A Reference Manual to the Linearization Engine oxyGen version 1.6

Nizar Habash
University of Maryland
Institute for Advanced Computer Studies

September 13, 2001

Contents

Chapter 1

oxyGen

1.1 Introduction

This is a manual for the language independent linearization engine, oxyGen. This system has been developed as part of the Machine Translation (MT) effort at the University of Maryland College Park [1, 8]. oxyGen has been used as an integral part of the Natural Language Generation (NLG) component of an interlingual Chinese-English MT project and a Spanish-English MT project. It has also been used to generate simple Spanish and Chinese sentences on a large scale of coverage [3]. Natural Language Generation is interested in taking non-linguistic representations as input and converting them into natural language output. NLG can be divided into two major distinct operations: Lexical Selection and Linearization. The former is concerned with selecting the correct natural language lexical item such as *eat* versus *devour* or *car* versus *vehicle*. The later is concerned with the relative positioning of lexical items on the surface such *man hit dog* versus *dog hit man* or *man dog hit*. oxyGen is an engine for developing programs to do the later operation: Linearization. The input to such programs is a labeled Feature Graph (FG) representation of a natural language sentence. The particular form of FGs used here is a modified version of Nitrogen's Abstract Meaning Representation (AMR) [5, 6]. AMRs are labeled directed feature graphs written using the syntax of the Penman Sentence Plan Language [4]. The output of the linearization programs developed using oxyGen is a word lattice, a compressed representation of the various possible generated sequences. See Figure 1.1.

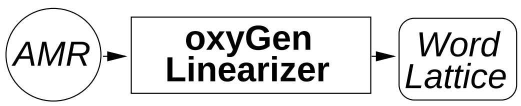

Figure 1.1: oxyGen Linearizer

1.2 Linearization

To exemplify the use of oxyGen and linearization in general, take the following input AMR:

(1)
```
(1 / |like|
   :POS Verb
   :Subject (2 / |man| :POS Noun)
   :Object (3 / |car| :POS Noun))
```

This AMR can be read as *there is a verb, like, and it has a subject, man, which is a noun and an object, car, which is also a noun.* In English, a proper word order would be *man like car* (or more fluently *the man likes the car*, but let's not worry about fluency for now). To specify that an SVO (subject verb object) order is desired in English (versus VSO or SOV), we need a linearization rule such as the following:

(2)
```
(?? (&eq @pos Verb) -> (@subject @/ @object)
 -> (@/))
```

This rule is written using oxyL (oxyGen Language), a flexible and powerful language that has the power of a programming language but focuses on natural language realization. This rule can be read as *if the part of speech (POS) of the current AMR is Verb, then linearize the subject AMR followed by the word instance followed by the object AMR; otherwise linearize the word instance by itself.* This is a very simple grammar that needs more extensions to handle real input with different phrase structures and parts of speech. But a *real* AMR is also complex on a different dimension: Ambiguity. Let's assume the input AMR is a result of a lexical selection process for the same sentence in (1) from a language that doesn't specify number (singular versus plural) and its word for *like* is ambiguous in that it covers the concepts of *desire* and *love*. This AMR could look as follows:

(3)
```
(0 :OR (1 / (*or* |like| |likes|)
          :POS Verb
          :Subject (2 / (*or* |man| |men|) :POS Noun)
          :Object (3 / (*or* |car| |cars|) :POS Noun))
   :OR (4 / (*or* |desire| |desires|)
          :POS Verb
          :Subject (5 / (*or* |man| |men|) :POS Noun)
          :Object (6 / (*or* |car| |cars|) :POS Noun))
   :OR (7 / (*or* |love| |loves|)
          :POS Verb
          :Subject (8 / (*or* |man| |men|) :POS Noun)
          :Object (9 / (*or* |car| |cars|) :POS Noun)))
```

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. However, the beauty of oxyGen is that it allows hiding the ambiguity of the input from the grammar description so that both AMRs (1 and 3) can be linearized using the same grammar rule in (2). Of course, the ambiguity of (3) will lead to a large set of sequences:

(4)

`man like car`	`man desire car`	`man love car`
`man like cars`	`man desire cars`	`man love cars`
`man likes car`	`man desires car`	`man loves car`
`man likes cars`	`man desires cars`	`man loves cars`
`men like car`	`men desire car`	`men love car`
`men like cars`	`men desire cars`	`men love cars`
`men likes car`	`men desires car`	`men loves car`
`men likes cars`	`men desires cars`	`men loves cars`

A statistical extraction module can be used to rank the different sequences using uni and bigram statistics or other language models. The statistical extraction component of Nitrogen [5, 6] is one such module.

In addition to hiding ambiguity from the grammars, oxyGen provides, through oxyL, a great power to the grammar writers by providing complex tools designed with natural language linearization in mind. oxyGen can also be extended and modified easily via second and third-party code.

1.3 oxyGen: A Hybrid System

oxyGen compiles target language grammars written in oxyL into compilable Lisp programs that take AMRs as inputs and generate word lattices that can be passed along to be ranked by some language model. This approach to linearization implementation is a hybrid between the declarative and procedural paradigms. oxyGen uses a linearization grammar description language (oxyL) to write declarative grammar rules which are then compiled into a programming language (Lisp) for efficient performance. This hybrid approach allows oxyGen to maximize the advantages and minimize the disadvantages of a pure procedural implementation (in Lisp or C) or a pure declarative implementation (in Nitrogen grammar). oxyGen contains three main 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 (Figure 1.2).

oxyGen linearizers are Lisp programs that require the oxyRun library of basic functions in order to execute (Figure 1.3). They take AMRs as input and create word lattices as output.

In addition to the oxyCompile and oxyRun components, there are currently two additional components oxyLin, a simple converter from word lattices to

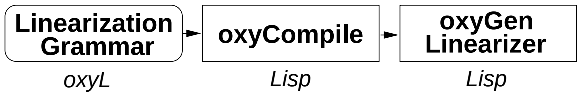

Figure 1.2: oxyGen Compilation Step

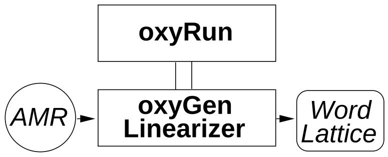

Figure 1.3: oxyGen Runtime Step

surface sequences, and oxyDebug, a support code for debugging the compiled linearization grammars. The specifications of all these components are in Chapter 4.

A more detailed discussion of the motivation and advantages of oxyGen is presented in [2]. There is also an evaluation of oxyGen 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). The three comparable linearization grammars were used to calculate speed and size. Overall, Oxygen had the highest number of advantages and its only disadvantage, speed, ranked second to the Lisp implementation (see Table 1.1). The version of oxyGen described in this manual is a more efficient implementation of Oxygen than the one evaluated in [2]. A second evaluation for a larger English grammar in oxyGen and Lisp showed Lisp is still faster than oxyGen. However the gap in speed between the Lisp and Oxygen implementations shrunk from Oxygen being 24 times slower than Lisp in [2] to only 1.5 times.

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

Table 1.1: Oxygen Evaluation

Chapter 2

oxyL

oxyL (oxyGen Language) is the language used by oxyGen to write linearization grammars. It is a flexible and powerful language that has the power of a programming language but focuses on natural language realization. As a prelude to describing the syntax of oxyL, we will describe the form of the structures oxyL commands are applied to, Abstract Meaning Representations. Then, we will discuss oxyL's basic tokens followed by the syntax of an oxyL file and oxyL rules and functions.

2.1 Abstract Meaning Representation

Abstract Meaning Representations (AMR) are labeled directed feature graphs written using the syntax of the Penman Sentence Plan Language [4]:

(5)

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

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, `:inst` (*instance*), which can be represented as a forward slash `/`. Values may be meaning bearing terminal tokens or AMR nodes. These terminal tokens can be semantic concepts such as `|china|` or `|love|`, syntactic categories such as `N` or `V`, plain surface text strings such as `"China"`, or a list of any of them headed by the special token `*or*` such as `(*or* man men)`. Except for a small number of reserved tokens used by oxyGen, most of the AMR tokens are user and application-defined. The only requirement is consistency between the AMRs and the oxyL grammars to linearize them. The roles and concepts of an AMR can be a mix of syntactic and semantic significance: thematic roles such as `:Agent` and `:Theme` and syntactic categories such as `:Subject` and `ADV`. The following is an example of a basic AMR for the sentence *The United States* *unilaterally reduced the China textile export quota* :

(6)
```
(1 / |reduce|
   :CAT V
   :Subject (2 / |united states| :CAT N)
   :Object (3 / |quota|
              :CAT N
              :MOD (4 / |china| :CAT N)
              :MOD (5 / |textile| :CAT Adj)
              :MOD (6 / |export|  :CAT Adj))
   :Manner (8 / |unilaterally| :CAT ADV))
```

In this example, `(a2 / |united states| :CAT N)` is the subject of the concept `|reduce|`. And similarly, N is the category of the concept `|united states|`. The basic role `:inst` or `/` is always present in a basic AMR.

However there are two other types of AMRs, that are *instance-less*: OR-AMR and AND-AMR. The first is a disjunction of basic AMRs, whereas the second is a conjunction of basic AMRs. Both are constructed using multiple copies of the same special role (`:OR` or `:AND`). An OR-AMR express lexical ambiguity, i.e., which structure to chose among many. For example, a variant of the above AMR in which the root concept is three way ambiguous would look as follows at the top node:

(7)
```
(# :OR (# / |reduce| . . . )
   :OR (# / |cut| . . . )
   :OR (# / |decrease| . . . ))
```

An AND-AMR, on the other hand, expresses linearization ambiguity, i.e., how to order the AMRs on the surface. The AMR in (6) expresses that ambiguity in the AMR for `quota`, which contains three identical roles (`:MOD`). That same AMR can be written using `:AND`s as follows:

(8)
```
(1 / |reduce|
   :CAT V
   :Subject (2 / |united states| :CAT N)
   :Object (3 / |quota|
              :CAT N
              :MOD (0 :AND (4 / |china| :CAT N)
                      :AND (5 / |textile| :CAT Adj)
                      :AND (6 / |export|  :CAT Adj))
   :Manner (8 / |unilaterally| :CAT ADV))
```

Handling `:AND`s and `:OR`s is done automatically and is hidden from the user-defined grammar. The ambiguity of an OR-AMR is passed on to the word lattice, while AMRs under `:AND`s are permuted to produce all possible linearizations.

There is one more special role, `:X-role`. It is used to express *role ambiguity*, i.e., a role can be of two or more role names. For example, The two AMR in (9) express the ambiguous sentence *John gave Paul a gift* and *John gave a gift to Paul.*

(9)
```
(# / |give|
   :subj |john|
   :obj |gift|
   :X-role (# / X
              :iObj |paul|
              :PP (# / |to|
                     :obj |paul|)))


(O :OR  (# / |give|
           :subj |john|
           :obj |gift|
           :iObj |paul|)

   :OR  (# / |give|
           :subj |john|
           :obj |gift|
           :PP (# / |to|
                  :obj |paul|)))
```

These AMRs are different in that the first AMR expresses the ambiguity locally as an ambiguous role (indirect object versus prepositional phrase), whereas the second AMR expresses the ambiguity at the top level as two different AMRs altogether. Handling `:X-role`s is done automatically and is hidden from the users. They are expanded to full fledged OR-AMRs.

2.1.1 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`).

In addition to general functions (built-in or user-defined), oxyL has a special class of functions called referential functions. These functions, which are prefixed with an `@` sign (e.g. `@goal`, `@this`), are used to access values corresponding to specific roles of the current AMR. For example, `@goal` returns the value corresponding to the role `:goal`. If the current AMR is (6) in section 2, `@subject` returns `(a2 / |united states| :cat n)`. The value of the instance role, `/`, is returned using the special referential functions `@/` or `@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 (6), `@subject.cat` returns `N`. If the current AMR contains multiple instances of the same role as in `:MOD` in 6, the values are returned in an AND-AMR. For example, if the current AMR is (6), `@object.mod.inst` returns `(# :AND |china| :AND |textile| :AND |export|)`. Access to the full current AMR is provided through the self-referential function `@this`. For example, `@this.subject` is equal to `@subject`.

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`.

2.2 oxyL File

An oxyL file contains a set of declarations (see Table 2.2). Some provide meta-level information such as `:Langauge` and `:Comment`, while others allow importing Lisp code such as `:Include` and `:Code`. The declarations `:Class`, `:Gloabl` and `:Macro` define variables for use by oxyCompile or oxyRun. The declaration `:Morph` allows the user to link the internal morphology handler to a specific user-define morphology function. And the declaration`:Debug` allows the user to turn on and off the debugging utility provided by oxyDebug. The declaration `:Recast` allows the user to define functions for modifying AMRs using a special class of oxyL functions called *recasts*. the declaration `:Rule` allows the user to define specific modules to handle different phenomena such as the different types of phrases. The most important and the only obligatory declaration is `MainRule` which defines the core of the grammar[1]. The next section will describe the structure of an oxyL rule. The details of the use of all other declarations is left to Chapter 4.

[1] In [2], a single declaration was available for the whole grammar :RULES. This has been since replaced with the declarations :Recast, :Rule and :MainRule which provide a higher level of modularity and efficiency.

Declaration	Function	Example
:Comment	Adds a Comment	:Comment "Hello World!"
:Language	Name of generated grammar	:Language "English"
:Include	Lisp file to load at runtime	:Include "EnglMorph.lisp"
:Code	User-defined Lisp functions	:Code (<lisp-code>)
:Class	Defines a class of roles	:Class :THETA (:AG :TH :GOAL)
:Global	Declares a global variable	:Global $MODE HTML
:Macro	Declares a macro	:Macro 3̂pS (:per 3 :num sing)
:Debug	Controls debugging mode	:Debug nil
:Morph	Defines the morphological generation function	:Morph (&morph @word @morphemes)
:Recast	Defines a recast	:Recast &PL (@this ++ (:num PL))
:Rule	Defines a rule	:Rule %S (-> (@S @V @O))
:MainRule	Defines the Main Function	:MainRule ((-> (do %XP)))

Table 2.1: oxyL Declarations

2.3 oxyL Rules

(10)

```
<RULE>    ::= ([== <ASSIGN>]
                ?? <COND>
                 -> <RESULT>*
                [-> <RESULT>] )
<ASSIGN>  ::= ((<variable> <value>)+)
<COND>    ::= <Boolean Expression>
<RESULT>  ::= <RULE> || <SEQUENCE> ||
              (DO <RULE-NAME> [<AMR>])

<SEQUENCE>::= ({<AMR>||<RECAST>}+)||
              (OR <SEQUENCE> <SEQUENCE>+) ||
              (LISP <lisp-code>) || (CODE <lisp-code>)

<RECAST>  ::= (<AMR> <RECAST-OP> <RECAST-OP-ARGS>+)
```

The above BNF describes the syntax of an oxyL rule. A 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 are introduced with ->. And finally an optional result is allowed as the default when all conditions fail. A result can be a rule in itself with all of the portions described above or it can be a sequence of AMRs or AMR-returning tokens such as variables or functions. It also can be a call to a user-defined rule using the special operator DO, which takes as an argument an optional AMR that defaults to `@this`. 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:

(11)
```
(== (($form @form))
 ?? (&eq $form S)
 -> (?? (&eq @voice Passive)
      -> (@object (&passivize @inst) "by" @subject)
      -> (@subject @inst @object)))
```

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 either 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. The rule described above, (11) could be modified to specify case as follows:

(12)
```
(== (($form @form))
 ?? (&eq $form S)
 -> (?? (&eq @voice Passive)
      -> ((@object ++ (:case nom)) (&passivize @inst)
           "by" (@subject ++ (:case gen)))
      -> ((@subject ++ (:case nom)) @inst
           (@object ++ (:case acc)))))
```

Table 2.3 provides a list of some oxyL recast operators with their usage formalism and functionality. Note that the use of `/` in recast operations is different from its role as a shorthand for `:inst`.

Multiple recast operators can be listed one after another in the same recast. A recast can also be embedded in another recast. For example, the recast `(@this && (:a (@a ++ (:b @b))) -- (:b))` moves the role `:b` and its value under `:b`'s sister `:a` using three different recast operations. Recasts can also

Name	Op	Usage
Add	++	(<AMR> ++ (<role> <value>+)) Add all <role>$_i$-<value>$_i$ pairs to AMR
Delete	--	(<AMR> -- (<role>+)) Remove all <role>$_i$-<value>$_i$ pairs
Replace	&&	(AMR && (<role> <value>+)) Replace all values of <role>$_i$
Simple Recast	<<	(AMR << (<new-role> / (<role>+))) Rename all existing <role>$_i$ as <new-role>
Hierarchy Recast	<!	(<AMR> <! ((<new-role>+) / (<role>+))) Hierarchically rename available <role>$_i$ as <new-role>$_i$

Table 2.2: oxyL Recast Operators

be accessed outside of results using the general recast function (& <recast>). This allows recasting an AMR any where before passing it to another function or Rule. For example, (do %V (& @this ++ (:punct "."))) adds a punctuation mark before passing the current AMR to the rule %V.

A result can also introduce alternative sequences using the special operator OR or make direct calls to Lisp functions using the special operator LISP (or CODE). The following example contains both OR and LISP operators:

(13)

```
(== (($name @name))
 -> (OR (LISP (FORMAT nil "~a loves me" $name))
        (LISP (FORMAT nil "~a hates me" $name))))
```

Note that calls to Lisp functions should return AMRs (including strings) for proper operation.

The special main rule declared with :MainRule consists of a list of regular rules. For example, the following main rule does one of two things every time it is accessed: terminate generation by realizing nothing if the instance of the current AMR is nil or *empty*, or pass the current AMR to the X-bar rule %XP.

(14)

```
:MainRule (
 ;; Nothing to generate
 (?? (&in @inst (|nil| |*empty*|))
  -> () )

 ;;Basic rule, go to XP
 (-> (do %XP)))
```

Chapter 3

Sample oxyL Grammar for English

This chapter presents a simple oxyL grammar that is used to linearize English syntactic dependency trees. The tokens used here are derived from the categories and relation in Dekang Lin's Minipar parser [7]. Sample input AMRs and outputs using oxyLin and Nitrogen's statistical extraction module are also presented.

3.1 The oxyL File

```
(
 :Language "Simple Inflected English Dependency"
 :Comment "This is an oxyGen grammar for English Generation"
 :Comment "version 1.0 / September 2001"
 :Include "nitrolin.lisp"

 :Debug nil

 :Global $V (V VBE V_I V_N V_P V_N_A V_N_C V_N_I V_N_N V_C_N V_N_N_A
             V_N_N_C V_N_N_P V_N_P_C V_N_N_P_A V_N_N_P_C V_N_N_P_N
             V_N_N_N XSAID SAID SAIDX)

 :Global $N (N NN NUM N_A N_C N_P)

 :Macro ^no-punct (:punct (a / |nil|))

 :Class :sub (:S :SUBJ)

 :Class :as (:AS-ARG :AS-HEAD :AS1)

 :Class :REST (:ABBREV :AGE :C :CN :DEST :FC :HEAD :I :INSIDE :LEX-DEP
               :LOCATION :POSS :SC :SPELLOUT :TITLE)
```

```
:Recast &whX (@this << (:wh / (:wha :whn :whp)))

:Recast &invX
 (@this << (:inv / (:INV-AUX :INV-BE :INV-HAVE)))

:Recast &AuxH
 (@this <! ((:aux1 :aux2 :aux3 :aux4) / (:aux :have :be :being)))

:Rule %DET (-> (@pre @rest @inst @post))

:Rule %N
 (-> (@conj-word @det @num @mod @lex-mod @gen @NN
      @inst
      @pnmod @person @appo-mod @appo @mod-post @comp1 @comp2
      @P @subcat (@vrel && ^no-punct) (@rel && ^no-punct)
      (@conj ++ (:conj-word "and"))))

:Rule %A
 (-> (@conj-word @rest @num @mod @lex-mod @amod @NN
      @inst
      @mod-post @P @subcat (@conj ++ (:conj-word "and"))))

:Rule %P
 (-> (@inst @rest @P-SPEC @Pcomp-N @PCOMP-C @subcat @punct))

:Rule %V
 (== (($to (0 / |to| :cond (&eq @tense inf))))
  ?? (&ex :inv)
  -> (@conj-word  @wh @inv @neg @sub @aux1 @aux2 @aux3 $to
      @inst
      @lex-mod @obj @obj2 @desc @pred @AS @AS2 @P @BY-SUBJ
      @guest @rest (@MOD && ^no-punct) @subcat @punct
      (@conj ++ (:conj-word "and")))

  -> (@conj-word @wh @sub @aux1 @neg @aux2 @aux3 @aux4 $to
      @inst
      @lex-mod @obj @obj2 @desc @pred @AS @AS2 @P @BY-SUBJ
      @guest @rest (@MOD && ^no-punct) @subcat @punct
      (@conj ++ (:conj-word "and"))))

:Rule %V-punct
 (?? (&ex :punct)
  -> (do %V)
  ?? (&ex :WH)
  -> (do %V (& @this ++ (:punct "?")))
  ?? (&eq @aspect IMPERATIVE)
  -> (do %V (& @this ++ (:punct "!")))
  -> (do %V (& @this ++ (:punct "."))))
```

```
:Rule %XP
 (== (($pos @pos))
  ?? (&in $pos $V) -> (do %V-punct (&auxH (&invX (&whx @this))))
  ?? (&in $pos $N) -> (do %N)
  ?? (&eq $pos DET) -> (do %DET)
  ?? (&eq $pos prep) -> (do %P)
  ?? (&eq $pos A) -> (do %A)
  -> (@inst))

:MainRule (
 ;; Nothing to generate
 (?? (&in @inst (|nil| |*trace*|))
  -> () )

 ;; Conditional generation technique
 (?? (&and (&ex :cond) (&null @cond))
  -> () )

 ;;Basic rule, go to XP
 (-> (do %XP)))
)
```

3.2 Input and Output

The follwing are four AMRs that were input to the linearization grammar described above. Each AMR is followed by oxyLin's output. the sentences in parentheses are Nitrogen's top choice.

```
(5 / |organized|
    :POS V
    :S (3 / |contest|
        :POS N
        :DET (1 / |the| :POS DET)
        :NN (2 / |writing| :POS N))
    :BE (4 / |was| )
    :BY-SUBJ (6 / |by|
        :POS PREP
        :PCOMP-N (8 / |office|
            :POS N
            :DET (7 / |the| :POS DET)
            :MOD-POST (9 / |of|
                :POS PREP
                :PCOMP-N (12 / |commissioner|
                    :POS N
                    :DET (10 / |the| :POS DET)
                    :MOD (11 / |official| :POS N))))))
```

(the writing contest was organized by the office of the official commissioner .)

```
(2 / |is|
    :POS VBE
    :WHA (1 / |how| :POS A)
    :PRED (6 / |system|
        :POS N
        :DET (3 / |the| :POS DET)
        :MOD (4 / |canadian| :POS A)
        :MOD (5 / |legal| :POS A)
        :VREL (7 / |constituted| :POS V_N_N)))
```

how is the legal canadian system constituted ?
(how is the canadian legal system constituted ?)

```
(5 / |courses|
    :POS N
    :DET (1 / |the| :POS DET)
    :MOD (2 / |following| :POS A)
    :MOD (3 / |general| :POS A)
    :NN (4 / |education| :POS N))
```

the general following education courses
(the following general education courses)

```
(3 / |mind|
    :POS N
    :LEX-MOD (1 / |peace| :POS *)
    :LEX-MOD (2 / |of| :POS *)
    :MOD-POST (4 / |of|
        :POS PREP
        :PCOMP-N (7 / |operation|
            :POS N
            :MOD (5 / |continuous| :POS A)
            :NN (6 / |system| :POS N))))
```

of peace mind of continuous system operation
(peace of mind of continuous system operation)

Chapter 4

oxyGen Reference

4.1 oxyGen Package

4.1.1 oxyGen Installation

The oxyGen package contains the following files:

`oxycompile.lisp`
`oxyrun.lisp`
`oxylin.lisp`
`oxydebug.lisp`
`oxyload.lisp`

The code files for the different oxyGen files. `oxyload.lisp` loads the files up.

`make-oxygen-core.sh`

A shell command for creating a dump of the oxygen system. The created dump file is called `oxygen.core`

`oxycompile`

A shell command for compiling oxyL files from the prompt. `oxycompile` needs oxygen.core to run properly.
Usage: `oxycompile <oxyl-filename> <out>`
The result of running oxyCompile is the creation of a `<out>.core` file and a shell command with the name `<out>`. The usage of the created shell command is:
`<out> <AMR-filename> <out-filename> <mode>`
where the optional argument `<mode>` is a keyword for the word lattice to surface module: *oxylin* or *nitrolin*. The default is *oxylin*.

oxypamr

A shell command for printing pretty AMRs. oxypamr needs oxygen.core to run properly.
Usage: oxypamr <amr-filename> <pretty-amr-filename>

nitrolin.lisp

Provides an interface between oxyGen and Nitrogen. This file needs to be included in an oxyL grammar if it is to be used. Activating nitroLin can be done by setting the <mode> argument to \verbnitrolin— in the appropriate functions.

4.1.2 oxyCompile

oxyCompile provides the functions necessary for compiling an oxyL grammar into a Lisp file. oxyCompile can be accessed directly from the shell using the shell command oxycompile described earlier.

(oxycompile <oxyl-grammar> <output-file>

Compiles <oxyl-grammar> into a Lisp program and outputs it to <output-file>. The optional <output-file> defaults to "oxyout.lisp".

(oxycompile-file <oxyl-file> <output-file>

Compiles the oxyL grammar in <oxyl-file> into a Lisp program and outputs it to <output-file>. The optional <output-file> defaults to "oxyout.lisp".

4.1.3 oxyRun

oxyRun provides functions necessary for proper operation of a compiled oxyL grammar.

(oxygen <AMR>)

Runs the oxyGen linearization grammar on an <AMR> and returns a word lattice.

(oxygen-file <AMR-file> <out-file> <mode>)

Runs the <AMR>s in <AMR-file> through the loaded oxyGen linearizer followed by the word lattice to surface module specified by the optional argument <mode> (*oxylin* or *nitrolin*). The output sentences are printed to <out-file>.

(&amrType <AMR>)

Returns the type of an AMR: word, wordlist, basicAmr, orAmr, andAmr, unknown

4.1.4 oxyLin

oxyLin provides functions for realizing a word lattice into strings. It is an alternative to using Nitrogen's Statistical Extraction module which realizes word lattices and assigns them uni/bigram scores.

`(oxylin <word-lattice> <stream>)`

Realizes `<word-lattice>` into strings and prints them to a file `<stream>`. `<stream>` is optional and is standard output by default.

`(check-size <word-lattice>)`

Returns the number of independent sequences in `<word-lattice>` without realizing it.

4.1.5 oxyDebug

oxyDebug provides functions for debugging a compiled oxyL grammar. It provides an output best comparable to Lisp's `trace`. Besides helping to figure out specific problems, the output of oxyDebug can be used to compare different grammars in terms of efficiency by comparing the number of calls they make to different functions. To use oxyDebug, an oxyL grammar should have the declaration `:Debug &true`. This forces oxyCompile to add calls to oxyDebug in the compiled grammar. Deactivating the debugging can be done by assigning the global variable `*oxydebug*` to `nil`.

`(oxydb-open <file>)`

Opens the file `<file>` and links it to the reserved output stream `*oxydb-stream*`. `<file>` is an optional argument that defaults to `"oxydb.out"`.

`(oxydb-close)`

Closes the reserved output stream `*oxydb-stream*`.

`(&oxydb <format> <var>)`

Allows users to send messages to `*oxydb-stream*` from inside an oxyL grammar. `<format>` is a string that can include Lisp's `format` instructions. `<var>` is an optional variable.

`(oxy-pamr <AMR> <stream>)`

Pretty prints `<AMR>` to the optional output stream `<stream>`. `<stream>` defaults to standard output.

`(oxy-pamr-file <in-file> <out-file>)`

Reads AMRs from `<in-file>` and pretty prints them to `<out-file>`.

4.2 Declarations

:COMMENT <string>

Includes the comment <string> in the compiled file. This declaration produces no action. A Lisp comment ";" can also be used in oxyL files.

Example

```
:COMMENT "This is a comment"
```

:LANGUAGE <string>

Specifies the name of the generated grammar. This declaration *currently* acts like :COMMENT.

Example

```
:LANGUAGE "English"
```

:GLOBAL <variable> <value>

Declares a global variable <variable> and sets its value to <value>.

Example

```
:Global $mode HTML
:Global $articles ("a" "an" "the" "")
```

:CLASS :<class> (<role>+)

Declares a class role :<class> to represent all the roles in (<role>+). A variable $<class> is created automatically for :<class>. The referential function @<class> returns a basicAMR if only one of the roles in (<role>+) exists; otherwise an andAMR of all existing roles is returned. In both cases, the matching role is remembered in the returned value as a value to the reserved role :role.

Example

:CLASS :THETA (:AGENT :THEME :SRC :GOAL :INSTRUMENT)

$THETA *returns* (:AGENT :THEME :SRC :GOAL :INSTRUMENT) *and it can be used in recasts such as* (@this -- $THETA) or (@this << (:new / $THETA))

@THETA of (0 / x :AGENT ag :x x :y y)

returns (0 / ag :ROLE :AGENT)

@THETA of (0 / x :AGENT ag :THEME th :x x :y y)

returns (0 :AND (0 / ag :ROLE :AGENT) :AND (0 / th :ROLE :THEME))

:MACRO ^<macro-name> <macro-body>

Declares a macro ^<macro-name> with the value <macro-body>. A macro acts like a global variable except that it is substituted by its value at compile time not run time. The use of macros makes the grammar description more concise.

Example

:MACRO ^NP-acc (:form NP :case acc)

(@this ++ ^NP-acc) *is compiled as* (@this ++ (:form NP :case acc))

`:CODE (<lisp-code>+)`

Adds Lisp code to the oxyL file. `:CODE` can be used to declare functions and variables. All user-defined functions must have the prefix `&` to run correctly. Similarly, all non-local variables must have the prefix `$`.

Example

```
:CODE ((setf $myvariable '(me me me))
       (defun &even (x) (evenp x))
       (defun &odd (x) (oddp x))
       (defun &concat (string1 string2)
          (format nil "~a~a"string1 string2))
```

`:INCLUDE <file-name>`

Loads the Lisp file named `<file-name>`. All user-defined functions and variables must have the appropriate prefixes run correctly. Example

```
:INCLUDE "wordnet-data.lisp"
:INCLUDE "brown-corpus-stats.fasl"
```

`:MORPH (<function> @word @morphemes)`

Defines the morphology handling function for the system to access. `<function>` is linked by oxyCompile to the internal morphology handler |(oxymorph @word @morphemes)—. `oxymorph` is fired by the morphology recast `+-`. `:MORPH` links the arguments `@word` and `@morphemes` to the input arguments of `function`.

Example

```
:MORPH (&english-morph @word @morphemes)
```

`:RECAST <recast-name> <recast-body>`

Allows the user to define a function `<recast-name>` for modifying AMRs using oxyL's built-in recasts. Recasts are well explained in Chapter 2.

Example

```
:Recast &move (@this && (:a (@a ++ (:b @b))) -- (:b))
```

moves the role `:b` *and its value under* `:b`*'s sister* `:a`*:*

```
(&move (0 / x :a (1 / a) :b (2 / b)))
```

returns `(0 / x :a (1 / a :b (2 / b)))`

`:RULE <rule-name> <rule-body>`

Defines a rule `<rule-name>` as `<rule-body>`. The definition of oxyL rules is well explained in Chapter 2. Rules can be named anything, but it is preferred that they have the prefix `%`. A rule can be activated with the special operator `DO` which takes an optional AMR as input. The default input is otherwise `@this`.

Example

```
:Rule %order (-> (@c @b @a @b @c))
(DO %order (0 / x :a a :b b :c c))
```

yields `c b a b c`

:MAINRULE (<rule>+)

Defines the main function in an oxyL grammar. This is the only obligatory declaration. The use of :MAINRULE is well described in Chapter 2.

:DEBUG <boolean>

Controls the inclusion of necessary code for debugging an oxyL grammar.

4.3 Built-in Functions

@<role-sequence>

Referential Function. Returns the value associated with the role at the end of the <role-sequence> of @this. <role-sequence> is constructed by listing the roles separated by periods and without the colon prefix.
Example
@this.subject.number returns the value of the role :number in the value of the role :subject under the current AMR.

(& <recast>)

General Recast Function. Returns the result of executing <recast>. This special function allows accessing oxyL built-in recasts as regular functions. This is useful for recasting an AMR before passing it as an argument to a rule or a function. This function cannot be used in a rule result.
Example
(do %NP (& (@Subject ++ (:case nom))))

(&ex <token> <AMR>)

Returns true if <token> exists in <AMR>. <token> can be a role or a word. <AMR> is optional and it defaults to @this.

(&nex <token> <AMR>)

Returns true if <token> doesn't exist in <AMR>. <token> can be a role or a word. <AMR> is optional and it defaults to @this.

(&eq {<value1> <value2>}+)

Returns true if all <value1>-<value2> pairs are equal.

(&neq {<value1> <value2>}+)

Returns true if all <varlue1>-<value2> pairs are *not* equal.

(&in <AMR> (<token>+))

If <AMR> is a *word*, &in returns true if <AMR> exists in (<token>+). If <AMR> is a *wordlist*, &in returns true if *any* word in <AMR> exists in (<token>+). If <AMR> is a *basicAMR*, &in returns true if <AMR>.inst exists in (<token>+). If <AMR> is an *orAMR* or *andAMR*, &in returns true if *any* <AMR>.inst exists in (<token>+).

&true
always returns T.

The following functions are implemented using their Lisp counterparts: **&and &eval &if &not &null &or &quote**

4.4 Built-in Recasts

(<AMR> ++ ({<role> <value>}+))
Add Recast. Returns a copy of <AMR> with added <role>-<value> pairs. Adding the reserved role :inst overwrites <AMR>.inst. If <AMR> is a *word* or a *wordlist*, a *basicAMR* of the form (0 / <AMR> {<role> <value>}+) is returned.
Examples
((0 / x :a a) ++ (:b b :c c)) *returns* (0 / x :a a :b b :c c)
("X" ++ (:d d)) *returns* (0 / "x" :d d)
((0 / x :a a) ++ (/ y :d d)) *returns* (0 / y :a a :d d)

(<AMR> -- (<role>+))
Delete Recast. Returns a copy of <AMR> with all <role>-<value> pairs removed. Deleting the reserved role :inst causes the replacement of the <value> of :inst with nil.
Examples
((0 / x :a a :b b1 :b b2) -- (:b :z)) *returns* (0 / x :a a)
((0 / x :a a :b b :c c) -- (/ :c)) *returns* (0 / nil :a a :b b)

(<AMR> && ({<role> <value>}+))
Replace Recast. Returns a copy of <AMR> with all values of <role> replaced with <value>. If <role> doesn't exist in <AMR>, it is added. If <AMR> is a *word* or a *wordlist*, a *basicAMR* of the form (0 / <AMR> {<role> <value>}+) is returned.
Examples
((0 / x :a a :b b1 :b b2) && (:b b3 :z z))
returns(0 / x :a a :b b3 :b b3)

(<AMR> << (<new-role> / (<role>+)))
Simple Recast. Returns a copy of <AMR> with all <role>s renamed as <new-role>.
Examples
((0 / x :a a :b b) << (:c / (:a :b))) *returns* (0 / x :c a :c b)

(<AMR> <? (<new-role> / (<role>+)) <cond>)

Conditional Recast. Returns a copy of <AMR> with all <role>s renamed as <new-role>, if <cond> is true. The special referential role @that should be used in <cond> to access the value of each recastable <role> one at a time. This is important in the case that several <role>s share the same name.

Examples

((0 / x :a a :b b1 :b b2) <? (:c / (:a :b)) (&eq @that.inst b1))

returns (0 / x :a a :c b1 :b b2)

Conditionally recast :a and :b into :c if the inst value of "that" recastable role (:a or :b) equals b1.

(<AMR> <! ((<new-role>+) / (<role>+)))

Hierarchical Recast. Returns a copy of <AMR> with <role>s hierarchically renamed as <new-role>s. Hierarchical renaming means that the first *existing* <role> is renamed to the first <new-role>; and the second *existing* <role> is renamed to the second <new-role>; and so on.

Examples

((0 / x :d d :g g :a a) <! ((:m :n) / (:a :b :c :d :e :f :g :h)))

returns (0 / x :n d :g g :m a)

(<AMR> <o <role>)

Order Recast. Returns a copy of <AMR> with its <role>s renamed as <role>-i where i enumerates the order in which <role> appears in <AMR>.

Example

((3 / x :a (1 / a) :b (2 / b1) :b (4 / b2)) <o :b)

returns (3 / x :a (1 / a) :b-1 (2 / b1) :b-2 (4 / b2))

(<AMR> <on <role>)

Label Order Recast. Returns a copy of <AMR> with its <role>s renamed as <role>-i where i is the node label of the value of <role>.

Example

((3 / x :a (1 / a) :b (2 / b1) :b (4 / b2)) <o :b)

returns (3 / x :a (1 / a) :b-2 (2 / b1) :b-4 (4 / b2))

(<AMR> <oi <role>)

Relative Order Recast. Returns a copy of <AMR> with its <role>s renamed as <role>-i or <role>+i where i is the absolute difference between the node label number of the value of <role> and the node label number of <AMR>. + is used for positive difference and - for negative difference. Obviously, this recast expects that the node labels are positive integers.

Example

((3 / x :a (1 / a) :b (2 / b1) :b (4 / b2)) <oi :b)

returns (3 / x :a (1 / a) :b-1 (2 / b1) :b+1 (4 / b2))

`(<AMR> +- <morpheme>)`

Morphology Recast. Returns a string that is the result of combining `<AMR>` with `<morpheme>`. This recast fires the internal morphology handler `oxymorph` which is linked to a user-defined morphology function through the oxyL declaration `:MORPH`. The form of the `<AMR>` (i.e. word, wordlist, basicAMR, etc.) and the form of `<morphem>` (i.e. word, list of words, even an AMR) is absolutely up to the user-defined morphology function.
Example

```
:MORPH (&english-morph @word @morphemes)
("walk" +- past)
```

returns `"walked"`

4.5 Reserved Tokens

Since the oxyL files are compiled into Lisp by a program written in Lisp using supporting Lisp functions, it is important that the oxyGen user shouldn't redefine any of the variables and functions that are necessary for the proper operation of the system. The following is a list of all the reserved tokens in the oxyGen system.

4.5.1 Reserved Variables

```
*oxycompile-class* *oxycompile-local* *oxycompile-debug*
*oxydb-stream* *oxydebug* $this $that
```

4.5.2 Reserved Roles

```
:INST :OR :AND :X-ROLE :ROLE (:THIS :THAT)
```

4.5.3 Reserved Functions

oxyL Functions

```
@this @that oxymain oxymorph @inst @/
(@or @and @x-role @role)
```

oxyCompile

```
oxycompile-file load-oxyl-file init-oxycompile oxycompile
print-compiled-grammar remove-oxydebug compile-grammar
compile-grammar-1 compile-grammar-def-recast
compile-grammar-def-rule compile-grammar-main-rule
compile-grammar-rule compile-grammar-set compile-grammar-conds
compile-grammar-conds-eq compile-grammar-conds-neq
compile-grammar-results compile-recast separate-roles amrp
```

```
compile-term variable-term local-term reserved-func-term
compiled-reserved-func-term role-term roleseq roleseq-1
roleseq-2 tokens
```

oxyRun

```
add-roles normalize-inst del-roles del-roles-1 replace-roles
sub-roles sub-roles-1 sub-roles-cond sub-roles-cond-1
sub-roles-hierarchy sub-roles-h-1 sub-order-role
sub-order-role-node sub-order-role-inode exval exval-1
inval inval-1 valof valof-1 prepare &amrType subamr-roles
permute permute-1 multiply-X-roles get-x-roles del-x-roles
multiply-subamr oxygen oxygen-file
```

oxyLin

```
oxylin gls-to-surface gls-to-surface-1 add-on check-size
format-surface format-surface-sentence
```

oxyDebug

```
oxydb-open oxydb-close oxydebug &oxydb oxy-pamr oxy-pamr1
oxy-pamr2 oxy-pamr-file
```

4.5.4 Reserved Strings

```
"*start-sentence*" "*end-sentence*" "*empty*"
```


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.

Bibliography

[1] Bonnie J. Dorr, Nizar Habash, and David Traum. A Thematic Hierarchy for Efficient Generation from Lexical-Conceptal Structure. In *Proceedings of the Third Conference of the Association for Machine Translation in the Americas, AMTA-98, in Lecture Notes in Artificial Intelligence, 1529*, pages 333–343, Langhorne, PA, October 28–31 1998.

[2] Nizar Habash. oxyGen: A Language Independent Linearization Engine. In *Fourth Conference of the Association for Machine Translation in the Americas, AMTA-2000*, Cuernavaca, Mexico, 2000.

[3] Nizar Habash and Bonnie Dorr. Large-Scale Language Independent Generation Using Thematic Hierarchies. In *Proceedings of MT Summit VIII, Santiago de Compostella, Spain*, 2001.

[4] ISI, University of Southern California. *The Penman Reference Manual*, December 1989.

[5] Irene Langkilde and Kevin Knight. Generating Word Lattices from Abstract Meaning Representation. Technical report, Information Science Institute, University of Southern California, 1998.

[6] Irene Langkilde and Kevin Knight. Generation that Exploits Corpus-Based Statistical Knowledge. In *ACL/COLING 98, Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics (joint with the 17th International Conference on Computational Linguistics)*, pages 704–710, Montreal, Canada, 1998.

[7] Dekang Lin. Dependency-Based Evaluation of MINIPAR. In *Proceedings of the Workshop on the Evaluation of Parsing Systems, First International Conference on Language Resources and Evaluation*, Granada, Spain, May 1998.

[8] David Traum and Nizar Habash. Generation from Lexical Conceptual Structures. In *Proceedings of the Workshop on Applied Interlinguas, North American Association of Computational Linguistics/Applied Natural Language Processing Conference, NAACL/ANLP-2000*, pages 34–41, Seattle, WA, 2000.