oxyGen: A Language Independent Linearization Engine

Nizar Habash

Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20740
phone: +1 (301) 405-6768
fax: +1 (301) 314-9658
habash@umiacs.umd.edu
http://umiacs.umd.edu/labs/CLIP

Abstract

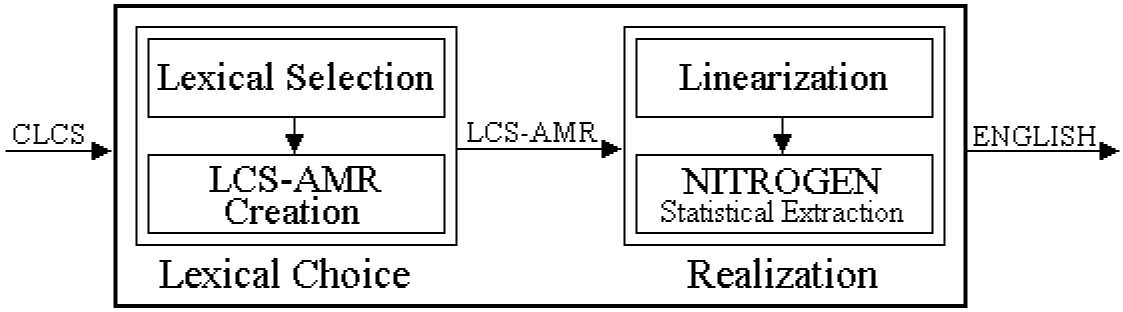
This paper describes a language independent linearization engine, oxyGen. This system compiles target language grammars into programs that take feature graphs as inputs and generate word lattices that can be passed along to the statistical extraction module of the generation system Nitrogen. The grammars are written using a flexible and powerful language, oxyL, that has the power of a programming language but focuses on natural language realization. This engine have been used successfully in creating an English linearization program that is currently used as part of a Chinese-English machine translation system.

1 Introduction

This paper describes a language independent realization engine, oxyGen. This system compiles linearization grammars into programs that run independently of the grammar and the compilation engine. The grammars are written in oxyL, a powerful and flexible natural language grammar description language. The syntax of oxyL is described in the paper. Currently, the input to the compiled grammar is a feature graph and the output is a word lattice to be fed into the statistical extraction module of the generation engine Nitrogen (Langkilde and Knight 1998a, 1998b 1998c).

2 Research context

The work described in this paper has been developed as part of an interlingual Chinese-English Machine Translation system at the University of Maryland College Park. (Dorr et al. 1998), (Traum and Habash 2000). The focus of this paper is only on the Linearization sub-module of the realization module in the generation component of the MT system. The realization module discussed is Nitrogen, a hybrid rule-based/statistical realization engine (Langkilde and Knight 1998a, 1998b 1998c). The system consists of two components, Linearization and Statistical Extraction (Graph1). First, a Feature Graph (FG) representation of the sentence to realize is converted into a word lattice of possible word sequence renderings, i.e. linearized. Then, the uni and bigram statistics are used to determine the most probable set of paths along the word lattice.



(Graph 1)

The particular form of FGs exemplified in this paper is a modified version of Nitrogen's Abstract Meaning Representation for our MT system's purposes(Dorr et. al 1998). AMRs are labeled directed feature graphs written using the syntax for the Penman Sentence Plan Language (Penman 1989):

```
<AMR>   ::= (<label> {<role> <value>}+ )
<value> ::= <AMR> || <terminal>
```

(BNF 1)

Every node in an AMR has a label and one or more role-value pairs. Roles, i.e. features, are marked by a colon prefix except for the default role *:instance* which can be represented as a forward slash `/`. Values can be meaning carrying terminal tokens or AMR nodes. Meaning carrying tokens can be semantic concepts such as `|china|` or `|love|`, syntactic categories such as `N` or `V`, or plain surface text strings such as `"Once upon a time"`. The roles and concepts of AMRs are a mix of syntactic and semantic significance: there are `:LCS-AG` (lexical conceptual structure agent) and syntactic categories such as `ADV`. The following is an example AMR for *The United States unilaterally reduced the China textile export quota*:

```
(a1 / |reduce|
    :CAT V
    :LCS-AG (a2 / |united states| :CAT N)
    :LCS-TH (a3 / |quota|
                :CAT N
                :LCS-MOD-THING (a4 / |china| :CAT N)
                :LCS-MOD-THING (a5 / |textile| :CAT N)
                :LCS-MOD-THING (a6 / |export|  :CAT N))
                :LCS-MOD-MANNER (a8 / |unilaterally| :CAT ADV))
```

(AMR 1)

In this example, `(a4 / |united states| :CAT N)`, is the agent of the concept `|reduce|`. And similarly, `N` is the category of the concept `|united states|` . The basic role `:instance` or `/` is always present in a non ambiguous AMR. An ambiguous AMR, i.e., a conglomeration of different AMRs has one or more role-value pairs using the special role `:OR`. For example, an variant of the above AMR in which the root concept is three way ambiguous would look as follows at the top node

```
(# :OR (# / |reduce| . . . )
   :OR (# / |cut| . . . )
   :OR (# / |decrease| . . . ))
```

(AMR 2)

Since such ambiguity can occur anywhere in an AMR, it presents a challenge to writing simple linearization rules whose application is conditional upon specific AMR role combinations at different depths. This issue is addressed later in this paper.

The output of the Linearization module is a word lattice of possible word sequence renderings. It includes ambiguous paths resulting from under-specified features, such as definiteness, and undetermined relative word orders, such as that of modifiers. The following is a possible word lattice corresponding to (AMR 1).

```
(SEQ (WRD "*start-sentence*" BOS)
     (WRD "united states" NOUN)
     (WRD "unilaterally" ADJ)
     (WRD "reduced" VERB)
     (OR (WRD "the" ART) (WRD "a" ART) (WRD "an" ART))
     (WRD "china" ADJ)
     (OR (SEQ (WRD "export" ADJ) (WRD "textile" ADJ))
         (SEQ (WRD "textile" ADJ) (WRD "export" ADJ)))
     (WRD "quota" NOUN) (WRD "*end-sentence*" EOS))
```

(WL 1)

Then the statistical extraction module evaluates the different paths represented in the word lattice using uni and bigram statistics and returns the following:

united states unilaterally reduced the china textile export quota . [LENGTH 10, SCORE -41.657174]
united states unilaterally reduced a china textile export quota . [LENGTH 10, SCORE -42.817673]
united states unilaterally reduced the china export textile quota . [LENGTH 10, SCORE -42.867434]
united states unilaterally reduced a china export textile quota . [LENGTH 10, SCORE -44.027932]
united states unilaterally reduced an china textile export quota . [LENGTH 10, SCORE -44.746711]
united states unilaterally reduced an china export textile quota . [LENGTH 10, SCORE -45.956971]

The focus of this paper is on the implementation techniques of the Linearization module of the realization system.

3 Motivation

The Linearization module is basically an implementation of a set of rules, a grammar, that governs the relative word ordering (syntax) and word form (morphology) of a target language. A linearization grammar can be implemented declaratively or procedurally. In the declarative approach, the system contains a grammar description formalism and a linearization engine that interprets the grammar on-line and applies its rules to the input sentence representation. The advantages of this approach are reusability, easy extendibility and language independence. Its main drawback is slow speed. Nitrogen's Linearization module is an example of this approach. It provides rules to decompose an AMR and order the results linearly. The Nitrogen grammar description formalism uses a recasting mechanism to transform AMRs into other AMRs. Besides the slowness inherited from the paradigm of its implementation, Nitrogen's grammar formalism is limited and inflexible:

- Rule application is conditional upon equality of concepts or existence of roles at the top level of an AMR only. This makes it impossible to write a single rule that is conditioned upon a combination of features at different levels. Cascading features is a solution to this problem that only increases the size of the grammar and aggravates the speed problem.

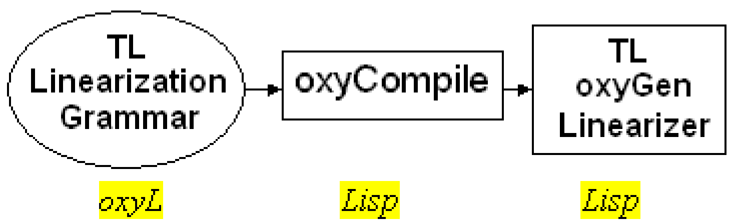
- Recasting operations are limited to adding feature-value pairs and introducing new nodes. Implementing a thematic hierarchy ordering in which thematic roles such as *agent* and *theme* are recast as syntactic roles such as *subject* and *object* cannot be implemented in a single recast operation. Again, cascading of features is the only way to do this. An implementation of thematic hierarchies using cascading features is discussed in (Dorr et al. 1998).

- There is no mechanism to perform range-unbounded or computationally complex transformations. For example, number formatting is a transformation problem that requires access to functions such as multiplication and addition which are not available to the grammar. One instance of this problem appeared in our system when translating Chinese numbers represented as multiples units of 10,000. For example, 80,000 is the concept |8| modified by the concept |10,000|. Multiplying Chinese number concepts and formatting them into English number sequences was necessary and is impossible to do using recasting without enumerating all combinations!

The procedural approach to Linearization grammars uses a programming language to implement the rules of the grammar. The main advantages of this approach are flexibility, power and speed. Having access to the full computing power of a programming language opens a lot of possibilities for efficient implementation. It also frees the linearizer's designer from the restrictions of a limited declarative grammar by providing access to the operating system, databases, the web, etc. However, a major disadvantage of this approach is that the linguistic knowledge is coupled with the programming code. This hard-coding of grammar rules makes the system rather redundant, difficult to understand and debug, non-reusable and language specific.

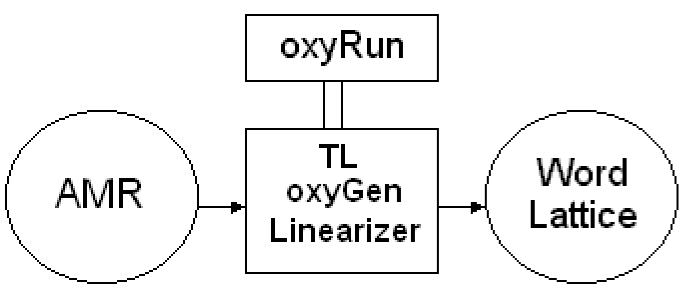
4 oxyGen

The oxyGen approach to implementing the Linearization module is a hybrid implementation between the declarative and procedural paradigms. oxyGen uses a linearization grammar description language to write declarative grammar rules which are then compiled into a programming language for efficient performance. oxyGen contains three elements: a linearization grammar description language (oxyL), an oxyL to Lisp compiler (oxyCompile) and a run-time support library (oxyRun). Target language linearization grammars written in oxyL are compiled off-line into oxyGen Linearizers using oxyCompile (Graph 2).



(Graph 2)

oxyGen Linearizers are Lisp programs that require the oxyRun library of basic functions in order to execute (Graph 3). They take AMRs as input and create word lattices that are passed on to some Statistical Extraction unit.



(Graph 3)

This implementation maximizes the advantages and minimizes the disadvantages inherent in the declarative and procedural paradigms: The separation between the linearization engine (oxyCompile and oxyRun) and the linearization grammar (oxyL) combines in one system the best of two worlds: the simplicity and focus of a declarative grammar with the power and efficiency of a procedural implementation. It also provides language independence and reusability since needs of the target language are only addressed in its specific oxyL grammar. Secondly, The run-time separation between language-specific code (compiled oxyL file – oxyGen Linearizer) and language-independent code (oxyRun) allows for efficient resource-sharing implementation especially when running multiple linearizers for different languages at the same time as in multilingual generation. Finally, oxyGen's linearization grammar description language, oxyL, is as powerful as a regular programming language but with the focus on linearization needs. This is accomplished through providing powerful linearization mechanisms for the most common needs of a linearization grammar and also by allowing embedding of code in a standard programming language (Lisp) to allow for efficient implementation of the more language specific realization problems (e.g., Chinese number formatting). oxyL linearization grammars are also simple, clear, concise and easily extendible. An example of the simplicity of oxyL grammars is that redundant issues such as the handling of `:OR` ambiguities are hidden from the linearization grammar designer and are treated only in the compiler and support library. The following section describes oxyL's syntax and the mechanism of application of oxyL rules.

5 oxyL

In many ways, it is similar to the language Nitrogen grammars are written in; however, it has several special features that makes it more powerful. First, oxyL linearization rules can be conditionally applied using general Boolean expressions and embedded if-then-else control flow structures which allows for powerful and compact linearization grammars. Second, oxyL provides accessibility functions that can return the value of any descendant of the AMR. Contrast these two features with Nitrogen's grammar's conditions of application which are flat if-then structures and use only equality of roles or role-value combinations at the top level of the AMR. Third, oxyL provides recasting mechanisms that are more powerful than Nitrogen's. For example, a thematic hierarchy recast in oxyL is implemented in a single rule whereas it requires as many rules as the number of hierarchy slots in Nitrogen. Finally, oxyL can embed calls to lisp functions that can be included in the oxyL file. This feature provides oxyL linearization grammars with access to all the tools available to a programming language. The rest of this section will describe oxyL's syntax.

5.1 OxyL Basic Tokens

The function of different tokens in oxyL is marked through their form using a prefix symbol: variables are prefixed with a dollar sign (e.g. `$form`, `$tense`), role-names are prefixed with a colon (e.g. `:agent`, `:cat`) and functions are prefixed with an ampersand (e.g. `&eq`, `&ProperNameHash`). Some of oxyL's functions resemble Lisp functions (e.g. `&eq` and `eq`). However, their implementation is different in oxyGen since ambiguity has to be handled. So, `&eq` for example is aware of the existence of `:OR`ed AMRs in which matching one of the possible `@OR`s is enough to return true, whereas lisp `eq` is not.

In addition to general functions, oxyL has a special class of functions called *referential functions*. These functions, which are prefixed with an *at* sign (e.g. `@agent`, `@this`), are used to access values corresponding to specific roles of the current AMR. For example, `@LCS-AG` returns the value corresponding to the role `:LCS-AG`. If the current AMR is (AMR 1) in section 2, `@LCS-AG` returns `(a2 / |united_states| :cat n)`. The instance role, `/`, is returned using the special referential function `@inst`. A referential function can specify the path from the current AMR's root to any value under it by concatenating the references along such path. For instance, if the current AMR is (AMR 1), `@LCS-AG.CAT` returns `N`. If the current AMR contains multiple instances of the same role as in `:LCS-MOD-THING` in (AMR 1), the values are combined in a `:OR` structure. For example, if the current AMR is (AMR 1), `@LCS-TH.LCS-MOD-THING.INST` returns `(# :OR |china| :OR |textile| :OR |export|)`. Access to the full current AMR is provided through the self-referential function `@this`. For example, `@this.agent` is equal to `@agent`.

The last oxyL basic token type is Macros, which are prefixed with a circumflex (e.g. `^NP-NOM`). Macros are treated like variables except that while variables appear as is in the compiled grammar, macros are substituted in the compiler. The use of macros makes the grammar description more concise. For example, if a set of role-value pairs is very commonly used such as `(:Form NP :Case NOM)`, they can be referred to using a single macro, `^NP-NOM`..

5.2 oxyL File

An oxyL file contains the a set of declarations. Some are obligatory (marked below with an asterisk) for proper compilation into Lisp code. Others introduce symbols that could be used eventually in the grammar rules such as global variable or special lisp functions. The following is a list of these declarations:

Declaration	**Function**	**Example**
`:Language`*	Name of generated grammar	`:Language "English"`
`:SupportCode`	User-defined Lisp functions	`:SupportCode ( <lisp code> )`
`:SupportInclude`	Lisp file to load at runtime	`:SupportInclude "support.lisp"`
`:CLASS`	Defines a class of roles	`:CLASS :THETA (:AG :TH :GOAL :SRC)`
`:GLOBAL`	Declares a global variable	`:GLOBAL $`
`:MACRO`	Declares a macro	`:MACRO ^NP-ACC (:CAT N :CASE ACC)`
`:MORPH`*	Defines the morphological generation function	`:SupportInclude "EnglMorph.lisp"` `:MORPH (&Morph @word @morphemes)`
`:RULES`*	Defines the grammar	`:RULES <Linearization-Grammar>`

*Obligatory declarations

All Lisp supporting code introduced through `:SupportInclude` or `:SupportCode` need all interfacing functions to be prefixed with an & like oxyL general functions.

A `:Class` is a "super" role. It is a cover symbol that can be used to reference different classes of roles. For example, `:THETA` can be defined to refer to all thematic roles and `:MOD` can refer to all types of modifiers. Once defined, referential functions can be used for it. Internally, class roles and regular roles are processed differently but that is hidden from the user.

The syntax of the oxyL grammar rules declared using `:RULES` is described in the next section.

5.3 oxyL Target Language Grammar

```
<GRAMMAR> ::= <RULE>+
<RULE>    ::= ([== <ASSIGN>]
                {?? <COND>
                 -> <RESULT>}*
                [-> <RESULT>] )
<ASSIGN>  ::= ((<variable> <value>)+)
<COND>    ::= <Boolean Expression>
<RESULT>  ::= <RULE> || <SEQUENCE>
<SEQUENCE>::= ({<AMR>||<RECAST>}+}) || (OR <SEQUENCE> <SEQUENCE>+)
<RECAST>  ::= (<AMR> {<RECAST-OP> <RECAST-OP-ARGS>}+)
```

(BNF 2)

(BNF 2) describes the syntax of an oxyL grammar. A grammar consists of a set of ordered rules each of which is considered for application over the current AMR. Each rule has an optional assignment section, introduced with `==`, in which local variables are defined. The second part of a rule is an optional condition and result pair that can be repeated multiple times. Conditions are introduced with `??` and results with `->`. And finally an optional result that is treated as the default if all conditions fail. A result can be a rule in itself with all of the described portions or it can be a sequence of AMRs or AMR-returning tokens such as variables or functions. The ability to embed rules within rules and declare local variable with deep scope allows users to limit the size of the grammar and increase the speed of its application logarithmically. The linear order of AMRs in the result specifies the linear order of the surface forms corresponding to these AMRs. The grammar is run recursively over each one of the different AMRs. This process continues until terminal values, i.e. surface forms, are reached. Consider the following oversimplified rule:

```
(== (($form @form))
 ?? (&eq $form S)
 -> (?? (&eq @voice Passive)
     -> (@object (&passivize @inst) “by” @subject)
     -> (@subject @inst @object)))
```

(Rule 1)

Initially, this rule takes the value of the role `:form` in the current AMR and assigns it to the variable `$form`. In the case the value of `$form` equals `S`, a second check on the voice of the current AMR is done. If the voice is passive, the passive word order is realized. Otherwise, the active voice word order is realized. The grammar is then called recursively over the AMRs of `@subject`, `@object` and `@inst`. The function `&passivize` takes the AMR of `@inst` as input and can return a passive verb AMR that gets processed by the grammar or a terminal word sequence.

In addition to AMRs, a linearization sequence can contain AMR recast operations. A recast operation is made out of an AMR followed by one or more pairs of recast operator and recast operator arguments. Recast operations modify AMRs before they are recursively run through the grammar. The recast mechanism is very useful in restructuring the current AMR or any of its components. For example, the ++ recast operator adds role-value pairs to an AMR. This is useful in cases such as adding case marking roles on the subject and object AMRs where such case markers are not specified in the original, more semantic, representation. (Rule 1) described in the previous section could be modified to specify case as follows:

```
(== (($form @form))
 ?? (&eq $form S)
 -> (?? (&eq @voice Passive)
     -> ((@object ++ (:case nom)) (&passivize @inst)
         “by” (@subject ++ (:case gen)))
     -> ((@subject ++ (:case nom)) @inst (@object ++ (:case acc)))))
```

(Rule 2)

The following is a list of oxyL recast operators and their usage formalism and functionality:

Name	OP	Usage	Function
Add	++	(AMR ++ :role$_0$ value$_0$:role$_1$:value$_1$ …)	Add role-value pairs to AMR
Delete	--	(AMR -- (:role$_0$:role$_1$ …)	Remove all role$_n$-value pairs
Replace	&&	(AMR && (:role$_0$ value$_0$:role$_1$ value$_1$ …)	Replace values of `:role`$_n$
Simple	<<	(AMR << (:new / :old$_0$:old$_1$ …))*	Rename all existing `:old`$_n$ as

Recast			:new
Hierarchy Recast	<!	(AMR <! (:new$_0$:new$_1$ … / :old$_0$:old$_1$ …))[*]	Hierarchically rename available :old$_n$ as :new$_n$
Morph	+-	(AMR +- morpheme)	Invoke the morphological generation function on the AMR if it is a value, or on its instance

[*] The use of / here is different from its role as a shorthand for :inst.

6 Evaluation

In this section, oxyGen is evaluated based on Speed of performance, Size of grammar, Expressiveness of the grammar description language, Reusability and Readability/Writability. The evaluation context is provided by comparing an oxyGen Linearization grammar for English to two other implementations, one procedural (using Lisp) and one declarative (using Nitrogen Linearization module). Three comparable linearization grammars are used to calculate speed and size. All three were actually implemented at different stages of development in the Chinese-English MT system mentioned in section 2.

Speed: Two tests were performed. The first test uses a small corpus of 100 simple AMRs of an average of 17 particles (label, role or terminal value) per AMR. The second test uses a corpus of 213 AMRs representing translated Chinese news article sentences. These averaged 463 nodes and 7 :ORs per AMR. The following table contains the times spent on average per system in milliseconds. The Lisp implementation is the fastest followed by oxyGen. Nitrogen lags behind considerably.

	Procedural (Lisp)	**oxyGen**	**Declarative (Nitrogen)**
Test 1	3.84 ms	37.67 ms	630.56 ms
Test 2	11.50 ms	278.45 ms	17028.00 ms

Size: The following table contains the size of code in *lines of code* of the three implementations. The oxyGen code size is the sum of the oxyL grammar (192 *loc*)and the Lisp English support functions (62 *loc*). The Nitrogen code size is the sum of Nitrogen's English grammar (1655 *loc*) and an extension grammar to make it compatible with our system (375 *loc*). Clearly, oxyGen performs the best.

	Procedural (Lisp)	**oxyGen**	**Declarative (Nitrogen)**
Size	763 *loc*	252 *loc*	2030 *loc*

Expressiveness: Lisp and oxyGen are equally expressive in the sense of their accessibility to computational tools as described earlier. Whereas Nitrogen falls behind.

Reusability: Both Nitrogen and oxyGen are language independent, an advantage over any procedural implementation.

Readability/Writability: All three approaches need a certain amount of training. However, oxyGen's simple syntax is an advantage over lisp (for linearization purposes, that is). Its compact powerful rules are an advantage over Nitrogen's simple rule mechanisms.

Overall: oxyGen has the best overall performance of the three systems.

	Procedural (Lisp)	oxyGen	Declarative (Nitrogen)
Speed	+	0	-
Size	0	+	-
Expressiveness	+	+	-
Reusability	-	+	+
Readability/ Writability	-	+	-

7 Future Work

This project is still in its initial phases and more work is still needed. As far as the oxyL language definition and the runtime library support oxyRun, more tools and function libraries are needed such as meta-level functions that return information about the current AMR, e.g., its role under its parent AMR, the number of theta roles or modifiers in it, its total depth, etc. Such information can be very helpful for sentence planning purposes. Other function libraries can be created to handle generation of specific domains such as time/date formatting, newspaper titles, etc. As for oxyCompile, more debugging tools and error handling routines are needed to make the system more robust and user-friendly. Independently of the engine itself, more oxyL grammars for other languages are needed to test the systems extendibility. Arabic and Spanish generation are especially under consideration since we currently have all the needed resources given our LCS-Based Machine Translation paradigm.

A possible extension to the oxyGen suite could be to allow different input formats yet still using the same common engine. Other possible input formats besides Penman sentence planning include NMSU F-Structures, XML and CycL. Such an endeavor would require a higher level of separation between the compiler and the input format which has to be specified to the compiler through some input language definition grammar.

Another area for possible future work is to use of oxyGen as part of NLP applications besides machine translation such as text summarization.

8 Conclusion

I have presented a language independent linearization engine that compiles target language grammars into programs that take abstract meaning representations as input and generate word lattice that can be passed along to a statistical extraction module. The grammars are written using a flexible and powerful language, oxyL, that has the power of a programming language but focuses on natural language realization. This approach was evaluated to be more efficient than other purely declarative or procedural approaches.

9 Acknowledgements

This work has been supported by NSA Contract MDA904-96-C-1250 and NSF PFF/PECASE Award IRI-9629108. I would like to thank members of the CLIP lab for helpful conversations and advice and especially Bonnie Dorr, Philip Resnik, David Traum and Amy Weinberg. I would also like to thank Kevin Knight and Irene Langkilde for making the Nitrogen system available and help with understanding the Nitrogen grammar formalism.

10 References

Dorr, Bonnie and Nizar Habash and David Traum. A Thematic Hierarchy for Efficient Generation from Lexical Conceptual Structure. In *Proceedings of the third Conference of the Association for Machine Translation in the Americas (AMTA)*, pages 333--343, Langhorne, PA. 1998.

Knight, Kevin and Vasileios Hatzivassiloglou. Two-Level, Many-Paths Generation. In *Proceedings of ACL-91*, pages 143--151, 1991.

Langkilde, Irene and Kevin Knight. *Generating Word Lattices from Abstract Meaning Representation*. Technical Report, Information Science Institute, University of Southern California, 1998a.

Langkilde, Irene and Kevin Knight. Generation that Exploits Corpus-Based Statistical Knowledge. *In Proceedings of COLING-ACL '98*, pages 704--710, 1998b.

Langkilde, Irene and Kevin Knight. The Practical Value of N-Grams in Generation. In *International Natural Language Generation Workshop*, 1998c.

Penman. *The Penman Documentation*. Technical report, USC/Information Sciences Institute. 1989.

Traum, David and Nizar Habash. Generation from Lexical Conceptual Structures. *In Proceedings of Workshop on Applied Interlinguas, NAACL/ANLP2000*, Seattle Washington, 2000.