description of each cluster held in the system. The cluster chart (figure 2) outlines the name of classes and clusters (cluster names are contained in parenthesis) and a description of each. Of particular interest to this paper is the class charts seen in figure 3. The main body of the class chart is broken into four sections

- Inherits From: List the classes which are direct ancestors to the class
- Queries: Describe the information other classes can ask of the class. Queries cannot change the state of the class; they provide information about the class .i.e. what is its colour.
- Commands: What services other commands can call on. Commands can change the state of the class i.e. Change Colour.
- Constraints: Constraints make the class "what it is" they are the rules which define the class i.e. Colour must be Blue or Red.

The queries, commands and constraints provide the text which is passed through the ESC/IBON tool to produce Formal typed BON.

1.3.2 Formal BON

This is the main part of the static model. The notation used in Formal BON has two variants textual and graphical. ESC/IBON focus on textual BON An example of a Formal BON class specification using textual Formal BON is given in figure 4. The example is the Formal representation of the informal class chart seen in figure 3. The notation used is not overly complicated and is easily understood by anyone familiar with O-O programming. The citizen class shown in figure 3 is described as deferred; it will not be implemented and has no interfaces. The class is exclusively used through inheritance. It is equivalent to an abstract class in Java or C#. Effective classes (which implement the interface of a deferred class) and interfaced classes (which interface with the outside world) are other possible class types. Formal BON charts are separated into features and invariants.

Features

Features are mapped to both the queries and commands in the Informal class charts. Features can be represented by the name of the feature followed by the TYPE returned, separated by a colon.

```
name : VALUE
```

The feature above could represent the query, what is your name? Features can be expanded to include argument type supplied in the query. The query, what is Johns name would have a signature

```
name : VALUE
       -> John : CITIZEN
```

Features do not solely represent fields, they may also represent a method or a method call. Parameters can be passed to the feature as part of a query or command. Pre-Conditions and Post-Conditions (described by the terms require and ensure) can also be associated with features. Figure 4 shows the feature *marry* requires, *sweetheart /= Void* and *can_marry* (*sweetheart*), meaning a sweetheart must exist and a separate feature can_marry (located within the class taking the parameter *sweetheart*) returns true. If these requirements are fulfilled the *marry* feature ensures, *spouse* = *sweetheart* or the citizens spouse is their sweetheart.

```
deferred class CITIZEN
    feature
            name, sex, age: VALUE
            spouse: CITIZEN –– Husband or wife
            children, parents: SET [CITIZEN] –– Close relatives, if any
            single: BOOLEAN –– Is this citizen single?
    ensure
            Result <–> spouse = Void
    end
    deferred marry –– Celebrate the wedding.
            –> sweetheart: CITIZEN
    require
            sweetheart /= Void and can_marry (sweetheart)
    ensure
            spouse = sweetheart
```

```
end
can_marry: BOOLEAN -- No legal hindrance?
        -> other: CITIZEN
require
        other /= Void
ensure
        Result -> (single and other.single
        and other not member_of children
        and other not member_of parents
        and sex /= other.sex)
end
divorce -- Admit mistake.
require
        not single
ensure
        single and (old spouse).single
end
invariant
        single or spouse.spouse = Current;
        parents.count = 2;
        for_all c member_of children it_holds
        (exists p member_of c.parents it_holds p = Current)
        end -- class CITIZEN
```

Figure 4. formal specifications using textual BON

Invariants

Invariants are mapped to the constraints described in the informal class charts. They must always be true both before and after any visible feature is called by a client. The invariants described Figure 4 are translated below,

```
single or spouse.spouse = Current
```

If you are a citizen then you must be single or else married to someone who is also married to you.

```
parents.count = 2;
```

A Citizen has two parents

```
for_all c member_of children it_holds
(exists p member_of c.parents it_holds p = Current)
```

For each child *c* which is a member of the list of children of this citizen,there is a parent *p* among the members of the list of parents to *c*, such that *p* is the current object (this citizen).

1.4 Tool Support

There are a number of BON development tools to aid in the creation of BON charts and the use of BON notation. These tools include tool suites and class skeletons which dynamically create BON notation from Javadoc and JML (Java Modeling Language). The two most recent tools developed are BONc and Beetlz.

BONc is an eclipse plugin which acts as a BON type checker for the user when creating the various charts described in the above sections. BONc provides the seamlessness environment where contract based charts are easily created.

Beetlz is a tool which can be used to create Java classes from formal BON charts. A large part of Beetlz's function is to provide reversibility from source code to BON formal charts.

Currently there is a gap in the BON tool set. The informal class charts are not incorporated into this seamless reversible process provided by the other tools discussed. Changes made to Formal BON charts or source code must be manually reflected in the Informal charts. Usually this would mean that the informal charts are not updated and become obsolete.

A downside of using BON is the need to have to understand the BON notation which, although quite small does prevent instant use of the method. The ESC/IBON   application will begin the process of addressing these issues. Although at first only focusing on class charts, it will help bring reversibility through to the first step in the design process, the informal class chart. ESC/IBON will help new users to BON understand the notation of the formal classes.

2 Gf (Grammatical Framework)

Gf is grammar formalism. It is a programming language used to build multilingual grammar applications and it can work with many languages, including formal languages, simultaneously. Gf was first created in 1998 at the Xerox Research Centre Europe, Grenoble, in the project Multilingual Document Authoring. At Xerox, it was used for prototypes including a restaurant phrase book, a database query system, a formalization of alarm system instructions with translations to five languages, and an authoring system for medical drug descriptions.

The most current release of the application (version 3.1.6) was released in April 2010. It has been used for building text translators, multilingual web gadgets, dialogue systems and many more language based systems. Gf is open source and available for all major operating systems and via compilation to JavaScript almost any platform which has a web browser.

Gf as a functional programming language like Haskell or ML and borrows much of its semantics and syntax from these languages. A Gf programme is known as a grammar. A simple grammar consists of two main parts, the abstract syntax, and the concrete syntax.

The abstract syntax describes the categories and functions of a grammar and how they are combined together to form abstract syntax trees. Abstract syntax is based on Martin-Löf type theory a logical system and a set theory based on the principles of mathematical constructivism. Thankfully Gf protects us from this and allows us to build programme based on hardcore mathematics without actually understanding the maths.

The concrete syntax of the grammar describes the linearization rules of the abstract syntax trees in a particular language. A grammar can have multiple concrete syntaxes all displaying the abstract trees in a different language.

Another important feature of the Gf application is the resource grammar library which provides an API to types and functions of common linguistic structures in various different languages (at the time of writing Gf provides resource grammars for 22 different languages)

2.1 Abstract Syntax

As mentioned above the abstract syntax resides at the core of the Gf application. An abstract tree has rule labels as its nodes and leaves. It ignores what order the rules come in and what words represent the rules just ensures that the rules are maintained. In keeping with the example of a citizen in the section on BON figure 5 displays a sample abstract syntax of a family.

```
abstract Family = {

      flags startcat = Phrase ;

      cat
      Phrase ; Citizenrole ; Citizen ; Role ;

      fun
      CitizenIsCitizenRoles : Citizen -> Citizenrole -> Phrase ;
      MakeCitizenRoles :  Role  -> Role -> Citizenrole ;
      Man,Women,Boy,Girl: Citizen ;
      Father,Mother,Brother,Sister,Son,Daughter:Role;

}
```

Figure 5. Family Abstract Syntax

The Family abstract syntax defines four categories Phrase, Citizen, Role and Citizenrole. The Categories represent the types of syntax trees. At this point no actual type (i.e. String or Noun) is associated with the categories. The abstract syntax describes the functions which make up the categories and how the categories interact with one another. Functions construct the abstract syntax trees. The first line in the abstract syntax "flags startcat = Phrase;" basically means that functions start with take it's position at the top of the syntax tree, figure 6 shows the layout of the abstract syntax tree

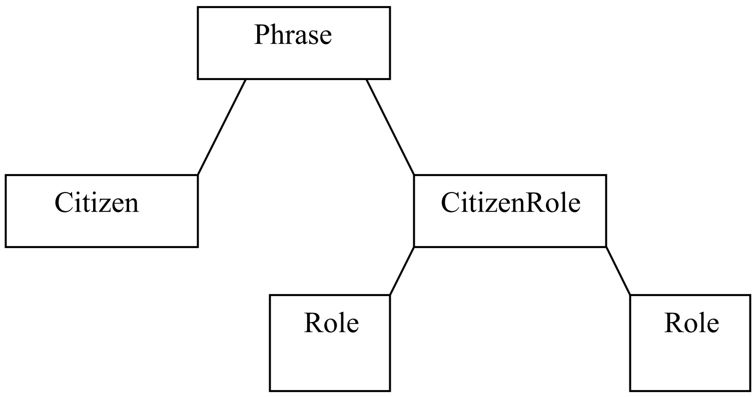


Figure 6. Family Abstract Syntax Tree

Phrase's are comprised of a Citizen and a Citizenrole as described by the function "CitizenIsCitizenRoles", a Citizenrole in turn can be made up of two Role's as described by the function "MakeCitizenRoles". Each phrase will consist of a citizen and the two roles associated with that citizen. The functions Man, Women, Boy and Girl produce the Citizen category and the functions Father, Mother, Brother, Sister, Son and Daughter produce the Role category. Below is an example of a Phrase created from this abstract syntax.

```
CitizenIsCitizenRoles Boy (MakeCitizenRoles Brother Son)
```

The concrete syntaxes can then linearize the syntax tree using the language associated with it.

2.2 Concrete Syntax

As mentioned earlier a number of concrete syntaxes representing different languages can be associated with an abstract syntax. Figures 7 and 8 display an English and Irish concrete syntax of the Family abstract syntax used in the previous example.

The first line of the concrete syntax displays the name and the name of the abstract syntax from which it is derived. In the case of the English concrete syntax we have "FamilyEng of Family". The next section provides us with the lincat judgments. This defines the linearization types of each category or the types of linguistic objects that each tree is converted to. In this case all trees are converted to strings.

The lin judgments define the linearization functions corresponding to each of the functions in the abstract syntax. In both the Irish and English linearization function of "CitizenIsCitizenRoles", a citizen and a citizenrole are the parameters of the function. The parameters are then combined to produce a phrase, represented by a string.

```
    concrete FamilyEng of Family = {

    lincat
          Phrase, Citizenrole,Citizen, Role = {s : Str} ;
    lin
    CitizenIsCitizenRoles citizen citizenrole = {s = "The" ++ citizen.s ++
"is a" ++ citizenrole.s};
    MakeCitizenRoles  role1 role2 = {s = role1.s ++ "and a" ++ role2.s} ;
    Man = {s = "man"};
    Women = {s = "women"} ;
    Boy= {s = "boy"} ;
    Girl = {s = "girl"} ;
    Father = {s = "father"} ;
    Mother= {s= "mother"};
    Brother={s= "brother"};
    Sister={s= "sister"};
    Daughther={s= "daughter"};
    Son={s= "son"};

    }
```

Figure 7. FamilyEng concrete syntax

```
concrete FamilyIrish of Family = {

    lincat
          Phrase, Citizenrole,Citizen, Role = {s : Str} ;

    lin
    CitizenIsCitizenRoles citizen citizenrole = {s = "Tá an" ++ citizen.s
++  "a" ++ citizenrole.s};
    MakeCitizenRoles  role1 role2 = {s = role1.s ++ "agus a" ++ role2.s} ;
    Man = {s = "fear"} ;
    Women = {s = "mná" } ;
    Boy= {s = "buachaill"} ;
    Girl = {s = "cailín"};
    Father = {s = "athair"} ;
    Mother={s= "mháthair"};
    Bother={s="deartháir"};
    Sister={s="deirfiúr"};
    Daughter={s="iníon"};
    Son={s ="mac"};
    }
```

Figure 8. FamilyIrish concrete syntax.

In the FamilyEng function"CitizenIsCitizenRoles", "The" is concatenated to the string representation of citizen (i.e. "man") which is concatenated to the string representation of Citizenrole. Citizenrole is created by the function "MakeCitizenRoles". The MakeCitizenRoles function concatenates the string "is a" to role1 (i.e. brother) and the string " and a" followed by roles2 (i.e father) . Putting the two functions together forms the phrase "The man is a brother and a father".

To perform a translation from one language to another, Gf uses parsing and linearization. Text is passed to a concrete syntax where it is parsed to produce an abstract syntax tree this is in turn is linearized through a separate concrete syntax providing the translation.

To complete a translation using the discussed example both concrete syntaxes are imported into Gf. The language which will be used to parse the phrase (translate from) and the language used to linearize the phrase (translate to) are specified in the command. A sample Gf command is detailed in figure 3.6.

```
p -lang=FamilyEng “The boy is a son and a brother” | l -lang=FamilyIrish
```

Figure 9. Sample Gf command

The first half of the command parses the phrase to the abstract syntax tree seen below, using the FamilyEng concrete syntax.

```
CitizenIsCitizenRoles Boy (MakeCitizenRoles Brother Son)
```

The second half laniaries the abstract tree to output “Tá an buachaill a deartháir agus a mac”, the FamilyIrish concrete syntax interpretation of the tree (probably a very bad Irish translation!). A number of other examples are displayed in figure 3.7.

English	Syntax Tree	Irish
The man is a father and a brother	CitizenIsCitizenRoles Man (MakeCitizenRoles Father Brother)	Tá an fear a athair agus a deartháir
The girl is a daughter and a sister	CitizenIsCitizenRoles Girl (MakeCitizenRoles Daughter Sister)	Tá an cailín a iníon agus a deirfiúr
The boy is a brother and a son	CitizenIsCitizenRoles Boy (MakeCitizenRoles Brother Son)	Tá an buachaill a deartháir agus a mac
The women is a mother and a sister	CitizenIsCitizenRoles Women (MakeCitizenRoles Mother Sister)	Tá an mná a mháthair agus a deirfiúr
The women is a father and a sister	CitizenIsCitizenRoles Women (MakeCitizenRoles Father Sister)	Tá an mná a athair agus a deirfiúr

Figure 10. Sample Family Translations

From figure 3.7 it is clear that the same syntax tree is common to both the Irish and English phrases. The abstract syntax is what all Gf grammars are based on. As previously mentioned if the rules of the abstract syntax are fulfilled it doesn't matter what the actual words used are. The last row in figure 3.7 shows an English phrase of “The women is a father and a sister” or “Tá an mná a athair agus a deirfiúr” in Irish, this obviously impossible but meets the criteria of the syntax tree and is parsed.

2.3 Gf Resource Library and API

The “family” grammar example described in the preceding sections did not incorporate any Gf resource grammar libraries, an important part of the Gf programme. In the example all categories where linearized to strings and not specific grammar types such as nouns or verbs. The purpose of the example was solely to portray the key concepts used in Gf and not use the specific tools or libraries used to construct complex linguistic syntax trees.

A typical Gf grammar such as the one written for the ESC/IBON tool describes a particular fragment of natural language. The syntax trees are purposely built to translate BON informal and formal charts, focusing on the type of sentences a user will enter when completing a BON chart. The resource grammar library provides an API for the construction of these sentences, removing the need for the Gf programmer to be a linguistic expert. Linguistic experts have created resource grammars for 22 different languages to date (unfortunately Irish is not one of them, the previous example is potentially the first Irish Gf grammar ever written!).

In order to access the English resource library, the concrete syntax of the application needs to open SyntaxEng and ParadigmsEng concrete syntaxes. The resource library of a language is broken down by categories. The lower level categories correspond to actual grammar types such as nouns, adjective, verbs, pronouns and adverbs. These lower level categories combine to produce Noun Phrases, Verb Phrase, Adjective Phrases which are combined to form Clauses or Question Clauses. This combination of categories eventually reaches to top level or top of the abstract syntax tree which is the category Text. A hierarchical view the Gf resource grammar categories are displayed in appendix 7.1.

The API of the resource grammar library displays how the categories combine to form other categories. An extract of the API is seen in figure 3.8. The full API is located on the Gf website[10] .

Function	Type	Example
mkNP	Numeral --> N --> NP	*Twenty men*
mkNP	Det --> N --> NP	*The first man*
mkVP	V --> VP	*walk*
mkVP	V2 ---> NP --->VP	*love her*
mkCl	NP-->VP --> Cl	*John walks here*
mkQCl	Cl --> QCl	*Does John walk?*
mkS	Cl-->S	*John walks*
mkPhr	S -->Phr	*John walked*
mkText	Phr--> Text	*But John walks.*
mkText	Text --> Text -> Text	*Where?When?Here? Now!*

Figure 11. Extract from Gf Resource Library API

The functions in figure 11 describe how various categories are combined to produce the specified category. The line describes a mkNP (make Noun Phrase) function takeing a Numeral and a Noun to produce a Noun Phrase. The job of the Gf programmer is to manipulate the API to produce the syntax trees required to parse and linearize the fragment of natural language on which the Gf grammar is focused.

3 ESC/IBON

The ESC/IBON tool comprises of two parts a Gf grammar and a Java API all of which is packaged neatly in a JAR file. The Gf grammar consists of a number of Gf files, primarily a BON abstract syntax and a concrete syntax for both English or Informal BON and Formal BON.

The Java API allows the user pass text as a string to a Gf process running the Gf grammar. The Gf grammar will translate the text string, to or from Formal BON and will return the translated text to the user.

3.1 Gf Grammar

The Gf grammar consists of seven files. Each of the files is interlinked, calling the English resource library. Each file is discussed separately in the preceding sections.

3.1.1 BONAbs.gf

BONAbs is the main abstract syntax of the Gf grammar. The categories and functions used in the file combine to construct syntax trees, specifically focusing on the type of natural language used in informal BON charts. The construction of the syntax trees was completed using a large cross section of existing BON charts. Similarities in how queries commands and constraints where phrased were used as the basis of the function.

The categories and functions where created with the resource grammar in mind, categories needed to be combined in a way that allowed the concrete syntaxes linearize the grammar types.

BONAbs extends three other abstract syntax DictEngAbs, BONTermsAbs, Numeral. When an abstract syntax extends another it inherits all the categories and functions within the extended syntax and syntax extended from the extended syntax. The abstract syntax becomes a combination of itself plus all the extended files.

The Numeral abstract syntax is extended to allow the grammar access to numeral categoriess within the resource library

The DictEngAbs abstract syntax extends the Cat abstract syntax located in within the Gf resource library. Cat contains the base categories of the resource library. BONAbs can use the base categories within its functions through this extension. DictEngAbs and BONTermsAbs will be explained further in later sections.

Figure 12 shows the extract from the opening lines of the BONAbs. Appendix 7.2 shows the full BONAbs file

```
abstract BONAbs = DictEngAbs,BONTermsAbs,Numeral ** {

       flags startcat=Output;

       cat
           Output;
           Phrase;
           Sentence;
           SentenceList;
           Interrogative ;
           Noun;
           Adjective;
           Verb;
```

Figure 12. BONAbs header

The first line of Figure 12 shows how the file extends other files. The startcat or head of the abstract syntax is the category Output. An incomplete list of categories follows the startcat expression. A number of categories take the name of common grammar categories however this does not mean that they are these categories as will be proven when discussing the various concrete syntaxes. Figure 13 displays various functions taken from the file. Each of the functions combine to form the Output category or combine to form categories used to produce Output.

```
--Overall
MakeText : Phrase -> Punctuation -> Output;
MakeTextSentence : Sentence -> Output;
MakeSentenceConj : Conjunction -> Sentence -> Sentence -> Sentence;
--Queries
DoesItVerb : Verb -> Phrase;
DoesTheNounVerb : Noun -> Verb -> Phrase;
IsNounVerbAdv : Noun -> VerbPhrase -> Adverb -> Phrase;
--Commands
ActionNounCommand : Verb2 ->  Noun -> Phrase;
ModifyActionCommand : Verb -> Adverb -> Phrase;
--Constraints
TheNounHasNumberAtMost: Noun ->  NumeralAdverb -> Number -> Noun ->
  Sentence;
ItHasNumberAtLeast:   NumeralAdverb -> Number -> Noun -> Sentence;
--Base Categories
Action : V -> Verb;
Object : N -> Noun;
Numbers : Digits -> Number;
AtMost,AtLeast,MoreThan,LessThan: NumeralAdverb;
```

Figure 13. Extract from BONAbs function

The “Overall” functions in figure 13 show how the Output category is produced.. The MakeSentenceConj function is a sentence made up of a Conjunction category and two Sentence categories. Either or both of the Sentence categories within the function could be produced by the same function which allows produce quite complex output.

The functions are named in way to describe the type of text to be parsed. The “DoesItVerb” function parses queries like “Does it run?” “Does it walk?”.

Three of the functions in the Base Categories Action, Object and Number produce the categories Verb, Noun and Number from the categories V, N and Digits respectively. The categories are not described in the BONAbs.gf file; instead they are inherited from the Cat abstract syntax through the extension of DictEngAbs.

3.1.2 InformalBON.gf

The InformalBON.gf file is the concrete syntax used to linearize the BONAbs syntax trees to English natural language. The categories and functions from the English resource grammar library are used to complete this task. Figure 14 shows an extract from the header of the InformalBON syntax. Appendix 7.3 shows the full text from InformalBON.gf

```
--# -path=.:../alltenses


concrete InformalBON of BONAbs = DictEng,BONTermsEng,NumeralEng **
open SyntaxEng,ParadigmsEng in{

   lincat
     Output = Text;
     Phrase = Phr;
     Sentence = S;
     Interrogative=IP;
     Noun = CN;
     Adjetive = AP;
     Verb = V;
```

Figure 14. Extract from InformalBON header

Figure 14 shows InformalBON is created from BONAbs and extends DictEng, BONTermEng and NumeralEng. In doing so InformalBON includes the linearization’s of the categories and functions within the extended concrete syntaxes. The concrete syntaxes which InformalBON extends are linked to the abstract syntaxes which BONAbs extends.

The open command allows the syntax access to the (in this case) English Resource library. This is common command used at the start of most Gf concrete syntaxes. The linearization of the categories in InformalBON corresponds to category types taken from the resource library. Functions which produce the Output category must be linearized to produce Text. Figure 15 displays an extract from the functions in InformalBON.

```
--Overall
MakeText phrase punctuation = mkText phrase punctuation;
MakeTextSentence  sentence = mkText sentence;
MakeSentenceConj conjunction sentence1 sentence2 = mkS conjunction
sentence1 sentence2;
--Queries
DoesItVerb verb = mkPhr(mkQS  (mkCl (mkVP verb)));
DoesTheNounVerb noun verb = mkPhr(mkQS  (mkCl (mkNP (mkDet the_Quant)
noun) (mkVP verb)));
IsNounVerbAdv noun verbphrase adverb = mkPhr(mkQS  (mkCl (mkNP  (mkDet
the_Quant)  noun)       (mkVP verbphrase adverb)));
--Commands
ActionNounCommand verb2 noun = mkPhr(mkImp verb2 (mkNP noun));
ModifyActionCommand verb adverb = mkPhr(mkImp (mkVP (mkVP verb) adverb));
--Constraints
TheNounHasNumberAtMost  noun1 numeraladverb number noun2 = mkS( mkCl(mkNP
(mkDet    the_Quant) noun1) have_V2 (mkNP  (mkCard numeraladverb (mkCard
number)) noun2));
ItHasNumberAtLeast  numeraladverb number noun = mkS( mkCl(mkNP it_Pron)
have_V2 (mkNP    (mkCard numeraladverb (mkCard number)) noun));
--Base Class
Action v = v;
Object n =   mkCN n;
Numbers d = d;
AtMost = at_most_AdN;
AtLeast = at_least_AdN;
MoreThan = mkAdN "more than";
LessThan = mkAdN "less than";
```

Figure 15. Extract from linearization functions in Informal Bon

At first glance the linearization of the syntax trees can look quite complex. It can be broken down into its different element to explain how the Text is constructed. Using the function DoesTheNounVerb as an example , the abstract syntax (figure 14) shows the function creates a Phrase category from the Noun and Verb categories. These three categories are linearized to Phr (phrase), CN (common noun) and V (verb) respectively in the concrete syntax. The Noun category is created from the Object function. The linearization of the Object function takes a noun, N and returns a common noun, CN. The Verb category is created from the Action function. This function takes a verb, V and returns a verb, V. The DoesTheNounVerb has two parameters noun and verb which have been linearized to CN and V. It creates a Phr from a Question mkQS, the question is in turn created from a Clause mkCl. This Clause, Cl is created from combining a noun phrase, mkNP and a verb phrase, mkVP. The noun phrase NP is made up of a Determiner mkDet and a common noun CN .The CN in this case is the parameter “noun”. The Determiner is in this example is made from a Quantifier Quant, the quantifier used is “the” or the_Quant (the resource grammar libraries representation of “the”) . The second half of the Clause category, the verb phrase (mkVP), is created from the second parameter in the function verb. The construction of this linearization function is entirely sourced from the resource API. This function provides the grammar with the ability to parse and linearize any query with the structure Does the [noun] [verb]? Where [noun] can be any noun i.e. car, house, cat, dog and [verb] can be any verb i.e. move, run, walk etc.

3.1.3 FormalBON.gf

The FormalBON concrete syntax is used to linearize the BONAbs syntax trees to formal BON typed notation. To complete this task a combination of resources from the English grammar library and Strings (like the Family example discussed earlier) are used. The notation symbols used in BON are not available through Gf which forced the use of strings. An extract from the header of the FormalBON concrete syntax is shown in figure 16. Appendix 7.4 shows the full

```
          --# -path=.:../alltenses

           concrete FormalBON of BONAbs = DictEng,BONTermsEng,NumeralEng **
     open   SyntaxEng,ParadigmsEng in {

            lincat
              Output = { s : Str};
              Phrase= {s : Str};
              Sentence = {s : Str};
              Noun = {s : Str};
              Interrogative= {s : Str};
              Adjetive = {s : Str};
              Verb = {s : Str};
```

Figure 16. Extract from FormalBON header

The FormalBON header is similar to that of the InformalBON syntax header. Both extend and open the same syntaxes. The obvious difference is, all categories in the FormalBON syntax are linearized to Strings. Figure 17 shows an extract from the functions section of the FormalBON syntax.

```
--Overall
MakeText phrase punctuation = {s = phrase’s};
MakeTextSentence  sentence = {s = sentence’s};
MakeSentenceConj conjunction sentence1 sentence2 = {s = sentence1.s ++
conjunction’s ++       sentence2.s};
--Queries
DoesItVerb verb = {s =  verb’s  ++ ":" ++ "BOOLEAN"};
DoesTheNounVerb noun verb = {s =  verb.s ++ ":" ++ "BOOLEAN" ++ "->"  ++
noun.s ++":      VALUE"};
IsNounVerbAdv noun verbphrase adverb = {s = verbphrase.s  ++ "_" ++
adverb.s ++ ":" ++      "BOOLEAN"};
ActionNounCommand verb2 noun = {s = verb2.s ++ "_" ++ noun.s};
ModifyActionCommand verb adverb = {s = verb.s ++ "_" ++ adverb.s};
--Constraints
ANounHasNumberAtMost noun1  numeraladverb number noun2 = {s = noun2.s ++
".count" ++      numeraladverb.s  ++ number.s };
ItHasNumberAtLeast  numeraladverb number noun =  {s = noun.s ++".count" +
+ numeraladverb.s      ++ number.s };
--Base Class
Object n = mkUtt (mkNP(mkCN n));
Action v = mkUtt(mkVP v);
Numbers d = mkUtt (mkCard d) ;
AtMost = {s = "<="};
AtLeast = {s = ">="};
MoreThan = {s = ">"};
LessThan = {s = "<"};
```

Figure 17. Extract from linearization functions in Formal Bon

The FormalBON syntax functions look totally different to that of the InformalBON syntax. The categories in both syntaxes are linearized to different types and yet both work because the different functions requirements are been fulfilled. Using the function DoesTheNounVerb used as a previous example, the function fulfils the requirements of producing a Phrase category from Noun and Verb categories. The abstract syntax of the function has not been altered. As mentioned the FormalBON syntax uses a combination of the resource library and strings. The use of the resource library is seen in the Object, Action and Numbers functions. When discussing the BONAbs class it was mentioned that the Object, Action and Numbers functions produce the Noun, Verb and Digits categories respectively. The Noun, Verb and Digits categories are created from the categories N, V and d categories. These categories are inherited from the Cat abstract syntax located in the resource library. The CatEng concrete syntax (which FormalBON inherits) linerizes the three categories to actual nouns N, verbs V and digit d. In order to use the categories the FormalBON syntax must use the resource library API. Converting categories from the resource library to a string involves converting to an Utt (Utterence), Gf's equivalent to a string. The Object function has an actual resource library noun N as a parameter and produces a Noun category which in the FormalBON syntax case is a string.

3.1.4 DictEngAbs.gf and DictEng.gf

The English Dictionary abstract syntax (DictEngAbs.gf) and concrete syntax (DictEng) files are extended by the BONAbs abstract syntax and the InformalBON and FormalBON concrete syntax. The BON files inherit all that is included in these files which includes for over 4000 words. The inclusion DictEngAbs and DictEng in the application allows linearization of all the words included recognising the words in their various different forms .i.e. singular, plural, polite plural etc.

DictEngAbs

The abstract syntax of the English dictionary files describes a function for each of the words in the English Dictionary. An extract from the start of the file is seen in figure 4.5

```
abstract DictEngAbs = Cat ** {
fun a_bomb_N : N;
fun a_fortiori_Adv : Adv;
fun a_level_N : N;
fun a_posteriori_A : A;
fun a_posteriori_Adv : Adv;
fun a_priori_A : A;
fun a_priori_Adv : Adv;
fun aa_N : N;
fun aachen_PN : PN;
fun aarhus_PN : PN;
fun ab_initio_Adv : Adv;
fun aback_Adv : Adv;
fun abacus_N : N;
fun abaft_Adv : Adv;
fun abandon_N : N;
fun abandon_V2 : V2;
fun abandoned_A : A;
fun abandonment_N : N;
```

Figure 18. Extract from DictEngAbs

The DictEngAbs file does not define its own categories but inherits the categories from Cat. Cat is an abstract syntax located within the resource library which defines all the categories used within Gf. Each function within the abstract syntax creates a category inherited from Cat. The function a_bomb_N creates a category N and the function abandon_V2 creates a category V2.

DictEng

The concrete syntax of the English Dictionary files describes how each of the functions defined in the abstract syntax are linearized. The concrete syntax extends CatEng, the concrete syntax of the Cat abstract syntax. The N category is linearized in CatEng to form a noun meaning all functions in DictEngAbs which create an N category must be linearized to create a noun in DictEng. An extract from DictEng is seen in figure 4.6.

```
concrete DictEng of DictEngAbs = CatEng ** open ParadigmsEng, IrregEng in
{

flags
  coding=utf8 ;

lin a_bomb_N = mkN "a-bomb" "a-bombs";
lin a_fortiori_Adv = mkAdv "a fortiori";
lin a_level_N = mkN "a-level" "a-levels";
lin a_posteriori_A = compoundA (mkA "a posteriori");
lin a_posteriori_Adv = mkAdv "a posteriori";
lin a_priori_A = compoundA (mkA "a priori");
lin a_priori_Adv = mkAdv "a priori";
lin aa_N = mkN "aa" "-" {- FIXME: no plural form -};
lin aachen_PN = mkPN "Aachen";
lin aarhus_PN = mkPN "Aarhus";
lin ab_initio_Adv = mkAdv "ab initio";
lin aback_Adv = mkAdv "aback";
lin abacus_N = mkN "abacus" "abacuses";
lin abaft_Adv = mkAdv "abaft";
lin abandon_N = mkN "abandon" "-" {- FIXME: no plural form -};
lin abandon_V2 = mkV2 (mkV "abandon" "abandons" "abandoned" "abandoned"
"abandoning");
```

Figure 19. Extract from DictEngAbs`

The a_bomb_N linearizes to a noun by using the "mkN string string " paradigms. English Paradigms are available to the concrete syntax as it is included in the open statement in the header of the file. Paradigms provide numerous ways to create the lower categories within Gf. There is a set of paradigms for each language. The abandon_V2 creates a two place verb V2 from a verb with five different forms.

3.1.5 BONTermsAbs and BONTermsEng

The BONTermsAbs and BONTerms syntaxes are extensions of the DictEngAbs and DictEng. They where included to allow new words specifically used when dealing with BON that are not icluded in the English Dictionary file. The Java API will provide a method to allow the user to update the BON Terms files to allow the grammar access to them.

3.2 Java API and Classes

The Java API and Java classes created as part of the ESC/IBON to handle the task of launching and communicating with the Gf application. The Gf application is launched and run as a background process, the BON Gf files are imported into Gf and the Java API provides a method of communicating with the Gf grammar. A description of each Java class follows.

3.3 Gf.java

The Gf java class uses system knowledge to choose the correct Gf binary to launch the Gf application. There are specific Gf binaries depending on the operating system in use. The class then creates a GFProcess passing it the binary in the form of a temporary file.

3.4 GfProcess.java

The GfProcess class launches the Gf executable. A Java OutputStreamWriter and two Java InputStreamReaders (the first for regular input and the other to read errors) are created to communicate with the Gf application. The OutputStramWriter is used to pass Gf commands to the process using a Java BufferedWriter and the InputStreamReaders collect the response using a BufferedReader.

3.5 FileUtils.java

The FileUtils class as its name suggests completes all file tasks such as the creation of the temporary file used to launch the Gf application and reading and writing to the file. The class also copies the BON Gf executable files to a local drive to allow the Gf application gain access to them.

3.6 GfCommands.java

This is the API for accessing the GfProcess. Figure 20 shows the methods available through the API

```
public interface GfCommands {

      public String translateFromFormalBON(String Sentence);

      public String translateToFormalBON(String Sentence);

      public Map translateQueryToFormalBON(String Sentence);

      public String importLanguage(String language)throws IOException;

      public String addBONTerm(String type,String singular,String
      Plural)throws IOException;

      public void quitProcess()throws IOException;

      public boolean ProcessIsAlive();
      }
```

Figure 20.   Java API

3.7 GfCommandsImpl.java

This is the implementation of the GfCommands interface. The class creates Gf commands from the string commands passed through the API. The string “What is the cars color” for example, being passed through the method translatetoFormalBON(string Sentence) is converted to the command seen in below, before being passed to the base classes

```
ps -lextext “What is the cars color?” | p -lang=InformalBON | l -
lang=FormalBON
```

4 Evaluation and Testing

Before testing the application the vocabulary or fragment of language which the tool could parse was compiled. The vocabulary is broken down by queries, commands and constraints

4.1 BON Queries

The of queries which ESC/IBON will translate is as follows,

Informal BON	Example	Formal BON
[noun]?	House?	house:BOOLEAN
[verb]?	Go?	go : BOOLEAN
[adjetive]?	Big?	big : BOOLEAN
What is [pronoun][noun]?	What is its name?	name : VALUE
What is the [noun][noun]	What is the man's name?	name : VALUE -> car : VALUE
What is its value?	What is its value?	value : INTEGER
Is it [adjetive]?	Is it single?	single : BOOLEAN
Does it [verb]?	Does it move?	move : BOOLEAN
Does the [noun] [verb]?	Does the car move?	move : BOOLEAN

		-> car : VALUE
Is it [verb]?	Is it moving?	moving : BOOLEAN
Is the [noun][verb]?	Is the car moving?	moving : BOOLEAN -> car : VALUE
Is it [verb][adverb]?	Is it moving up?	moving_up : BOOLEAN
Is the [noun][verb][adverb]	Is the elevator moving up?	moving_up : BOOLEAN -> car : VALUE
Is the [noun] a [noun]?	Is the cat an animal?	cat : SET[animal]
Is a [noun] a [noun]?	Is a dog an animal?	dog : SET[animal]
Does the [noun] [verb]?	Does the bird fly?	fly : BOOLEAN -> bird : VALUE
How many [noun] is there?	How many birds is there?	birds : INTEGER
What is [pronoun][noun]?	Who is your doctor?	doctor : VALUE

Figure 21. BON Queries

4.2 BON Commands

The list of command which ESC/IBON will translate is as follows,

Informal BON	Example	Formal BON
[verb][adverb]?	Move right!	move_right
[verb]?	Go!	go
[adjective]?	Big!	Big
[verb] a/the [noun]	Move the Car!	move -> car : VALUE
[verb][noun]	Marry Emma!	Marry -> emma : VALUE
[verb][noun] the [noun]	Give John the book	Give_book ->

		John : VALUE Book : VALUE

Figure 22. BON Command

4.3 BON Constraints

The of constraints which ESC/IBON will translate is as follows,

Informal BON	Example	Formal BON
There are at most/least [number][noun]?	There are at most 3 cars?	Cars.count <= 3
A/The [noun] has at most/least [number][noun]	A man has at least 3 chidren.	Exists man it_holds Children.count >= 3.
A/The [noun] has [number][noun]	A man has 3 children.	Exists man it_holds Children.count = 3.
It has at most/least [number][noun]	it at most 3 chidren.	Children.count <= 3.
It has [number][noun]	It has 3 children.	Children.count = 3.
The/A [noun] exists	The amimal exists	Animal /=VOID
No [noun] exists	No animal exists	Animal = VOID
The/A [noun] is a [noun]	The dog is an animal	Dog member_of animal
The/A [noun] isn't a [noun]	The fish isn't an animal	Fish not member_of animal
It is [verb]?	It is moving?	move
The [noun]is [verb]?	The car is moving?	car_move
It is [verb][adverb]?	It is moving up?	moving_up

It is [verb]?	It isn't moving?	Not move
The [noun]is [verb]?	The car isn't moving?	Not car_move
It is [verb][adverb]?	It isn't moving up?	Not moving_up
The [noun] is [verb][adverb]	The elevator isn't moving up?	Not Elevator.moving_up

Figure 23. BON Constraints

A number of constraints separated by a conjunction can be translated at anyone time. The list of conjunctions available is as follows,

- And
- Or
- If, then
- Exclusive or
- Such that

The ability to combine constraints allows ESC/IBON translate quite complex constraints. An example of complex translation is as follows,

“If it has at most 4 wheels then the vehicle is a car or the vehicle is a tractor and the vehicle is not a truck” is translated to

```
for all wheel.count <= 4 it holds vehicle member of car or vehicle member
of tractor and vehicle not member of truck
```

4.4 Testing

Testing to this point has only involved using J-Unit testing to test each of the queries, commands and constraints listed. Further testing has involved using complex constraints. The next step in testing is to compile a list of BON queries, constraints and commands from existing BON documentation and run them through the tool. This will highlight the gaps in the vocabulary. When the tool is in use we will get further feedback on the types of BON phrases frequently used which are not catered for.

5 Conclusion and Further Work

ESC/IBON has fulfilled its initial goal in that we have a tool which can translate a limited subsection of natural language to Formal BON. One of the benefit of using Gf has been the reverse is also the case, English language can be translated from formal BON.

The next step in the process will be to increase number of phrases that the tool can translate. A formal test using existing documentation will provide some information on the holes in the vocabulary. BON notation allows for arithmetic i.e value + 50000 unfortunatly this has not been included in the tool but is something that can be added in the future.

Gf as tool provides the platform to further increase the complexity of the translations. Moving away from the use strings in the FormalBON concrete syntax to the development of a Formal BON notation library would provide a lot more flexibility with the types of translations. The Introduction of a Formal BON library to Gf would also provide the possibility to translate from languages other than English to Formal BON. Users could write Informal BON charts in their native language translate the charts to Formal BON and they could be translated to English for those who only speak English and broken Irish!

Finally, the use of online ontholagies such as WordNet could provide further information into exactly what a person means when they write informal BON. Gf identifies the type of words written but WordNet could provide meaning. For Example ESC/IBON can translate the query "who are your brothers?" into "borthers :SET[]" . Using WordNet we could take the translation further, translating the query into "brothers: SET[PEOPLE]".

6 References

1. Business Object Notation, Kim Walden http://www.bon-method.com/handbook_bon.pdf

2. Seamless Object-Orientated Software Architecture: Kim Walden and Jean-Marc Nearson. http://www.bon-method.com/book_print_a4.pdf

3. A Comparison of the Business Object Notation and the Unified Modeling Language: Richard F Paige and Johnathan S. Ostroff http://www.cse.yorku.ca-/~jonathan/publications/1999/uml99.pdf

4. HSE #4 BON. The Business Object Notation : Joe Kiniry http://vimeo.com/groups/-43603/videos/10803576

5. HSE #1 Introduction : Joe Kiniry http://vimeo.com/groups/43603/videos/10803334

6. HSE #3 Effectively Using Formal Methods : Joe Kiniry http://vimeo.com/groups-/43603/videos/10803511

7. Formal and Informal Specifications: Kristofer Johannisson. http://www.cs.chalmers.se/-~krijo/thesis/thesisA4.pdf

8. Grammars as Software libraries: A Ranta http://-www.cs.chalmers.se/~aarne/articles/libraries-kahn.pdf

9. A Mutilingingual Grammer Formalism: A Ranta

10. The Grammatical Framework website http://www.grammaticalframework.org/

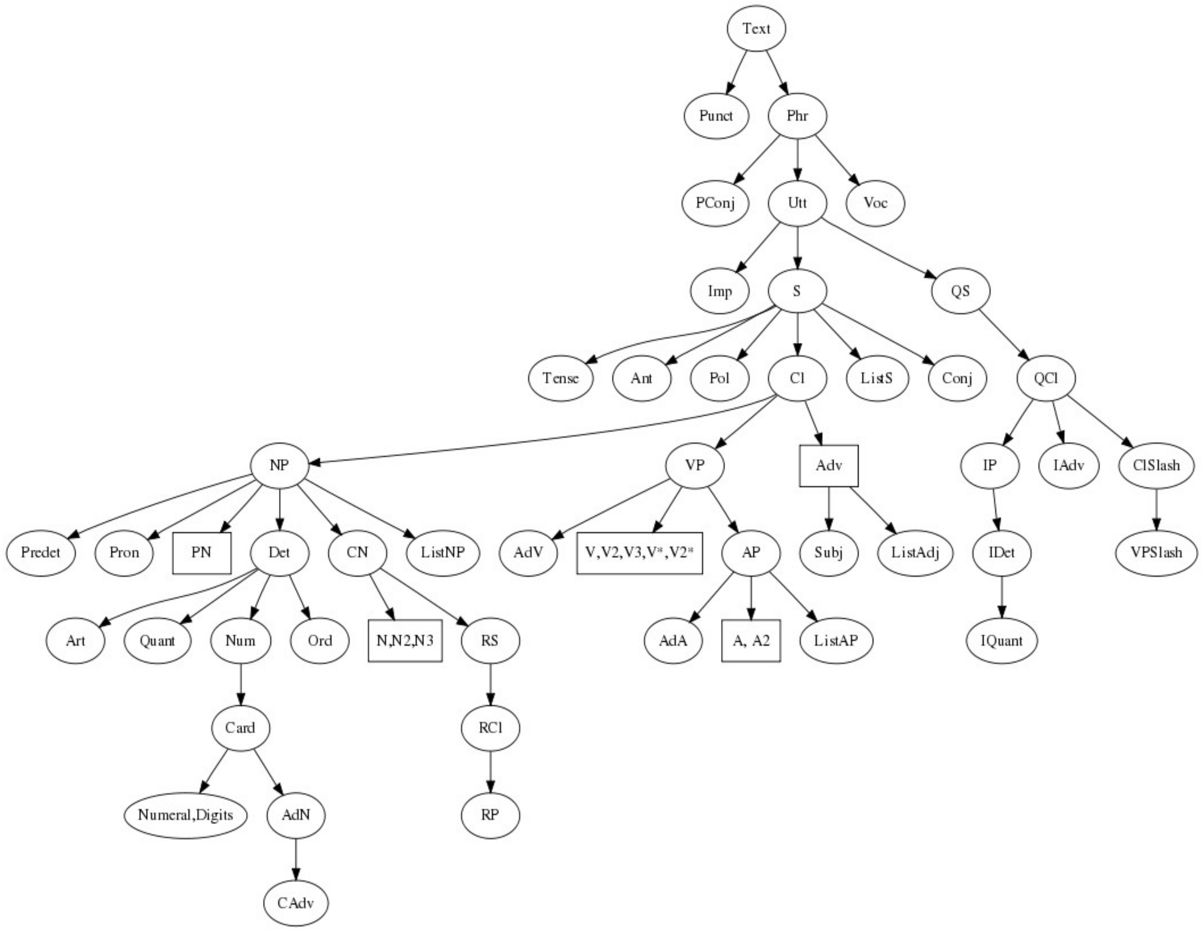
11. Gf Google Group website http://google.groups.ie/group/gf-dev?hl=en

12. Gf Resource Grammer Summer School 2009 http://google.groups.ie/group/gf-resource-school-2009?hl-en

13. Gf as a Functional-logic Language: Krasimir Angelov

7 Appendices

7.1 GF Hierarchical of Categories



7.2 BONAbs.gf

```
abstract BONAbs = DictEngAbs,BONTermsAbs,Numeral ** {

flags startcat=Output;

cat

Output;
Phrase;
Sentence;
SentenceList;
Interrogative ;
Noun;
Adjective;
Verb;
ComplementVerb;
VerbPhrase;
Pronoun;
Number;
NumeralAdverb;
Verb2;
IQuantifier;
IDeterminer;
Quantifier;
Imperative;
Adverb;
Punctuation;
NounPhrase;
Conjunction;
CAdverb ;



fun

--Overall
MakeText : Phrase -> Punctuation -> Output;
MakeTextSentence : Sentence -> Output;
--MakeSentenceFromList : Conjunction -> SentenceList -> Sentence;
MakeSentenceConj : Conjunction -> Sentence -> Sentence -> Sentence;
--MakeSentenceList : Sentence -> Sentence -> SentenceList;
--AddToSentenceList : Sentence -> SentenceList -> SentenceList;


--Queries
NounQuest : Noun -> Phrase;
AdjectiveQuest : Adjective -> Phrase;
WhatIsTheNounsNoun : Noun -> Noun -> Phrase;
WhatQuestValue: Phrase;
WhatQuestNoun :  NounPhrase -> Phrase;
WhoQuestNoun :  NounPhrase -> Phrase;
DoesItVerb : Verb -> Phrase;
DoesTheNounVerb : Noun -> Verb -> Phrase;
IsItVerb : VerbPhrase -> Phrase;
IsNounVerb : Noun -> VerbPhrase -> Phrase;
IsItVerbAdv : VerbPhrase -> Adverb -> Phrase;
```

```
IsNounVerbAdv : Noun -> VerbPhrase -> Adverb -> Phrase;
IsItAdj : Adjective -> Phrase;
IsTheNounANoun:  Noun -> Noun -> Phrase;
IsANounANoun:  Noun -> Noun -> Phrase;
HowManyNoun :IDeterminer -> Noun -> Phrase;
WhichNoun : IQuantifier -> Noun  -> Phrase;



--Commands
ActionCommand : Verb -> Phrase;
ActionNounCommand : Verb2 ->  Noun -> Phrase;
ModifyActionCommand : Verb -> Adverb -> Phrase;

--Constraints

CanMust : ComplementVerb -> Verb -> Sentence;
CannotMustnot : ComplementVerb -> Verb -> Sentence;
AtMostLeast : NumeralAdverb -> Number -> Noun -> Sentence;
ANounHasNumberAtMost: Noun ->  NumeralAdverb -> Number -> Noun ->
Sentence;
TheNounHasNumberAtMost: Noun ->  NumeralAdverb -> Number -> Noun ->
Sentence;
ItHasNumberAtLeast:   NumeralAdverb -> Number -> Noun -> Sentence;
ItHasNumber:Number -> Noun -> Sentence;
ANounHasNumber: Noun ->  Number -> Noun -> Sentence;
TheNounHasNumber: Noun ->  Number -> Noun -> Sentence;
TheNounExists:  Noun -> Sentence;
ANounExists:  Noun -> Sentence;
NoNounExists: Noun -> Sentence;
TheNounIsNoun:Noun ->Noun ->Sentence;
TheNounIsNotNoun:Noun ->Noun ->Sentence;
ANounIsNoun:Noun ->Noun ->Sentence;
ANounIsNotNoun:Noun ->Noun ->Sentence;
ItIsVerb : VerbPhrase -> Sentence;
NounIsVerb : Noun -> VerbPhrase -> Sentence;
ItIsVerbAdv : VerbPhrase -> Adverb -> Sentence;
NounIsVerbAdv : Noun -> VerbPhrase -> Adverb -> Sentence;
ItIsNotVerb : VerbPhrase -> Sentence;
NounIsNotVerb : Noun -> VerbPhrase -> Sentence;
ItIsNotVerbAdv : VerbPhrase -> Adverb -> Sentence;
NounIsNotVerbAdv : Noun -> VerbPhrase -> Adverb -> Sentence;

--Noun Phrases
PronounNounPhrase : Pronoun ->Noun -> NounPhrase;

--Verb Phrases
ProgressiveVerbPhrase : V -> VerbPhrase;

--Base Categories
Action : V -> Verb;
Object : N -> Noun;
ModifyAction : Adv -> Adverb;
AdjectivePhrase: A -> Adjective;
TwoPlaceAction : V2 -> Verb2;
Numbers : Digits -> Number;
What,Who:Interrogative;
Its,You,He,She,We,They,Pron_I:Pronoun;
Which : IQuantifier;
HowMany: IDeterminer;
AtMost,AtLeast,MoreThan,LessThan: NumeralAdverb;
The,QuantA,That,This,No: Quantifier;
FullStop,Exclamation,QuestionMark: Punctuation;
```

```
Can,Must: ComplementVerb;
ConjunctionOr,
ConjunctionAnd,ConjunctionIf,ConjunctionXor,ConjunctionSuchThat:
Conjunction;
--More,Less: CAdverb;


}
```

7.3 InformalBON.gf

```
--Copyright (c) 2007-2009, Aidan Morrissey, University College Dublin
under the BSD licence.

--# -path=.:../alltenses


concrete InformalBON of BONAbs = DictEng,BONTermsEng,NumeralEng ** open
SyntaxEng,ParadigmsEng in{

lincat
Output = Text;
Phrase = Phr;
Sentence = S;
SentenceList = ListS;
Interrogative=IP;
Noun = CN;
Adjective = AP;
Verb = V;
ComplementVerb = VV;
Verb2 = V2;
VerbPhrase = VP;
Pronoun = Pron;
Number = Digits;
NumeralAdverb = AdN;
IQuantifier = IQuant;
IDeterminer = IDet;
Quantifier = Quant;
Imperative = Imp;
Adverb = Adv;
CAdverb = CAdv;
Punctuation = Punct;
NounPhrase = NP;
Conjunction = Conj;


lin

--Overall
MakeText phrase punctuation = mkText phrase punctuation;
MakeTextSentence  sentence = mkText sentence;
--MakeSentenceFromList conjunction sentencelist = mkS conjunction
sentencelist;
MakeSentenceConj conjunction sentence1 sentence2 = mkS conjunction
sentence1 sentence2;
--MakeSentenceList sentence1 sentence2 = mkListS sentence1 sentence2;
--AddToSentenceList sentence sentencelist = mkListS sentence
sentencelist;

--Queries
```

```
WhatIsTheNounsNoun noun1 noun2 = mkPhr(mkQS  ( mkCl(mkNP (mkDet
the_Quant) noun1) (mkVP (mkNP (mkDet a_Quant) noun2))));
NounQuest noun =mkPhr(mkUtt (mkNP noun));
--AdjectiveQuest adjetive =mkPhr(mkUtt adjetive);
WhatQuestNoun  nounphrase =mkPhr(mkQS (mkQCl whatSg_IP nounphrase));
WhatQuestValue  =mkPhr(mkQS (mkQCl whatSg_IP (mkNP it_Pron (mkCN
value_N))));
WhoQuestNoun  nounphrase =mkPhr(mkQS (mkQCl whoPl_IP nounphrase));
IsItAdj adjetive = mkPhr(mkQS (mkCl(mkVP adjetive)));
DoesItVerb verb = mkPhr(mkQS  (mkCl (mkVP verb)));
DoesTheNounVerb noun verb = mkPhr(mkQS  (mkCl (mkNP (mkDet the_Quant)
noun) (mkVP verb)));
IsItVerb verbphrase = mkPhr(mkQS  (mkCl verbphrase));
IsNounVerb noun verbphrase = mkPhr(mkQS  (mkCl (mkNP  (mkDet the_Quant)
noun) verbphrase));
IsItVerbAdv verbphrase adverb = mkPhr(mkQS  (mkCl (mkVP verbphrase
adverb)));
IsNounVerbAdv noun verbphrase adverb = mkPhr(mkQS  (mkCl (mkNP  (mkDet
the_Quant)  noun) (mkVP verbphrase adverb)));
IsTheNounANoun noun1 noun2 = mkPhr(mkQS (mkCl (mkNP  (mkDet the_Quant)
noun1) (mkNP (mkDet a_Quant) noun2)));
IsANounANoun noun1 noun2 = mkPhr(mkQS (mkCl (mkNP  (mkDet a_Quant)
noun1) (mkNP (mkDet a_Quant) noun2)));
HowManyNoun idet noun = mkPhr(mkQS  (mkQCl (mkIP idet noun)));
WhichNoun iquant noun =   mkPhr(mkQS  (mkQCl (mkIP iquant noun )));


--Commands
ActionCommand verb = mkPhr(mkImp verb); -- returns imp i.e go
ActionNounCommand verb2 noun = mkPhr(mkImp verb2 (mkNP noun));-- returns
imp i.e move the car
ModifyActionCommand verb adverb = mkPhr(mkImp (mkVP (mkVP verb)
adverb));-- returns imp i.e move right

--Constraint
AtMostLeast  numeraladverb number noun = mkS( mkCl(mkNP  (mkCard
numeraladverb (mkCard number)) noun));
CanMust complementverb verb = mkS( mkCl(mkVP  complementverb (mkVP
verb)));
CannotMustnot complementverb verb = mkS negativePol ( mkCl(mkVP
complementverb (mkVP verb)));
ANounHasNumberAtMost  noun1 numeraladverb number noun2 = mkS( mkCl(mkNP
(mkDet a_Quant) noun1) have_V2 (mkNP  (mkCard numeraladverb (mkCard
number)) noun2));
TheNounHasNumberAtMost  noun1 numeraladverb number noun2 = mkS( mkCl(mkNP
(mkDet the_Quant) noun1) have_V2 (mkNP  (mkCard numeraladverb (mkCard
number)) noun2));
ANounHasNumber  noun1 number noun2 = mkS( mkCl(mkNP (mkDet a_Quant)
noun1) have_V2 (mkNP  (mkCard number) noun2));
TheNounHasNumber  noun1 number noun2 = mkS( mkCl(mkNP (mkDet the_Quant)
noun1) have_V2 (mkNP  (mkCard number) noun2));
TheNounExists  noun  =  mkS ( mkCl(mkNP (mkDet the_Quant) noun) exist_V);
ANounExists noun  =  mkS( mkCl(mkNP (mkDet a_Quant) noun) exist_V);
NoNounExists  noun  =  mkS ( mkCl(mkNP (mkDet no_Quant) noun) exist_V);
TheNounIsNoun noun1 noun2 = mkS ( mkCl(mkNP (mkDet the_Quant) noun1)
(mkVP (mkNP (mkDet a_Quant) noun2)));
TheNounIsNotNoun noun1 noun2 = mkS negativePol ( mkCl(mkNP (mkDet
the_Quant) noun1) (mkVP (mkNP (mkDet a_Quant) noun2)));
ANounIsNoun noun1 noun2 = mkS ( mkCl(mkNP (mkDet a_Quant) noun1) (mkVP
(mkNP (mkDet a_Quant) noun2)));
ANounIsNotNoun noun1 noun2 = mkS negativePol ( mkCl(mkNP (mkDet a_Quant)
noun1) (mkVP (mkNP (mkDet a_Quant) noun2)));
```

```
NounIsVerb noun verbphrase = mkS (mkCl (mkNP  (mkDet the_Quant)  noun)
verbphrase);
NounIsVerbAdv noun verbphrase adverb = mkS (mkCl (mkNP  (mkDet the_Quant)
noun) (mkVP verbphrase adverb));
NounIsNotVerb noun verbphrase = mkS negativePol (mkCl (mkNP  (mkDet
the_Quant)  noun) verbphrase);
NounIsNotVerbAdv noun verbphrase adverb = mkS negativePol (mkCl (mkNP
(mkDet the_Quant)  noun) (mkVP verbphrase adverb));
ItIsVerbAdv verbphrase adverb = mkS (mkCl (mkVP verbphrase adverb));
ItIsNotVerb verbphrase = mkS negativePol (mkCl verbphrase);
ItIsVerb verbphrase = mkS (mkCl verbphrase);
ItHasNumberAtLeast  numeraladverb number noun = mkS( mkCl(mkNP it_Pron)
have_V2 (mkNP  (mkCard numeraladverb (mkCard number)) noun));
ItHasNumber  number noun = mkS( mkCl(mkNP it_Pron) have_V2 (mkNP  (mkCard
number) noun));
ItIsNotVerbAdv verbphrase adverb = mkS negativePol(mkCl (mkVP verbphrase
adverb));

-- NounPhrases
PronounNounPhrase  pronoun noun = mkNP pronoun noun;


--VerbPhrases
ProgressiveVerbPhrase v = progressiveVP (mkVP v);

--AdjectivePhrases
AdjectivePhrase a = mkAP a ;

--Base Categoies
Object n =   mkCN n;
Action v = v;
TwoPlaceAction v2 = v2;
ModifyAction adv = adv;

--Interrogative
What = whatSg_IP;
Who = whoPl_IP;
Which = which_IQuant;
HowMany = how8many_IDet;

--Pronoun
Its = it_Pron;
You = youSg_Pron;
He = he_Pron;
Pron_I = i_Pron;
She=she_Pron;
They = they_Pron;
We =we_Pron;

--Digits
Numbers d = d;

--Numeral Adverb
AtMost = at_most_AdN;
AtLeast = at_least_AdN;
MoreThan = mkAdN "more than";
LessThan = mkAdN "less than";

--Quantifier
The = the_Quant;
QuantA = a_Quant;
That = that_Quant;
This = this_Quant;
```

```
No = no_Quant;

--Punctuation
FullStop = fullStopPunct;
Exclamation = exclMarkPunct;
QuestionMark = questMarkPunct;

--Complement_Verb
Can = can_VV;
Must = must_VV;

--Conjunctios
ConjunctionOr = or_Conj;
ConjunctionAnd = and_Conj;
ConjunctionIf = if_then_Conj;
ConjunctionXor = mkConj "exlcusive or";
ConjunctionSuchThat = mkConj "such that";

}
```

7.4 FormalBON.gf

```
--Copyright (c) 2007-2009, Aidan Morrissey, University College Dublin
under the BSD licence.

--# -path=.:../alltenses


concrete FormalBON of BONAbs = DictEng,BONTermsEng,NumeralEng ** open
SyntaxEng,ParadigmsEng in {

lincat
Output = { s : Str};
Phrase= {s : Str};
Sentence = {s : Str};
SentenceList = {s : Str};
Noun = {s : Str};
Interrogative= {s : Str};
Adjective = {s : Str};
Verb = {s : Str};
ComplementVerb = {s : Str};
Verb2 = {s : Str};
VerbPhrase = {s : Str};
Pronoun = {s : Str};
Number = {s : Str};
NumeralAdverb = {s : Str};
Quantifier = {s : Str};
IQuantifier = {s : Str};
IDeterminer = {s : Str};
Imperative = {s : Str};
Adverb = {s : Str};
Punctuation = {s : Str};
NounPhrase = {s : Str};
Conjunction = {s : Str};

lin

--Overall
MakeText phrase punctuation = {s = phrase.s};
MakeTextSentence  sentence = {s = sentence.s};
```

```
--MakeSentenceFromList conjunction sentencelist = {s = sentencelist.s ++
conjunction.s};
MakeSentenceConj conjunction sentence1 sentence2 = {s = sentence1.s ++
conjunction.s ++ sentence2.s};
--MakeSentenceList sentence1 sentence2 = {s = sentence1.s ++ "," ++
sentence2.s};
--AddToSentenceList sentence sentencelist = {s = sentencelist.s ++ "," ++
sentence.s};

--Queries
WhatIsTheNounsNoun noun1 noun2 = {s =  noun1.s ++ ":" ++ "VALUE" ++ "->"
++ noun2.s ++": VALUE"};
NounQuest noun = {s =  noun.s ++ ": BOOLEAN"};
--AdjectiveQuest adjetive = {s =  adjetive.s ++ ": BOOLEAN"};
WhatQuestValue = {s = "value : INTEGER" };
WhatQuestNoun  nounphrase  = {s = nounphrase.s ++ "VALUE" };
WhoQuestNoun  nounphrase  = {s = nounphrase.s ++ "VALUE" };
IsItAdj adjetive = {s = adjetive.s ++ ":" ++ "BOOLEAN"};
DoesItVerb verb = {s =  verb.s  ++ ":" ++ "BOOLEAN"};
IsItVerb verbphrase = {s = verbphrase.s  ++ ":" ++ "BOOLEAN"};
IsNounVerb noun verbphrase = {s = verbphrase.s  ++ ":" ++ "BOOLEAN" ++ "-
>"  ++ noun.s ++": VALUE"};
IsItVerbAdv verbphrase adverb = {s = verbphrase.s  ++ "_" ++ adverb.s ++
":" ++ "BOOLEAN"};
IsNounVerbAdv noun verbphrase adverb = {s = verbphrase.s  ++ "_" ++
adverb.s ++ ":" ++ "BOOLEAN"++ "->"  ++ noun.s ++": VALUE"};
IsTheNounANoun  noun1 noun2 = {s = noun1.s ++ ":" ++ "SET[" ++ noun2.s ++
"]" };
IsANounANoun  noun1 noun2 = {s = noun1.s ++ ":" ++ "SET[" ++ noun2.s ++
"]" };
DoesTheNounVerb noun verb = {s =  verb.s ++ ":" ++ "BOOLEAN" ++ "->"  ++
noun.s ++": VALUE"};
HowManyNoun idet noun = {s = noun.s ++ ":" ++ idet.s};
WhichNoun iquant noun =   {s = iquant.s ++ ":" ++ noun.s};


--Commands
ActionCommand verb = {s = verb.s}; -- returns imp i.e go
ActionNounCommand verb2 noun = {s = verb2.s ++ "_" ++ noun.s};-- returns
imp i.e move  car
ModifyActionCommand verb adverb = {s = verb.s ++ "_" ++ adverb.s}; --
returns imp i.e move right

--Constraints
AtMostLeast  numeraladverb number noun = {s = noun.s ++ ".count" ++
numeraladverb.s  ++ number.s};
CanMust complementverb verb = {s = complementverb.s ++ verb.s};
--CannotMustnot complementverb verb = mkS negativePol ( mkCl(mkVP
complementverb (mkVP verb)));
ANounHasNumberAtMost noun1  numeraladverb number noun2 = {s = "exists" ++
noun1.s ++ "it_holds" ++ noun2.s ++ ".count" ++ numeraladverb.s  ++
number.s };
TheNounHasNumberAtMost noun1  numeraladverb number noun2 = {s = "exists"
++ noun1.s ++ "it_holds" ++ noun2.s ++ ".count" ++ numeraladverb.s  ++
number.s };
ANounHasNumber noun1  number noun2 = {s = "exists" ++ noun1.s ++
"it_holds" ++ noun2.s ++ ".count" ++ "=" ++ number.s };
TheNounHasNumber noun1  number noun2 = {s = "exists" ++ noun1.s ++
"it_holds" ++ noun2.s ++ ".count" ++ "=" ++ number.s };
ItHasNumber number noun = {s = noun.s ++".count" ++ "=" ++ number.s };
ItHasNumberAtLeast  numeraladverb number noun =  {s = noun.s ++".count" +
+ numeraladverb.s  ++ number.s };
TheNounExists noun  =  {s=  noun.s ++ "/= VOID" };
```

```
ANounExists noun  =  {s=  noun.s ++ "/= VOID" };
NoNounExists noun  =  {s=  noun.s ++ "= VOID" };
TheNounIsNoun noun1 noun2 = {s=  noun1.s ++ "member_of" ++ noun2.s };
TheNounIsNotNoun noun1 noun2 = {s=  noun1.s ++ "not member_of" ++ noun2.s
};
ANounIsNoun noun1 noun2 = {s=  noun1.s ++ "member_of" ++ noun2.s };
ANounIsNotNoun noun1 noun2 = {s=  noun1.s ++ "not member_of" ++
noun2.s };
ItIsVerb verbphrase = {s =  verbphrase.s };
NounIsVerb noun verbphrase ={s=  noun.s ++ "." ++ verbphrase.s};
ItIsVerbAdv verbphrase adverb = {s= verbphrase.s ++ "_" ++ adverb.s};
NounIsVerbAdv noun verbphrase adverb = {s= noun.s ++ "." ++ verbphrase.s
++ "_" ++ adverb.s};
ItIsNotVerb verbphrase = {s = "not" ++  verbphrase.s };
NounIsNotVerb noun verbphrase ={s=  "not" ++ noun.s ++ "." ++
verbphrase.s};
ItIsNotVerbAdv verbphrase adverb = {s= "not" ++ verbphrase.s ++ "_" ++
adverb.s};
NounIsNotVerbAdv noun verbphrase adverb = {s= "not" ++ noun.s ++ "." ++
verbphrase.s ++ "_" ++ adverb.s};

--Noun Phrases
PronounNounPhrase  pronoun noun = {s = noun.s ++ pronoun.s};

--VerbPhrases
ProgressiveVerbPhrase v = mkUtt (mkImp v);

--AdjectivePhrases
AdjectivePhrase a = mkUtt(mkVP(mkAP a));

--Base Classes
Object n = mkUtt (mkNP(mkCN n));
Action v = mkUtt(mkVP v);
TwoPlaceAction v2 = mkUtt(reflexiveVP v2);
ModifyAction adv = mkUtt(mkVP adv) ;

--Interogatives
What = {s = "VALUE "} ;
Who = {s = "VALUE "} ;
Which = {s = "VALUE "};
HowMany = {s = "INTEGER"} ;

--Pronouns
Its = {s = ":"} ;
You = {s = ":"} ;
He = {s = ":"} ;
Pron_I = {s = ":"} ;
She={s = ":"} ;
They = {s = ":"} ;
We ={s = ":"} ;

--Digits
Numbers d = mkUtt (mkCard d) ;

--Numeral Adverbs
AtMost = {s = "<="};
AtLeast = {s = ">="};
MoreThan = {s = ">"};
LessThan = {s = "<"};

--Quantifiers
The = {s = "the"};
A_Quant = {s = "a"};
```

```
That = {s = "that"};
This = {s = "this"};
No = {s = "no"};

--Puntuation
FullStop = {s = ""} ;
Exclamation = {s = ""};
QuestionMark = {s = ""};

--Complement_Verb
Can = {s = "can"} ;
Must = {s = "must"} ;

--Conjunctions
ConjunctionOr = {s = "or"} ;
ConjunctionAnd = {s = "and"} ;
ConjunctionIf = {s = "it_holds"};
ConjunctionXor = {s = "xor"};
ConjunctionSuchThat = {s = "such that"}

}
```