LANGUAGE GENERATION 1

REALIZATION USING TEMPLATES, ATNS, AND A PHRASAL GRAMMAR

Theme

An overview of realization. Three methods of syntactic language generation: templates, Augmented Transition Networks, and a phrasal grammar.

Summary of Contents

1. Levels of Generator Sophistication

The whole pipeline: going from some complex representation of meaning to a list of grammatical sentences. Three major stages:
- content selection and major structuring (macroplanning)
- sentence-level structuring/organization (microplanning)
- grammar-based single-sentence production (realization)

Why should one break it up this way? The main reason is the complexity of the problem—there does seem to be a natural progression (from deciding what you want to say in general to deciding in particular to actually phrasing it), and different types of information come into play at different stages of the process. There's quite a lot of psycholinguistic evidence for the unconscious processes of realization being different from the conscious ones of macroplanning.

2. Realization

Realization is the oldest part of language generation. Here you start with some input that represents the content of a single sentence, somewhat structured already, and produce a linear grammatical string of natural language words (called the "surface form").

Theoretically speaking, for realization the main problem is how to define the input. As we'll see, the closer you make the input to the surface form, the less realization has to do—but then, you push the work onto the earlier stages! There's no obvious boundary between microplanning and realization, unfortunately. But the choice of representation "depth" is very important, and often is dictated to you by the system that wants to use the generator. In the class we'll use a fairly neutral level.

Procedurally speaking, for realization the core problem is representing syntactic information and exercising appropriate control.

Range of increasingly flexible encodings:
- canned text
- template-based generation
- phrasal expansion
- feature collection and linearization

Canned text vs. templates. Advantages: easy to create and modify. Disadvantages: inflexible and not reusable. Every programmer writes a template generator by his/her third program!

But there's a problem: the more you want to vary the output, the more flexibility you have to build into the generator's templates. There's a natural progression away from the surface form; increasingly, info. is represented first as sequences of parts of speech (i.e., phrase structure rules), then as phrase structure patterns including some actual words, and finally as individualized units of variation (usually as features). Whenever you generalize (e.g., replace a word by its part of speech, or a part of speech by a set of features) then you also need to annotate each lexical item that can fit there with the appropriate feature(s). Also of course you need features for non-ordered info, such as number agreement. Eventually, the problem becomes one of representation: which features are you gong to define and use? What engine is going to combine them in the right order?

3. Representing Grammar Very Simply

The difference between syntax and semantics:
- no syntax, no semantics: *Green furiously sleep ideas colorless*.
- yes syntax, no semantics: *Colorless green ideas sleep furiously*.
- yes syntax, yes semantics: *Tired old men sleep happily*.

Two aspects to realization: sentence form (word order) and sentence content. These can to some degree be separated. We first focus on the form and then bring in the content.

Fairly direct implementation of phrase structure rules.

Example: Simple grammar:

S	→	NP VP	AUX NP VP	NP AUX VP
NP	→	DET ADJ* N PP*	N PP*
VP	→	V NP	V NP PP*
PP	→	PREP NP

Simple lexicon:

N	→	{man wagon woman child dog car fence house}
V	→	{eat drink push run see dream}
AUX	→	{can may will}
DET	→	{the a}
ADJ	→	{green big strong small happy lazy}
PREP	→	{in on over under by with}

4. Realizing Random Sentences using ATNs

As with parsing, we can represent the grammar in a Recursive Transition Network (RTN). Each grammar rule is written as a sequence of states linked by arcs.

To make a realizer, we have the system produce outputs along the way, thereby changing the RTN into an Augmented Transition Network (an implementation is described in (Simmons & Slocum, 1972). Whenever it crosses over an arc, the system has to generate the constituent stated on the arc. If the constituent is simple (just a word), the system simply outputs the word and continues. If it is complex (that is, another constituent, which itself may be a sequence of states), the system has to temporarily suspend the current

ATN, push into the new constituent's ATN, and realize that. When it has successfully completed the new one, the system pops back to the old ATN, back to the sequence that initiated the push, and continues.

Algorithm: Use a stack to hold unprocessed material.

START: Place "S" on the stack

GEN:

1. pop the next item on the stack; call it I

2. if I is a lexical item, print it out

    else call EXPAND with I and push the results onto the top of the stack

3. unless the stack is empty, go to 1

EXPAND (X):

1. if X is a word class in the lexicon, return a word randomly selected from that class

2. if X is a the head of a grammar rule, return an expansion randomly chosen from that rule

3. else there's an error

Doing this, you can generate grammatical English sentences, starting by loading "S" onto the stack, or noun phrases, starting by loading "NP" on the stack.

What problems do you see immediately?

(infinite loop of ADJs or PPs; no number agreement; random meaning)

For certain grammatical phenomena (such as number agreement in English), you have to carry around some data across arcs. How can you do this? You need to:

* extend the lexicon (so that you can choose the correct form of the word)

N = { (man [:number singular]) (men [:number plural]) ...}

V = { (eats [:number singular]) (eat [:number plural]) ...}

* extend the grammar rules to include the same type of information as appropriate:

S   →   NP [:number ?n] VP [:number ?n]
    →   AUX [:number ?n] NP [:number ?n] VP [:number INF]

* either create a global variable that carries the required information (but that's ugly; it can become messy in complicated situations) or extend the parameter Xs to include node names and features:

((EXPAND (NP [:number sing]))

(EXPAND (VP [:number sing])))

* extend the choice operation so that it selects number as well and creates the correct form(s) to return

* change the core algorithm as needed to handle this

5. Realizing Meaningful Sentences

Now we tackle the problem of *meaningful* communication. We need some input, to specify what our generator should say.

Obviously, if we give the system as input a syntax tree, there's nothing to do: you just read off the leaves, left to right! So we start with a somewhat more semantic input. Let's make it easy and start with what is known as 'shallow semantics'.

Input 1:

```
[P1 :type EAT
     :agent [P2 :type BOY
                   :nationality SWISS
                   :age 7
                   :determiner T]
     :patient [P3 :type PEACH
                    :color RED
                    :size BIG]
     :location [P4    :type STORE
                      :determiner T
                      :age OLD
     :time PAST]]
```

Compare this with the (somewhat 'deeper') representation:

Input 2:

```
[P1 :type INGEST
     :agent [P2 :type HUMAN
                   :nationality [:type COUNTRY :name "Switzerland"] :identifiable T
                   :age [:type MEASURE :unit YEAR :number 7] :gender MALE :number 1]
     :patient [P3 :type PEACH :identifiable T :number 1 :color RED :size +6]
     :location [P4 :type STORE :number 1 :identifiable T
                   :age [:type MEASURE :degree +6]]
     :eventtime 2001:10:05:12:03          (when did the eating happen?)
     :speakingtime 2001:10:06:13:13 (when does the speaking happen?)
     :referencetime 2001:10:06:13:13      (when is the speaker's point of reference?)
     :polarity POSITIVE                   (not NEGATIVE)
     :speechact ASSERTION]                (not QUESTION or ORDER)
```

Note the differences in notation (the treatments of age and time, :determiner vs. :identifiability, etc.) and the amount of defaulted information (polarity, speechact, etc.). The linguistic (syntactic and semantic) theory/ies you are using will specify what kinds of relations and what kinds of fillers you must use in your notation. For now, we stay shallow and focus on the algorithm.

How can you now generate meaningful sentences? Using this as input, the ATN traversal has to be changed a little. You still follow the arcs and output words. However, two things change:
*   arc choice is no longer random,
*   lexical choice is no longer random.

Both these decisions must somehow be guided by the input. Each arc will express some portion of the input. As you travel 'down' through the grammar, the portions of the input you consume get smaller and smaller, bottoming out in words.

To do this, we extend the grammar's rules with links to the input:

1. S[head]	→	NP[head:agent] VP[head]
	→	AUX[head:aux] NP[head:agent] VP[head]
	→	NP[head:agent] AUX[head:aux] VP[head]
2. NP[head]	→	DET[head:determiner] ADJ1*[head] N[head:type] PP1*[head]
	→	N[head:type] PP1*[head]
3. VP[head]	→	V[head:type head:time] NP[head:patient]
	→	V[head:type head:time] NP[head:patient] PP1*[head]
4. PP1[head]	→	PP["to" head:destination]
	→	PP["at" head:location]
	→	PP["from" head:nation]
	→	PP["with" head:instrument]
	→	...
5. ADJ1[head]	→	ADJ[head:age]
	→	ADJ[head:nationality]
	→	ADJ[head:size]
	→	…
6. PP[role head]	→	PREP[role] NP[head]
7. N[head]	→	<get head from lexicon; get :number and inflect>
8. V[head time]	→	<get head from lexicon; inflect for time, :number>
9. AUX[head]	→	<get head from lexicon, inflect for time, :number>
10. ADJ[head]	→	<get head from lexicon>
11. PREP[head]	→	<get head from lexicon>

and then perhaps the lexicon may look like this:

N[man]	→	singular: man
	→	plural: men

6. Phrasal Expansion Generation

To sum all this up: we have arrived at full phrase expansion algorithm:

Data structures:

- stack, with head for input

    Input: above shallow or deep semantics

    Grammar: above rules, as extended

    Lexicon: as above

    Algorithm:

    1. get the top of the stack, bind its head to the head of the input, and push the result onto stack
    2a. pop the top of the stack
      b. if it is a lexical item, print it
      c. else: find an appropriate expanded phrase for the top from the phrasal grammar
                (if there are alternatives, execute some selection function)
          for each portion of this phrase, find the approp. portion of the input, as specified
                in the portion (this may be a lexeme) and bind it into the phrase

> perform any side effects (number agreement)
> push the list of bound portions back onto the stack
>
> 3. unless stack is empty, go to 2a

Example: work through algorithm

But now we see the following problems:

1. Allocating info to grammatical position: express nationality as ADJ or PP? ("the boy from Switzerland" or "the Swiss boy"?)

2. How to encode locutions for Age, Nationality ("seven-year-old...")? In lexicon? In phrase-grammar patterns?

3. Backtracking: yes or no? May go down an arc, into a subgrammar, and then get stuck. Need to know when—your need a planner or a set of fully determinate criteria on top. The MUMBLE generator (McDonald, 1980) could get stuck; some backtracking in Spokesman's planner (Meteer et al.).

4. Need very syntactically oriented input (realiz classes)—whoever creates the input for the generator needs to know lots of grammar! How else to guarantee that the realiz class will provide enough consituents to cover the whole input? Problem of guaranteeing expressibility: Meteer thesis (Meteer, 1990).

5. Hard to build extensive grammar.

7. Going beyond phrasal realization

So far, we have built a *deterministic* system—a system that will always do one thing, the right thing, predictably. Our problem has been to ensure that we can handle all the inputs we are likely to get easily. We can still continue along the path of determinism—and we will—but we need to make the phrasal representation more flexible. Later, we will discuss non-deterministic realization, and the statistical approaches which might be seen as a kind of semi-non-determninism.

We go about making the realization representation more flexible as follows. Note that although we can still represent the grammar in an ATN, we are starting to put so much information on the arcs and in the selection that a different coding scheme may be more comfortable. The problem gets even worse when there are many alternative ways of saying essentially the same thing—choosing the right verbs and nouns, for example "terrorist" or "freedom fighter" or "guerrilla". One approach is to encode the 'extended grammar' as a set of functions, one function per extended rule, each function with more or less the following shape:

```
function (parameters)
 if (condition 1) then fill and return expansion pattern 1
   else if (condition 2) then fill and return expansion pattern 2
     else if (condition 3) then fill and return expansion pattern 3
       ...
```

In one way or another, this is what this style of generator tends toward. In MUMBLE, the most famous generator of this type, McDonald called these functions "realization classes" (McDonald, 1980).

A very nice feature of this approach is the flexibility of the realization class patterns. They can be traditional phrase structure rules, but they can equally easily be fixed or semi-fixed phrases:

> VP[DIE :tense ?t :numb ?n]   →   [V["kick" :tense ?t :numb ?n] the bucket]

One might argue that such multi-word semi-fixed phrases are actually lexical items, not grammatical patterns. This feature illustrates the continuity from grammar to lexicon. Some grammarians believe that

the phrase structure patterns are simply very general examples of ordering patterns that may be arbitrarily specific, in the limit pinned to a single word. Tree Adjoining Grammar (TAG (Joshi)) is one such example, and is fairly popular in Computational Linguistics. Many phrasal patterns are 'anchored' in specific words.

Optional Readings

ATN realization: Simmons, R.F. and J. Slocum. 1972. Generating English Discourse from Semantic Networks. *Communications of the ACM* 15(10).

Phrasal realization: McDonald, D.D. 1980. Natural Language Production as a Process of Decision Making under Constraint. Ph.D. dissertation, Massachusetts Institute of Technology.

Guaranteeing phrasal expressibility: Meteer, M.W. 1990. The 'Generation Gap': The Problem of Expressibility in Text Planning. Ph.D. dissertation, University of Massachusetts (Amherst). Available as BBN Technical Report 7347.

TAG grammar: Joshi, A. Tree Adjoining Grammar. Proceedings of various ACL conferences.