James Kilbury, Düsseldorf

Parsing and Machine Translation

My remarks here are not intended for experts in computational linguistics but rather for translators who are involved in practical work and who are interested in machine translation (MT) as one aspect of computational linguistics. I will begin with some elementary points and touch on more difficult issues later.

A definition of parsing is complicated by the fact that the word is used in several distinct senses. Parsing is generally acknowledged to involve the syntactic decomposition of complex linguistic forms into their constituent phrases and words, thereby producing a syntactic representation in the form of a tree or some other formal structure. In recent years the first sense of parsing has been extended to include the production of a semantic representation of some kind.

Although not all investigators agree, parsing in the sense of syntactic analysis is generally viewed as a prerequisite both to semantic analysis and to MT. Differences in syntax, such as those involving features of word order, can correlate with differences in meaning that must be captured in a translation. Whether or not parsing in the sense of semantic analysis constitutes a prerequisite to MT depends on one's understanding of semantics and on specific features of the source and target languages. Obviously, the meaning conveyed by a text in the source language must also, in some clear sense, be conveyed by the translation in the target language. Syntactic parsing alone does not guarantee such a transfer of meaning, so at least some semantic analysis is necessary.

On the other hand, a "complete" semantic analysis is not always necessary or even desirable for MT. We often can translate a text without understanding it completely. Consider the following passage from a German newspaper:

(1) Ein russisches Kampfflugzeug hat gestern [...] einen georgischen Hubschrauber abgeschossen. Beide Piloten kamen ums Leben.

A Russian fighter aircraft [...] shot down a Georgian helicopter yesterday. Both pilots lost their lives.

It takes some ingenuity on the part of the reader to recognize that the pilot

of the fighter plane probably is not dead, since he just shot the helicopter down, and that the helicopter must therefore have had two pilots. No existing commercial system for MT can handle such problems; where understanding of this sort is necessary for translation, currently available MT systems simply fail. Some advanced systems under development in research projects seek to simulate such aspects of text understanding. But fortunately in this example, we don't have to identify the referent of German ‹beide›; we simply use English ‹both› and retain the ambiguity which an "adequate" semantic analysis could be expected to resolve.

Whether or not it includes semantics, parsing in the sense that is directly relevant to MT constitutes analysis in algorithmic steps that can be implemented in computer programs. Parsing in the sense of the cognitive processes that go on in people's heads when they understand language may be quite different from the algorithmic model we assume; we know too little about such processes and whether they can be modelled with the algorithmic manipulation of symbols. Algorithms themselves are finite sequences of explicit operations that can be carried out in a finite number of steps to solve some explicitly defined problem. We can view a recipe for baking a cake as an informal algorithm, although it normally is not stated in a way that would allow us to implement it on a computer.

It should be intuitively clear that the syntactic parsing of a natural language involves information about the syntax of that particular language as well as information concerning strategies for finding the structure of a given sentence. In MT systems up to the 1970's these two kinds of information, now distinguished as declarative and procedural information, respectively, were mixed together in single, integrated packages such as the Augmented Transition Networks of William Woods. Beginning in the decade that followed, linguists following Martin Kay sought to separate these two kinds of information in computational linguistic systems. A view emerged around 1980 according to which grammatical information about a particular language should be stated in a declarative formalism, i.e. a special formal language, which in turn is interpreted by a procedurally-based algorithm. In this sense we can say that a parser is a special program which interprets the grammatical description of a language as a knowledge base in order to analyze particular sentences.

The advantage of this division of labour is considerable. Since the parser is independent of particular languages, we can use the same parser with a new grammar to translate from a new source language. On the other hand, if we want to translate not only from but also into a given language, we need a generator as well as a parser, and if we are clever enough, we may be able to design a language-independent program for generation that interprets the same grammars as our parser.

I have said that a parser interprets a grammar formalism and that the latter is the special formal language in which an explicit grammatical description is stated. A simple such formalism is that for context-free grammars. These distinguish between phrasal categories (such as **sentence**) and lexical categories (such as **verb**). In its most primitive form, a lexicon simply lists words and matches them with their lexical categories. Syntax rules are stated with a single phrasal category to the left and one or more lexical or phrasal categories to the right of an arrow. In the examples for parsing, the following set of context-free syntax rules will serve as our grammar:

(2) (a) S --> NP VP
(b) NP --> PropN
(c) NP --> Det N
(d) VP --> V
(e) VP --> V NP

The category symbols employed are abbreviations for ‹sentence›, ‹noun phrase›, ‹verb phrase›, ‹proper noun›, ‹determiner›, ‹noun›, and ‹verb›, in the order of their appearance.

Although the rules stated here are highly simplified and cover only a small part of English syntax, they nevertheless serve to illustrate the type of grammar rules that can be interpreted by the parsing algorithms I shall discuss. A genuine problem lies in the fact that these rules represent grammatical categories as atoms which bear no grammatical features such number and which fail to capture relations like that of agreement between a finite verb and its subject.

In conjunction with the notion of unificationgrammar introduced by Martin Kay around 1980, grammatical categories have come to be widely regarded as complex informational structures that can be represented as feature structures such as the following:

(3)
$$\begin{bmatrix} \text{category:} & \text{noun} \\ \text{agreement:} & \begin{bmatrix} \text{number:} & \text{singular} \\ \text{person:} & \text{third} \end{bmatrix} \end{bmatrix}$$

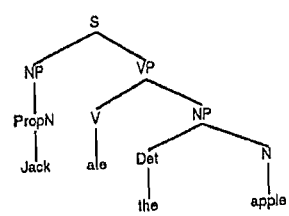
This represents a noun bearing the agreement features of singular number and third person. Within feature structures, features may have either atoms or other complex feature structures as their values. Features such as **agreement** with complex values serve to bundle packages of grammatical information that belong together. The rules of a unification grammar, in this case PATR-II as developed by Stuart Shieber and others, can then refer to this bundled information:

(4) S --> NP VP: <NP agreement> = <VP agreement>

Here the capital letters denote feature structures corresponding to grammatical categories, and the equation states that these feature structures are to share certain specifications. Such equations involve unification, which constitutes the only information-combining operation of unification grammar and which is based on the principle that information in feature structures can be added but never changed within an analysis. The context-free base of unification grammar allows us to profit from the latter's expressive power although we use the same parsing algorithms that were developed for context-free grammars. Unification, which allows no information to be changed during the course of the analysis of a sentence, in turn guarantees that we can use a declaratively formulated grammar with procedurally different parsing algorithms precisely because the grammar does not contain or depend on procedural information.

Now we can look at several concrete procedural strategies for syntactic analysis. First consider the analysis of a sentence represented in the following tree:

(5)



The tree consists of nodes labelled with syntactic categories or words and connected to each other by edges. Terminal nodes, or leaves, of the tree are labelled with words, and the other nodes with syntactic categories. Nonterminal nodes directly above leaves are labelled with lexical categories, and other nonterminal nodes with phrasal categories. The edges in this tree have been numbered so that we can refer to them simply. Moreover, we can say that a node has the same number as the edge leading downward to it. This is useful in order to distinguish node 1 from node 7, both of which are labelled with **NP** in this tree. The root of the tree is that node into which no edge leads.

The first strategy is top-down depth-first (TDDF) and begins with the expectation that the given input string to be analyzed, here ‹Jack ate the apple›, is in fact a sentence. The parser then attempts to verify this hypothesis. In order to carry out this verification, the algorithm uses a stack, which behaves like a stack of papers on your desk: only the top element of the stack is visible at any moment, and you can pop (‹take off›) symbols from the stack in their order from top to bottom, or else push (‹put›) symbols on the stack.

Our TDDF parser first has the stack **[S]** because it expects a sentence. Using rule a from the grammar in (2) above, it then pops S from the stack and pushes VP and NP in that order onto the stack, which then appears as **[NP VP]**. Now the parser looks at the top symbol, NP, and finds the two rules b and c which expand it. Our parser takes rule b, pops NP, and pushes PropN, so that the stack now appears as **[PropN VP]**. The current word ‹Jack› in the input string happens to be a proper noun, so PropN is popped from the stack, leaving **[VP]**. Now VP can be expanded using either rule d or e. Our parser has bad luck with its choice this time, because it goes looking for a verb according to rule d, and ‹ate› is a verb. After the pops and pushes the stack is empty, so the analysis of the input string should be finished, but instead the substring ‹the apple› still needs to be analyzed as part of the sentence. So the parser is stuck in a dead end.

What the parser needs here is a systematic way to backtrack. Any child knows how to get out of a maze, or labyrinth, if there is a way out: You need lots of string and a piece of chalk; you tie the string somewhere and take the other end with you so you know where you have been; whenever the

path splits and you have to make a choice, you mark the path you select so that you can try the others if your choice later proves to be wrong.

Using this technique the parser can restore the stack **[VP]** as it was after ‹Jack› was analyzed. Now VP is popped from the stack and the symbols V and NP from the right side of rule *e* are pushed in reverse order, giving **[V NP]**. Then V can be popped when the lexical entry for ‹ate› is found, and the stack **[NP]** results when ‹the apple› is left as the remainder of the input string. Parsing terminates successfully when the stack and the remainder of the input string are both empty. The parser keeps a record of the successful choices it has made in the course of the analysis, and the result can be reported in the form of the tree in (5). The complete analysis is given in (6):

Input string	**stack**	**operation**
[Jack ate the apple]	[S]	pop S & push VP and NP by a
[Jack ate the apple]	[NP VP]	pop NP & push PropN by b
[Jack ate the apple]	[PropN VP]	shift & pop PropN
[ate the apple]	[VP]	pop VP & push V by d
[ate the apple]	[V]	shift & pop V
[the apple]	[ ]	backtrack
[ate the apple]	[VP]	pop VP & push NP and V by e
[ate the apple]	[V NP]	shift & pop V
[the apple]	[NP]	pop NP & push PropN by b
[the apple]	[PropN]	backtrack
[the apple]	[NP]	pop NP & push N and Det by c
[the apple]	[Det N]	shift & pop Det
[apple]	[N]	shift & pop N
[ ]	[ ]	

A problem arises for the TDDF parser if linguists decide to add a rule like that of (7) to the grammar to account for relative clauses:

(7) NP --> NP S

If the parser is given a grammar with such a rule and any ungrammatical sentence as input, it fails to reject the input and instead continues for ever with attempts to expand the NP of the right-hand side of this rule with the same rule in which it is contained, so that NP is popped and S and NP are pushed without ever stopping. This means that a grammarian writing a grammar to be used with a TDDF parser must avoid writing rules of this left-recursive form.

Other strategies like that of a bottom-up shift-reduce (BUSR) parser avoid this restriction. Whereas the TDDF search for a successful analysis

is directed (or "driven") by the grammar, with BUSR it is driven by the input string itself. The parser begins with an empty processing stack and without any syntactic expectation. Lexical categories are pushed onto the stack in shift operations on the basis of lexical entries, while reduce operations pop the symbols of the right-hand side of a rule (in reverse order) and push the symbol on the left-hand side of the same rule. A successful parse has the steps shown in (8):

input string	**stack**	**operation**
[Jack ate the apple]	[ ]	shift *Jack* & push PropN
[ate the apple]	[PropN]	pop PropN & push NP by b
[ate the apple]	[NP]	shift *ate* & push V
[the apple]	[V NP]	...
...	...	shift *the* & push Det
[apple]	[Det V NP]	shift *apple* & push N
[ ]	[N Det V NP]	pop N and Det & push NP by c
[ ]	[NP V NP]	pop NP and V & push VP by e
[ ]	[VP NP]	pop VP and NP & push S by a
[ ]	[S]	-

Since the remainder of the input string is empty and only S is left on the stack, the input has been analyzed as a sentence. The part of the table with dots ‹...› shows where the parser gets into a dead end by the incorrect choice of a rule for reduction, and then has to backtrack.

Notice that the parser would just as well have analyzed the input string ‹the apple› alone as a noun phrase, so that the BUSR strategy does not depend on having the correct expectation, as TDDF does. BUSR winds up making a lot of bad rule choices for reductions exactly because it maintains no expectations about what will come next, as TDDF does. A good strategy should be grammar-driven and data-driven at the same time and in such a way that the two kinds of information complement each other.

A left-corner (LC) strategy in fact combines the advantages of both TDDF and BUSR. I shall omit the formal presentation with stacks here. An LC parser takes the first word ‹Jack› of the input string and recognizes PropN as the left corner, or first symbol on the right-hand side, of rule b. Since there are no further symbols on the right-hand side of b, the parser gets an NP. This in turn is the left corner of rule a, so that a VP is now needed in order to get an S. The next word ‹ate› gives the parser a V, which by rules d and e is the left corner of a VP. (Note that we again have a choice with

the VP rules. A sensible grammar, of course, will include information about subcategorization and the difference between intransitive and transitive verbs, which helps to solve this problem.) The analysis continues and finds the NP which is needed by the VP which is needed to get an S.

The performance of a LC parser is greatly improved if it has a top-down filter, which says, for example, that the PropN we get from ‹Jack› not only is the left corner of an NP but furthermore that this PropN, because of rule a, can in turn be the beginning of an S.

The LC strategy yields a highly efficient algorithm for syntactic parsing, and experts in the field of cognitive linguistics say that there is, in fact, much to suggest that humans use a similar strategy in their processes of language understanding. This is all the more remarkable since the LC strategy simply depends on the abstract formalism of context-free grammars and incorporates no special principles involving natural, as opposed to formal, languages. More recently developed strategies, such as head-driven parsing, depend crucially on specifically linguistic notions such as the head, or central constituent, of an endocentric syntactic construction. Current research is seeking to develop parsers that can switch flexibly between strategies, depending on the immediate problem to be solved.

We have seen that all three of the strategies presented can reach dead ends because of incorrect rule choices, which in principle are unavoidable because of structural ambiguity in natural language. So in the familiar example ‹A boy saw the girl with a telescope›, syntactic information cannot tell us whether the girl who was seen was one who had a telescope, or whether the boy who saw the girl did so with the use of a telescope. Backtracking provides one solution to this problem. An alternative appears in the form of the chart, which was introduced - again, by Martin Kay - as a special data structure that allows a parser to store all the information about alternative analyses that accumulates during parsing.

Up to now I have said nothing about semantic parsing. The latter is normally taken to be syntactic parsing together with the translation of an input string into an expression in some logical formalism; this logical expression is then viewed as the semantic representation of the input string. So, given lexical semantic representations like those in (9)

(9) every for all x : if P(x) then Q(x)
baby baby'
cries cry'

substitution of the expression baby' for P in the semantic representation for every gives the representation ‹for all x : if baby'(x) then Q(x)› for the noun phrase ‹every baby›, and the subsequent substitution of ‹cry'› for Q in the latter gives the representation ‹for all x:if baby'(x) then cry'(x)› as the result of the semantic analysis of the sentence ‹Every baby cries.› None of this technique, developed mainly by Richard Montague, depends on the strategy for syntactic parsing that is employed - it works with any of them. What it does depend on is the principle of compositionality, which says that the meaning of a phrase or complex form is determined exclusively by the meanings of the constituents that make up the complex form. Compositionality can be neatly implemented using the techniques of unification grammar, and the logical formulas can be encoded as feature structures.

But compositionality holds only in part, because of phrasal lexemes, and this points to the first major area of problems in parsing: In order to parse a sentence we have to get the lexical entries of the words making it up. When a parser analyzes the sentence ‹Intelligence has a lot to do with flexibility›, it needs a lexicon that puts the idiosyncratic, nontransparent meaning of ‹have a lot to do with› in an appropriate place where the information can be found. This is an aspect of lexical representation that still poses problems.

Another difficulty arises when the parser finds words in the input string for which there are no appropriate entries in the lexicon. In the most fortunate case, that of overt gaps, there is no entry at all for some particular word, and the parser at least knows that information is missing so that it can try its best on the basis of contextual information. Far worse is the situation in which the parser must conclude that its lexicon indeed contains one or more entries for the given input word but that none of these entries fits the present context, and thus that the word is a case of covert gaps. Such lexical gaps are not merely a technical problem that can be solved with bigger dictionaries but rather reflect an essential feature of language: languages change and adapt their lexica to the

needs of users confronted with constantly changing situations in the world.

Even if the given input word is no clear case of a gap, it may be extremely difficult - even for a human linguist - to decide which of the senses listed for a word in fact matches the occurrence in a particular context. And if humans can't make such decisions after reflection, the MT systems can hardly be expected to perform better.

Parsing can be complicated by other problems which are ultimately connected with the lexicon. If a text to be translated contains typing errors, we would like these to be recognized as such rather than being treated as new and unknown words. More serious are the cases where the author of the text has suffered a lapse and written something which is meaningful but which he simply didn't intend. Good human translators catch this, and computers don't.

Of course, there are special problems in syntax that cause difficulties for parsers. Many of these involve cases of so-called long-distance dependencies, in which one constituent is removed from another which it grammatically depends on. Examples, involving topicalization (with parasitic gaps) and gapping, respectively, appear in (10):

(10) Corn some people use [corn] merely to feed pigs with [corn].

Alice wants to visit Barbara, and Susan [wants to visit] Jane.

Computational linguists like such problems very much and direct a lot of engery at solving them - much in the spirit of the Nasrudin story of the man who has lost his key and searches for it under the street lamp, not because he expects it there, but because there is more light. The real challenges for parsing lie in meaning and the lexicon, not in odd corners of syntax.

The success of automatic parsing, like that of MT in general, depends greatly on its concrete aims and tasks. The most successful example of automatic MT is probably the TAUM-METEO project at the University of Montreal, which developed a system employed practically to translate daily weather reports from English into French. This translation work was so mechanical and tedious that human translators nearly went mad after

doing it for several months. An attempt was later made to adapt the system to the domain of aircraft repair manuals; this could have been very useful, but the attempt failed.

Automatic MT is most successful with relatively dull and unimportant texts that most people wouldn't want to read very long. For the forseeable future, MT systems will probably remain the useful assistants and apprentices of human translators rather than becoming their competitors.

The potential of automatic MT should not be underestimated, however. It is easy to fall prey to an argumentum ad ignoratiam, supposing that that which we consider difficult or beyond our imagination, is in fact technically impossible. A new attempt to make an adaptation of TAUM-METEO might be much more successful, and people might then want to use such an automatic MT system practically even if it had serious defects.

The automatic translation of aircraft repair manuals would confront us with a new problem. It is unrealistic to suppose that editing could avoid all errors in the translations, but errors could lead to accidents and the loss of human life. Moreover, a system that could translate the repair manuals of commercial aircraft could undoubtedly be used with the manuals of weapon systems, and this would be a great advantage for people who are in the business of exporting modern weapons. So it is not enough just to ask in general whether automatic MT is possible or not; instead we must look at specific applications and ask ourselves whether these are desirable and acceptable.