Notes on Natural Language Processing (NLP)

Allen B. Tucker

February, 2002

These notes provide a framework for a beginning study of contemporary issues and strategies in natural language processing. They are accompanied by software and examples drawn from various sources. To gain the most from these notes, readers should be familiar with general ideas in computer science and programming, including (ideally) a modest knowledge of Prolog.

References

1. Allen, James, *Natural Language Understanding* 2e, Benjamin Cummings, 1995.

2. Charniak, Eugene, *Statistical Language Learning*, MIT Press, 1993.
3. Clocksin, W. and Mellish, C., *Programming in Prolog 4e*, Springer, 1997.
4. Covington, Michael, *Natural Language Processing for Prolog Programmers* , Prentice Hall, 1994.
5. Cowie, J. and Lehnert, W., Information Extraction, *Communications of the ACM* 39, 1 (Jan 1996), 80-91.
6. Guthrie, Louise et al., The Role of Lexicons in Natural Lnaguage Processing, *Communications of the ACM* 39, 1 (Jan 1996), 63-72.
7. King, Margaret, Evaluating Natural Lnaguage Processing Systems, *Communications of the ACM* 39, 1 (Jan 1996), 73-79.
8. Lewis, D. and Sparck-Jones, K., Natural Language Processing for Infromation Retrieval, *Communications of the ACM* 39, 1 (Jan 1996), 92-101.
9. Manning, C. and H. Schutze, *Foundations of Statistical Natural Language Processing*, MIT Press, 1999.
10. Matthews, Clive, *An Introduction to Natural Language Processing through Prolog*, Longman, 1998.
11. Pereira, F.C.N and Shieber, S, *Prolog and Natural Language Analysis* , CSLI, 1987.
12. Wiebe, J et al., Language Use in Context, *Communications of the ACM* 39, 1 (Jan 1996), 102-111.
13. Wilks, Yorick, Natural Language Processing, *Communications of the ACM* 39, 1 (Jan 1996), 60-62.

Modified: 19-Jul-1999

Overview of NLP: Issues and Strategies

Natural Language Processing (NLP) is the capacity of a computer to "understand" natural language text at a level that allows meaningful interaction between the computer and a person working in a particular application domain.

Application Domains of NLP:

- text processing - word processing, e-mail, spelling and grammar checkers
- interfaces to data bases - query languages, information retrieval, data mining, text summarization
- expert systems - explanations, disease diagnosis
- linguistics - machine translation, content analysis, writers' assistants, language generation

Tools for NLP:

- Programming languages and software - Prolog , ALE , Lisp/Scheme, C/C++
- Statistical Methods - Markov models, probabilistic grammars, text-based analysis
- Abstract Models - Context-free grammars (BNF), Attribute grammars, Predicate calculus and other semantic models, Knowledge-based and ontological methods

Linguistic Organization of NLP

- Grammar and lexicon - the rules for forming well-structured sentences, and the words that make up those sentences
- Morphology - the formation of words from stems, prefixes, and suffixes

  E.g., eat + s = eats

- Syntax - the set of all well-formed sentences in a language and the rules for forming them

- Semantics - the meanings of all well-formed sentences in a language

- Pragmatics (world knowledge and context) - the influence of what we know about the real world upon the meaning of a sentence. E.g., "The balloon rose." allows an inference to be made that it must be filled with a lighter-than-air substance.

- The influence of discourse context (E.g., speaker-hearer roles in a conversation) on the meaning of a sentence

- Ambiguity
  - lexical - word meaning choices (E.g., *flies*)

- syntactic - sentence structure choices (E.g., *She saw the man on the hill with the telescope*.)
- semantic - sentence meaning choices (E.g., *They are flying planes.*)

Grammars and parsing

Syntactic categories (common denotations) in NLP

- np - noun phrase
- vp - verb phrase
- s - sentence
- det - determiner (article)
- n - noun
- tv - transitive verb (takes an object)
- iv - intransitive verb
- prep - preposition
- pp - prepositional phrase
- adj - adjective

A *context-free grammar (CFG)* is a list of rules that define the set of all well-formed sentences in a language. Each rule has a left-hand side, which identifies a syntactic category, and a right-hand side, which defines its alternative component parts, reading from left to right.

E.g., the rule s --> np vp means that "a sentence is defined as a noun phrase followed by a verb phrase." Figure 1 shows a simple CFG that describes the sentences from a small subset of English.

Figure 1. A grammar and a parse tree for "the giraffe dreams".

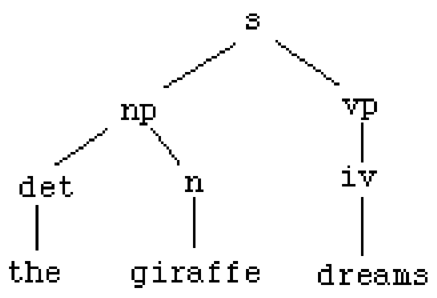
```
s    --> np vp

np   --> det n
vp   --> tv np
     --> iv

det  --> the
     --> a
     --> an

n    --> giraffe
     --> apple

iv   --> dreams
tv   --> eats
     --> dreams
```



A *sentence* in the language defined by a CFG is a series of words that can be derived by systematically applying the rules, beginning with a rule that has s on its left-hand side. A *parse* of the sentence is a series of rule applications in which a syntactic category is replaced by the right-hand side of a rule that has that category on its left-hand side, and the final rule application yields the sentence itself. E.g., a parse of the sentence "the giraffe dreams" is:

s => np vp => det n vp => the n vp => the giraffe vp => the giraffe iv => the giraffe dreams

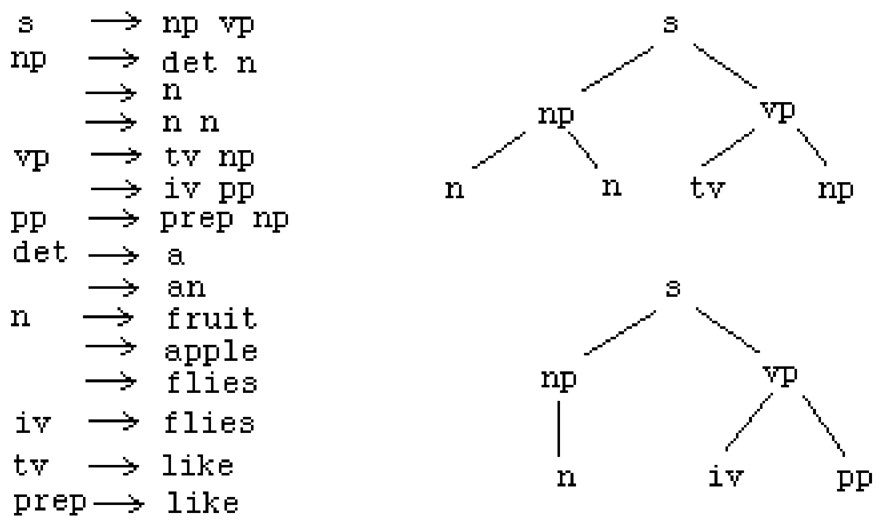
A convenient way to describe a parse is to show its *parse tree*, which is simply a graphical display of the parse. Figure 1 shows a parse tree for the sentence "the giraffe dreams". Note that the root of every subtree has a grammatical category that appears on the left-hand side of a rule, and the children of that root are identical to the elements on the right-hand side of that rule.

If this looks like familiar territory from your study of programming languages, that's a good observation. CFGs are, in fact, the orignin of the device called BNF (Backus-Naur Form) for describing the syntax of programming languages. CFGs were invented by the linguist Noam Chomsky in 1957. BNF originated with the design of the Algol programming language in 1960.

Goals of Linguistic Grammars

- Permit ambiguity - ensure that a sentence has all its possible parses (E.g., "fruit flies like an apple" in Figure 2)

Figure 2. An ambiguous grammar and partial parse trees for "fruit flies like an apple."



```
s    → np vp
np   → det n
     → n
     → n n
vp   → tv np
     → iv pp
pp   → prep np
det  → a
     → an
n    → fruit
     → apple
     → flies
iv   → flies
tv   → like
prep → like
```

- Limit ungrammaticality - E.g., require agreement in number, tense, gender, person. Disallow "the giraffe eat the apple" (Figure 1)

- Ensure meaningfulness - E.g., disallow "the apple eats the giraffe" (Figure 1)

NLP vs PLP (Programming Language Processing):

There are some parallels, and some fundamental distinctions, between the goals and methods of progamming language processing (design and compiler strategies) and natural language processing. Here is a brief summary:

	NLP	PLP
domain of discourse	broad: what can be expressed	narrow: what can be computed
lexicon	large/complex	small/simple
grammatical constructs	many and varied - declarative - interrogative - fragments etc.	few - declarative - imperative
meanings of an expression	many	one
tools and techniques	morphological analysis syntactic analysis semantic analysis integration of world knowledge	lexical analysis context-free parsing code generation/compiling interpreting

References

1. Matthews, Clive, *An Introduction to Natural Language Processing through Prolog*, Longman, 1998.
2. Allen, James, *Natural Language Understanding* 2e, Benjamin Cummings, 1995.
3. Wilks, Yorick, "Natural Language Processing," *Communications of the ACM* 39, 1 (Jan 1996), 60-62.
4. Covington, Michael, *Natural Language Processing for Prolog Programmers* , Prentice Hall, 1994.
5. Manning, C. and H. Schutze, *Foundations of Statistical Natural Language Processing*, MIT Press, 1999.

Prolog and NLP

Review of Prolog Fundamentals

Prolog programs are made from *terms*, which can be either *constants*, *variables*, or *structures*.

- A *constant* is either an atom (like `the`, `zebra`, `'John'`, and `'.'`) or a nonnegative integer (like 24)
- A *variable* is a series of letters (A-Z, a-z, _) that begins with a capital letter (like `John`).
  Note: constants cannot begin with a capital letter unless they are enclosed in quotes.
- A *structure* is a predicate with zero or more arguments, written in functional notation. E.g.,
  ```
      n(zebra)
      speaks(boris, russian)
      np(X, Y)
  ```
  The number of arguments is called the predicate's "arity" (1, 2, and 2 in these examples).

A *fact* is a term followed by a period (.). A *rule* is a term followed by `:-` and a series of terms (term1, term2, ..., termN) separated by commas and ended by a period (.). That is, rules have the following form:

```
term :- term1, term2, ..., termN .
```

A Prolog program is a series of facts and rules:

```
speaks(boris, russian).
speaks(john, english).
speaks(mary, russian).
speaks(mary, english).
understands(Person1, Person2) :- speaks(Person1, L), speaks(Person2, L).
```

This program asserts the facts that Boris and Mary speak Russian and John and Mary speak English. It also defines the relation "understands" between two persons, which is true exactlyl when they both speak the same language, denoted by the variable L.

A Prolog rule succeeds only when there are assignments (called "instantiations") of its variables for which each of the terms on its right-hand side (of the `:-` operator) is simultaneously satisfied for those assignments. Otherwise, the rule is said to fail.

To exercise a Prolog program, one writes "queries" in reply to the Prolog prompt `?-`. Here is a simple query that asks the question "Who speaks Russian?"

```
     ?- speaks(Who, russian).
```

In reply, Prolog tries to satisfy a query by finding a fact or series of fact/rule applications that will resolve the query; that is, one that will assign values to the variables in the query that cause the fact or rule to succeed, in the pattern-matching sense.

In this particular example, Prolog examines all the facts and rules that have "speaks" as a predicate (there are four), in the order they are written in the program. Since "russian" is provided as a constant, the only variable to be resolved is the variable "Who". Using the first fact in the program, Prolog replies with:

```
     Who = boris
```

since that assignment to the variable Who causes the first fact to succeed. At this point, the user may want to know if there are other ways of satisfying the same query, in which a semi-colon (;) is typed. Prolog continues its search through the program, reporting the next success, if there is one. When there are no more successful instantiations for the variable "Who," Prolog finally replies "No" and the process stops. Here is the complete interaction:

```
     ?- speaks(Who, russian).
     Who = boris ;
     Who = mary ;
     No
     ?-
```

Another sort of query that this program can handle is one that asks questions like "Does John understand Boris?" or "Who understands Boris?" or "Find all people that understand each other." These can be written as the following Prolog queries, with the replies to the first two shown as well:

```
     ?- understands(john, boris).
     No
     ?- understands(Who, boris).
     Who = boris ;
     Who = mary ;
     No
     ?- understands(P1, P2).
```

To see how these queries are satisfied, we need to see how the rule for "understands" is evaluated. Any query of the form "understands(X, Y)" appeals to that rule, which can be satisfied only if there is a common instantiantion for the variables X, Y, and L for which "speaks(X, L)" and "speaks(Y, L)" are both satisfied. These two terms are often called "subgoals" of the main goal "understands(X, Y)".

Prolog takes the subgoals in a rule from left to right, so that a search is first made for values of X and L for which "speaks(X, L)" is satisfied. Once such values are found, these same values are used wherever their variables appear in the search to satisfy additional subgoals for that rule, like "speaks(Y, L)".

Let's follow the process of trying to satisfy the first query above:

```
understands(john, boris)
        |    5 | fail
        |      |
      1 |      |
   speaks(john, L)       speaks(boris, L)
        |                   3  /      |
        |      L = english  /         |
      2 |                    /      4 |
   speaks(john, english)      fail
```

This query fails because the only instantiation of L that satisfies "speaks(john, L)" does not simultaneously satisfy "speaks(boris, L)" for this program. The numbers assigned to the links in the search diagram indicate the order in which subgoals are tried.

The second query above has a somewhat more complex search process, which begins in the following way.

```
understands(Who, boris)
        |    5 |
        |      |  Who = boris
      1 |      |
   speaks(Who, L)           speaks(boris, L)
        |                      3 /      |
        |      Who = boris    /         |
        |       L = russian  /          |
      2 |                   /        4 |
   speaks(boris, russian)      speaks(boris, russian)
```

Once this step is completed, with the reply "Who = boris," the process continues to search for other instantiations of Who and L that will satisfy goals 1 and 3 simultaneously. Thus, the process eventually uncovers the answer "Who = mary", and no more.

Sentences as Lists of Atoms

The basic data structure in Prolog programming is the *list*, which is written as a series of terms separated by commas and enclosed in brackets []. Sentences are usually represented as lists of atoms, as in the following example:

```
     [the, giraffe, sleeps, '.']
```

The empty list is denoted by [], while a "don't care" entry in a list is represented by an underscore (_). The following denote lists of one, two, and three elements, respectively.

```
     [X]
     [X, Y]
     [_, _, Z]
```

The head (first) element of a list is distinguished from its remaining elements by a vertical bar. Thus,

```
     [X | Y]
```

denotes a list whose head is the element X and whose remaining elements form the list [Y].

Here is a simple Prolog function that defines the concatenation of two lists together to form one (Matthews, 74). The first two arguments for this function represent the two lists, and the third represents the result.

```
     concat([], X, X).
     concat([H | T], Y, [H | Z]) :- concat(T, Y, Z).
```

This rather odd-looking recursive definition has two parts. The "base case" is defined by the first line, which simply affirms that the empty list concatenated with any other list returns that other list as the result. The recursive case, defined by the second line, says that if Z is the result of concatenating lists T and Y, then concatenating any new list [H | T] with Y gives the result [H | Z].

The dynamics of execution in this case is important to understand, since this form of recursive definition occurs persistently in Prolog. Consider the query

```
?- concat([english, russian], [spanish], L).
```

which presumably will yield the list L = [english, russian, spanish]. Here is a partial trace of the various instantiations of variables H, T, X, Y, and Z as this result is developed.

```
concat([english, russian], [spanish], L).
        |
        |   H = english, T = [russian], Y = [spanish], L = [english | Z]
      1 |
concat([russian], [spanish], [Z]).
        |
        |   H = russian, T = [], Y = [spanish], [Z] = [russian | Z']
      2 |
concat([], [spanish], [Z']).
        |
        |   X = [spanish], Z' = spanish
      3 |
concat([], [spanish], [spanish]).
```

The first two calls use the recursive rule, which just strips the head H from the first argument and recalls with a shorter list as first argument. The final call uses the base case, which forces instantiantion of the variable X = [spanish] and Z' = spanish. Thus, the result [H | Z'] in the second call is resolved as [russian, spanish] and identified with the list Z in the first call. Finally, the list L is resolved to [english, russian, spanish], using this newly-discovered value for Z as its tail.

The use of primes (') in this analysis helps to distinguish one use of a variable in a rule at one level of recursion from another use of the same variable at another level.

Pre- and Postprocessing Sentences

In order to have a user-friendly interface, a natural language processing system must provide support for users to type sentences in the ordinary way, and to view them on the screen in their usual form. Since the normal style of sentence representation for Prolog processing is as a list of atoms, auxiliary functions must be provided to convert the user's sequence of keystrokes that comprise a sentence into an appropriate list, and then to go the other way and display the list representation of a sentence as a series of words.

Below is a collection of functions that accomplish these tasks. Those which preprocess the user's typing are headed by the function read_sent. The function which displays a sentence is called display_sent.

```
% Prolog functions to read a sentence (adapted from Clocksin & Mellish 1994).
read_sent([W | Ws]) :- get0(C), read_word(C, W, C1), rest(W, C1, Ws).

% Given a word and the next character, read the rest of the sentence
rest(W, _, []) :- terminator(W), !.
rest(W, C, [W1 | Ws]) :- read_word(C, W1, C1), rest(W1, C1, Ws).

% Read a single word, given an initial character, and remember next character.
read_word(C, W, C1) :- punctuation(C), !, name(W, [C]), get0(C1).
read_word(C, W, C2) :- in_word(C, NewC), !, get0(C1), rest_word(C1, Cs, C2),
                       name(W, [NewC | Cs]).
read_word(C, W, C2) :- get0(C1), read_word(C1, W, C2).
rest_word(C, [NewC | Cs], C2) :- in_word(C, NewC), !, get0(C1),
                                 rest_word(C1, Cs, C2).
rest_word(C, [], C).

% These are the ASCII codes for single-character words (punctuation marks).
punctuation(44).   % ,
punctuation(59).   % ;
punctuation(58).   % :
punctuation(63).   % ?
punctuation(33).   % !
punctuation(46).   % .

% These can be inside a word; capitals are converted to lower case.
in_word(C, C) :- C>96, C<123.           % ASCII a, b, ..., z
in_word(C, L) :- C>64, C<91, L is C+32.  %       A, B, ..., Z
in_word(C, C) :- C>47, C<58.            %       0, 1, ..., 9
in_word(39, 39).                        %       '
in_word(45, 45).                        %       -

% These terminate a sentence.
terminator('.').
terminator('!').
terminator('?').


% Function to display a sentence that is stored as a list.
display_sent([]) :- nl.
display_sent([W | Rest]) :- name(W, [W1 | Rest]), punctuation(W1), !,
                            write(W), display_sent(Rest).
display_sent([W | Rest]) :- write(' '), write(W), display_sent(Rest).
```

To help clarify these definitions, it is important to understand the use of the Prolog functions get0 and name. The call get0(C) simply retrieves the next character typed by the user and associates it with the variable C. The call name(W, L) associates any word W, which is an atom, with the list L of ASCII codes which represent its characters. Thus, the call `name(giraffe, X)` returns the list X = [103, 105, 114, 97, 102, 102, 101]. Similarly, the call `name(X, [103, 105, 114, 97, 102, 102, 101])` returns the atom X = giraffe.

The auxiliary facts punctuation, in_word, and terminator help to classify characters as punctuation marks, normal letters within a word, and sentence-terminators.

Finally, the function display_sent distinguishes two cases: one in which a word is a punctuation mark (in which case it is displayed without a leading space) and the other in which a word is not a punctuation mark. The Prolog function write displays an individual atom without any leading or trailing spaces. the function nl denotes "go to a new line on the screen."

Programming for NLP: a First Look

Suppose we want to define a Prolog program that is effectively a grammar that will parse sentences, like the grammar illustrated in Figure 1. Assuming the list representation for sentences, we can write Prolog rules that partition the sentence into its grammatical categories, following the structure of the grammar rules themselves. For example, consider the grammar rule:

```
     s --> np vp
```

A corresponding Prolog rule would be:

```
     s(X, Y) :- np(X, U), vp(U, Y).
```

Here, we have added variables that stand for lists. In particular, X denotes the list representation of the sentence being parsed, or "recognized," and Y represents the resulting tail of the list that will remain if this rule succeeds. The interpretation here mirrors that of the original grammar rule: "X is a sentence, leaving Y, if the beginning of X can be identified as a noun phrase, leaving U, and the beginning of U can be identified as a verb phrase, leaving Y."

The entire Prolog program for the grammar in Figure 1 is shown below (and is called "grammar1" in the course directory):

```
     s(X, Y) :- np(X, U), vp(U, V), terminator(V, Y).

     np(X, Y) :- det(X, U), n(U, Y).
```

```
    vp(X, Y) :- iv(X, Y).
    vp(X, Y) :- tv(X, U), np(U, Y).

    det([the | Y], Y).
    det([a | Y], Y).

    n([giraffe | Y], Y).
    n([apple | Y], Y).

    iv([dreams | Y], Y).

    tv([dreams | Y], Y).
    tv([eats | Y], Y).

    terminator(['.'], []).
    terminator(['?'], []).
    terminator(['!'], []).
```

The first rule in this program is slightly expanded to accommodate the terminator that occurs at the end of a sentence. The facts that describe the terminators are self-explanatory. The remaining rules in this program correspond identically with those of the grammar in Figure 1. Note that the facts that identify lexical items (like giraffe, eats, etc.) effectively strip those items from the head of the list being passed through the grammar.

To see how this works, consider the following Prolog query, which asks whether or not "the giraffe dreams." is a sentence.

```
    ?- s([the, giraffe, dreams, '.'], [])
```

Here, X and Y are identified with the two lists given as arguments, and the task is to find lists U and V that will satisfy, in the order given, each of the following three goals (using the right-hand side of the first rule in the program).

```
    np([the, giraffe, dreams, '.'], U)      vp(U, V)      terminator(V, [])
```

One way to see the dynamics of this process is to trace the processing of the query itself:

```
ï 1 | 1 call s([the,giraffe,dreams,.],[])
ï 2 | 2 call np([the,giraffe,dreams,.],_7091)
ï 3 | 3 call det([the,giraffe,dreams,.],_7401)
ï 3 | 3 exit det([the,giraffe,dreams,.],[giraffe,dreams,.])
ï 4 | 3 call n([giraffe,dreams,.],_7091)
ï 4 | 3 exit n([giraffe,dreams,.],[dreams,.])
ï 2 | 2 exit np([the,giraffe,dreams,.],[dreams,.])
ï 5 | 2 call vp([dreams,.],_7089)
ï 6 | 3 call iv([dreams,.],_7089)
ï 6 | 3 exit iv([dreams,.],[.])
ï 5 | 2 exit vp([dreams,.],[.])
```

```
ï 7 | 2 call terminator([.],[])
ï 7 | 2 exit terminator([.],[])
ï 1 | 1 exit s([the,giraffe,dreams,.],[])
```

From this trace, we can see that the variables U and V are instantiated to [dreams, '.'] and ['.'] respectively, in order to satisfy the right-hand side of the first grammar rule.

Notice that, upon exit from each level of the trace, one or more words are removed from the head of the list. The call to terminator removes the final atom from the list, completing the successful parse of the sentence.

A more careful analysis of this tace reveals a direct correspondence between the various successful calls (rule applications) and the nodes of the parse tree shown in Figure 1 for this sentence (excluding the extra nodes added for the terminator). So reading a trace can be helpful when developing a complex parse tree for a sentence.

Definite Clause Grammars: Simplifying the Notation

Using Prolog to encode complex grammars in this way is often more cumbersome than helpful. For that reason, Prolog provides a very compact notation that directly mimics the notation of context-free grammar rules themselves.

This notation is called the *Definite Clause Grammar* (DCG), and is very simple to asimilate. The new operator `-->` is substituted in place of the operator `:-`, and the intermediate list variables are dropped. So the Prolog rule

s(X, Y) :- np(X, U), vp(U, V), terminator(V, Y).

can be replaced by its equivalent simplified version:

s --> np, vp, terminator.

In making this transformation, it is important that we are not changing the arity of the function s (still 2) or the meaning of the rule itself. These notations are introduced as sort of "macros" which allow Prolog rules to be written almost identically to the grammar rules which they emulate. A complete rewriting of this grammar is shown below (and is available as "grammar2" in the course directory):

```
     s --> np, vp, terminator.

     np --> det, n.

     vp --> iv.
     vp --> tv, np.

     det --> [the].
     det --> [a].
     n --> [giraffe].
     n --> [apple].
     iv --> [dreams].
     tv --> [dreams].
```

```
    tv --> [eats].

    terminator --> ['.'] ; ['?'] ; ['!'].
```

The semicolon that separates the different kinds of terminators in the last rule means "or". Thus, this rule is equivalent to three separate rules for terminator that have three different right-hand sides. This grammar is fully equivalent to the previous grammar; queries are formed and program execution occurs in exactly the same fashion as before.

Further Refinements

To simplify and clarify our later work, we introduce two additional refinements to the grammar-writing rules, the capability to generate a parse tree directly from the grammar and a basis for designing a lexicon with lexical features.

The above grammar can be modified so that a query gives not just a "Yes" or "No" answer, but a complete parse tree in functional form as a response. For instance, the functional form of the parse tree in Figure 1 is:

```
    s(np(det(the), n(giraffe)), vp(iv(dreams)), terminator('.'))
```

This modification is accomplished by adding an additional argument to the left-hand side of each rule, and appropriate variables to hold the intermediate values that are derived in the intermediate stages of execution. For instance, the first rule in the grammar above would be augmented as follows:

```
    s(s(NP,VP)) --> np(NP), vp(VP), terminator.
```

This means that the query needs an extra argument, alongside the sentence to be parsed and the empty list. That argument, appearing first in the query is a variable that will hold the resulting parse tree, as shown below:

```
    ?- s(Tree, [the, giraffe, dreams, '.'], []).
```

The second modification recognizes that the rules for grammars are usually divided into two classes; those which describe the syntactic structure of a sentence (s, np, vp) and those which define individual lexical items (n, iv, tv, det, terminator). The latter group defines, in effect, the lexicon of individual words and their properties that form the vocabulary of a language. The following provides a useful convention for making individual entries in the lexicon:

For every word category (like n), and set of rules of the form (n --> [word]), define a new rule of the form

n(n(N) --> [N], {n(N)}.

and a set of facts of the form:

n(word).

For instance, the above grammar has two nouns, giraffe and apple, defined by the set of rules:

n --> [giraffe].
n --> [apple].

This set can be replaced by the following new rule and two facts:

n(n(N) --> [N], {n(N)}.
n(giraffe).
n(apple).

Now if we want to add more nouns to this grammar, we simply add more facts to the two already there.

Acomplete revision of the above grammar that accommodates these two refinements is given below (this one's called "grammar3" in the course directory).

```
    s(s(NP,VP)) --> np(NP), vp(VP), terminator.

    np(np(D,N)) --> det(D), n(N).

    vp(vp(IV)) --> iv(IV).
    vp(vp(TV,NP)) --> tv(TV), np(NP).

    det(det(D)) --> [D], {det(D)}.
    det(the).
    det(a).

    n(n(N)) --> [N], {n(N)}.
    n(giraffe).
    n(apple).

    iv(iv(IV)) --> [IV], {iv(IV)}.
    iv(dreams).

    tv(tv(TV)) --> [TV], {tv(TV)}.
    tv(dreams).
    tv(eats).

    terminator --> ['.'] ; ['?'] ; ['!'].
```

Exercises

1. Using one of the programs `grammar1` or `grammar2`, along with the read_sent program as an input aide, determine whether or not each of the following is a valid sentence:
   a. The giraffe eats the apple.
   b. The apple eats the giraffe.
   c. The giraffe eats.

2. Suggest a small change in the grammar that would make all three valid sentences.
3. Exercise `grammar3a` along with read_sent to obtain parse trees for each of the valid sentences in question 1.
4. In the style of `grammar3a`, define a new Prolog grammar that will implement the language defined in Figure 2. That is, ambiguous sentences like `fruit flies like an apple` should yield two different parse trees. Exercise your grammar to be sure that it generates both parse trees for this sentence.
5. Consider the sentence "the woman saw the man on the hill with the telescope."
   a. Briefly discuss the different ways in which this sentence can be interpreted (i.e., the sentence is syntactically ambiguous).
   b. Define an ambiguous context free grammar which allows these different interpretations to be realized.
   c. Design a Prolog grammar that is equivalent to your grammar in part b, and exercise it to be sure that these interpretations occur as different parse trees for the given sentence.
6. (optional) If you know Prolog well, consider writing functions that will display the output of `grammar3` in a more tree-like, readable fashion. That is, the result

   ```
   s(np(det(the), n(giraffe)), vp(iv(dreams)), terminator('.'))
   ```
   would be displayed as follows:
   ```
       s
           np
               det(the)
               n(giraffe)
           vp
               iv(dreams)
           terminator('.')
   ```

References

1. Clocksin and Mellish, *Programming in Prolog* 4e, Springer, 1997.
2. Matthews, Clive, *An Introduction to Natural Language Processing through Prolog*, Longman, 1998.

Modified: 19-Jul-1999

Skip Navigation and go to content

You may be using a browser that will cause viewing problems on our web site... please visit our browser upgrade page to learn more.

Bowdoin

Computer Science

Writing and Running Prolog Programs: A Quick Tutorial

You can run Prolog on any of the Linux machines. The version of Prolog that we are using at Bowdoin is "SWI Prolog." A reference manual for the full implementation is available on-line.

Starting and Stopping Prolog

Use the command "pl" to start the Prolog interpreter, which will give you the "?-" prompt (following a few lines of copyright information):

```
[allen@lynx04 allen]$ pl
Welcome to SWI-Prolog (Version 5.0.9)
Copyright (c) 1990-2002 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is
free software and you are welcomes to redistribute it
```

```
  under certain conditions.

  For help, use ?- help(Topic). or ?- apropos(Word).

  ?-
```

From within Emacs, you can either use the "M-x run-prolog" command or start a shell (M-x shell) and type the command "pl". You may choose to split your emacs window into two halves (ctrl-x 2), and edit the text of your Prolog program in the top half while running Prolog in the bottom half.

Loading files of Prolog function definitions has the same effect as typing them directly to the Prolog interpreter. For example, to load the file named "diff" that contains the Prolog function "d" you should use the command:

```
  consult(diff).
```

To exit Prolog, type `ctrl-d` .

Loading and Executing Code

You typically type your Prolog facts and rules into a file, and then use the "consult" command to cause that file to be loaded into Prolog, asserting each of those facts and rules. Then you can pose queries to the Prolog interpreter in the form of assertions containing variables, and it will try to find answers for you.

Here are some general guidelines for writing Prolog programs:

1. Identifiers that begin with a capital letter or an underscore character are taken as variables; all other identifiers are treated as constants.
2. There must be no space between the name of the predicate and the left parenthesis that opens its argument list.
3. All facts, rules, and queries must end with a period.
4. Any code file to be consulted MUST END WITH A CARRIAGE RETURN! Consulting a file that does not have a final carriage return causes Prolog to die.

Thus, for example, suppose we have the following code file named "teaches.pl":

```
teaches(chown, cs, 231).
teaches(chown, cs, 101).
teaches(garnick, cs, 101).
teaches(garnick, cs, 360).
teaches(tucker, cs, 220).
teaches(barker, math, 171).
exalted_guru(Prof) :-
        teaches(Prof, cs, Course).
```

Then we can do the following in Prolog:

```
1 ?- consult(teaches).
[WARNING: (/home/allen/cs250/prolog/teaches:7)
        Singleton variables: Course]
teaches compiled, 0.00 sec, 1,504 bytes.
```

```
Yes
```

The "yes" says Prolog successfully loaded the file, so we can now go ahead and pose queries:

```
2 ?- teaches(X, cs, 101).

X = chown

Yes
3 ?- teaches(chown, Dept, Course).

Dept = cs
Course = 231 ;

Dept = cs
Course = 101 ;

No
4 ?- exalted_guru(Prof).

Prof = chown ;

Prof = chown ;

Prof = garnick ;

Prof = garnick ;

Prof = tucker ;

No
```

The first query asks who teaches cs 101, since cs and 101 are both constants and X is a variable. The second query asks for the Dept and Course values for courses chown teaches. The system finds the first match, cs 231, and then waits. Typing a carriage return at that point brings you back to the prompt, as in the previous example. However, typing a semicolon asks the system to find another instantiation for the variables that would make the query true. After finding two answers, the system reponds "no", that it could find no others. The exalted_guru query looks for all bindings to Prof for which exalted_guru can be proved using to the rules in the file; note that it finds chown and garnick twice, since there are two different proofs in each case.

If there is a syntax error in the file, the system will give you an error message when you try to consult it. For example, here is what happens if we had typed "teaches (barker, math, 171).", leaving a space before the left parenthesis:

```
  5 ?- consult(teaches).
  [WARNING: (/home/allen/cs250/prolog/teaches:6)
          /home/allen/cs250/prolog/teaches:6: Syntax error:
Operator expected]
  [WARNING: (/home/allen/cs250/prolog/teaches:7)
          Singleton variables: Course]
  teaches compiled, 0.00 sec, -72 bytes.

  Yes
```

The same sort of thing happens for syntax errors in interactive queries:

```
2 ?- teaches (Prof, cs, Course).

[WARNING: Syntax error: Operator expected
teaches (Prof, cs, Course
** here **
) . ]
2 ?-
```

In this case, the system reprints the prompt, so that you can retype the query correctly.

Consulting is the normal way to load a file. If the filename has any funny characters (like periods) in it, use single quotes around it in the call to consult:

```
4 ?- consult('family.pro').
```

Lists

Lists are notated in Prolog using square brackets, with the elements divided using commas. Thus, [1,3,5,7] is a list of the first four odd numbers. The empty list is []. A vertical bar is used to mark the tail of the list (that's "cdr" in Lisp or Scheme). First, last, and member come out as follows:

```
first([Head|Tail], Head).
```

```
last([Item], Item).
last([Head|Tail], Item) :- last(Tail, Item).

member(Head, [Head|Tail]).
member(Item, [Head|Tail]) :- member(Item, Tail).
```

The following function is true of all lists with even numbers of elements:

```
even_length([]).
even_length([First,Second|Rest]) :- even_length(Rest).
```

Arithmetic

Prolog provides arithmetic operators in either prefix or infix form. Thus

```
?- <(3,6)
yes
?- 3<6.
yes
?-
```

However, note the following:

```
?- =(3+7,2+8).
no
?- =(+(3,7), X).
X = 3 + 7
```

```
yes
?-
```

3+7 does not equal 2+8 because this is not Lisp, and those arguments there that look like nested function calls are not evaluated before the equal test is made. The second example shows this, as X is bound to the expression 3+7, and not to 10.

The infix predicate "is" can be used to force evaluation:

```
?- X is 3+7.
X = 10
yes
?-
```

The "is" predicate requires that all variables on the right hand side be instantiated:

```
?- 10 is 3+X.
[WARNING: Arguments are not sufficiently instantiated]
 ^ Exception: (  7) 10 is 3+_G143 ?
```

This may seem restrictive, but consider "9 is X*Y.", which has an infinite number of answers, to see the kinds of problems one could get into.

Length of a list can thus be defined as follows:

```
length([], 0).
length([Head|Tail], Length) :-
                length(Tail, Tail_length),
                Length is Tail_length + 1.
```

A minor point: in order to keep => and <= free for use as arrows, Prolog uses =< for less-equals and >= for greater-equals.

Tracing and Break

The predicates trace and untrace are used to turn on and off the tracing of other predicates. Because there can be independent predicates with same name and different arities (numbers of arguments), trace expects the names of predicates to be followed by a slash and their arities. For example, with the predicate factorial defined as follows:

```
factorial(0, 1).
factorial(N, Res) :-
         N > 0,
         M is N - 1,
         factorial(M, SubRes),
         Res is N * SubRes.
```

we can do this:

```
17 ?- trace(factorial/2).
        factorial/2: call redo exit fail

Yes
18 ?- factorial(4,X).
```

```
T Call:  (  7) factorial(4, _G173)
T Call:  (  8) factorial(3, _L131)
T Call:  (  9) factorial(2, _L144)
T Call:  ( 10) factorial(1, _L157)
T Call:  ( 11) factorial(0, _L170)
T Exit:  ( 11) factorial(0, 1)
T Exit:  ( 10) factorial(1, 1)
T Exit:  (  9) factorial(2, 2)
T Exit:  (  8) factorial(3, 6)
T Exit:  (  7) factorial(4, 24)

X = 24
```

In the first call, the first argument is bound to 4, and the second is bound to an anonymous variable (_G173), and that applies down through the call with 0, whose second argument finally gets bound to 1, and then the recursion unwinds, doing the multiplications.

The predicate listing, as in "listing(factorial/2)." will print out all the current facts and rules for the argument predicate:

```
25 ?- listing(factorial/2).

factorial(0, 1).
factorial(A, B) :-
        A>0,
        C is A-1,
        factorial(C, D),
        B is A*D.
```

```
Yes
```

I/O and Strings

The predicate read(X) will read a term (everything up to a period followed by a cr or space) from the standard input and unify X with it.

The predicate write(X) writes out the current value of X to the current output stream, and nl generates a new line.

Strings (notated with double quotes) are stored as lists of the ASCII values of the characters:

```
?- "A string" = X.
X = [65,32,115,116,114,105,110,103]
yes
```

That format allows code to examine or alter the characters in the string.

Single quotes surrounding a sequence of characters are read as a constant whose name is that sequence of characters. Thus while X is a variable, 'X' is a constant:

```
?- X = 3.
X = 3
yes
?- 'X' = 3.
```

```
no
```

Since write('Mary') produces Mary, while write("Mary") produces [77,97,114,121], names are used in many cases were strings would be used in other languages.

The predicate name/2 is used to convert between names and strings:

```
?- name('Mary', X).
X = [77,97,114,121]
yes
?- name(X, [77,97,114,121]).
X = Mary
yes
```

The following code will repeatedly input numbers and output their cubes:

```
cube :-
        write('Next item, please: '),
        read(X),
        process(X).

process(stop) :- !.

process(N) :-
        C is N * N * N,
        write('Cube of '), write(N), write(' is '),
        write(C), nl,
        cube.
```

A run of this program looks like this:

```
?- cube.
Next item, please: 3.
Cube of 3 is 27
Next item, please: 4.
Cube of 4 is 64
Next item, please: 5.
Cube of 5 is 125
Next item, please: stop.
yes
```

Note that each input must be followed by a period, to mark the end of a term.

SWI-Prolog 4.0.11 Reference Manual

Jan Wielemaker
Dept. of Social Science Informatics (SWI)
Roeterstraat 15, 1018 WB Amsterdam
The Netherlands
Tel. (+31) 20 5256121

Abstract

SWI-Prolog is a Prolog implementation based on a subset of the WAM (Warren Abstract Machine). SWI-Prolog was developed as an *open* Prolog environment, providing a powerful and bi-directional interface to C in an era this was unknown to other Prolog implementations. This environment is required to deal with XPCE, an object-oriented GUI system developed at SWI. XPCE is used at SWI for the development of knowledge-intensive graphical applications.

As SWI-Prolog became more popular, a large user-community provided requirements that guided its development. Compatibility, portability, scalability, stability and providing a powerful development environment have been the most important requirements. Edinburgh, Quintus, SICStus and the ISO-standard guide the development of the SWI-Prolog primitives.

This document gives an overview of the features, system limits and built-in predicates.

About this document

This manual is written and maintained using LaTeX . The LaTeX source is included in the source distribution of SWI-Prolog. The manual is converted into HTML using a converter distributed with the SWI-Prolog sources. From the same source we generate the plain-text version and index used by the online help system (located in the file `MANUAL` in the library directory) as well as the PDF version. Sources, binaries and documentation can be downloaded from the SWI-Prolog download page.

The SWI-Prolog project **home page** is located here

Copyright © 1990-- 2001 , University of Amsterdam

Word Features and Agreement Issues

Feature Systems for English

Within a sentence, several additional structural requirements must be fulfilled, in addition to its basic structural integrity suggested by the grammar. These requirements include person and number agreement between and within noun and verb phrases, proper use of verb forms, appropriate phrase structures following verbs, and proper use of prepositional phrases with various verbs.

Many of these requirements can be ensured by adding so-called "features" to the lexicon, and then augmenting the grammar rules so that the features (like number and person) play a role in discriminating between correct and incorrect sentences.

Number and Person Agreement

The *number* feature of a noun or noun phrase is either singular or plural. Correspondingly, verbs and verb phrases must agree in number with the noun phrase that is the subject of the sentence. For instance,

```
The man sees the fish.
```

is correct, but not

```
The man see the fish.
```

Alongside number is the *person* feature of noun and verb phrases. Each of the above sentences is in the third person, since its subject is neither the speaker nor the hearer. When the subject of the sentence is the speaker, it is expressed in the first person, as in

```
I see the fish.
```

When the subject is the hearer, it is identified as the second person, as in

```
You see the fish.
```

Noun phrase-verb phrase agreement takes person into account as well as number.

A Simple Grammar with Number Agreement

Here is a simple grammar that forces number agreement between the noun and verb phrases of a sentence. A copy is available as `grammar4` in the `prolog` directory. In this program, singular and plural number are denoted by the atoms `s` and `p`, respectively.

```
s(s(NP,VP)) --> np(NP, Number), vp(VP, Number), terminator.

np(np(D,N), Number) --> det(D, Number), n(N, Number).
np(np(P), Number) --> pro(P, Number).

vp(vp(IV), Number) --> iv(IV, Number).
vp(vp(TV,NP), Number) --> tv(TV, Number), np(NP, _).

det(det(W), Number) --> [W], {det(W, Number)}.
det(the, _).
det(a, s).

n(n(W), Number) --> [W], {n(W, Number)}.
n(dog, s).
n(fish, _).
n(man, s).      n(men, p).
n(saw, s).

pro(pro(W), Number) --> [W], {pro(W, Number)}.
pro(he, s).

iv(iv(W), Number) --> [W], {iv(W, Number)}.
iv(cries, s).   iv(cry, p).

tv(tv(W), Number) --> [W], {tv(W, Number)}.
tv(sees, s).    tv(see, p).
tv(wants, s).   tv(want, p).
tv(was, s).     tv(were, p).

terminator --> ['.'] ; ['?'] ; ['!'].
```

This grammar has some interesting characteristics. First, the number feature is prominently carried in rules where it is needed to enforce number agreement among nouns, verbs, determiners, noun phrases, verb phrases, and sentences. Recall the Prolog convention that several occurrences of a variable, like `Number`, within a single rule must all instantiate to the *same* value (s or p in this case) whenever that rule is used in a parse.

Second, some words in the lexicon (e.g., `the` and `fish`) can represent associate with both singular and plural number, and this is identified by _ (don't care) number entries for those words in the lexicon.

Third, the grammar doesn't take person into account, as it should in reality. This extension is left as an exercise.

Verb Forms and Verb Subcategorization

In addition to person and number agreement, sentences may be expressed in different tenses and may have different constructions following the verb to express various kinds of events. Tense is expressed by various codings for *verb forms*, including the following (Allen, p 87):

`base` form - e.g., see
`pres` (simple present) - e.g., The dog *sees* the fish.
`past` - e.g., The dog *saw* the fish.
`ing` (present participle) - e.g., The dog is *seeing* the fish.
`pastprt` (past participle) - The dog has *seen* the fish
`inf` (infinitive) - The dog wants *to see* the fish

Different constructions that follow the main verb in a sentence are sometimes defined using "verb subcategorization," as in the following (Allen, p 88):

`_none` (intransitive verbs) - The dog smiles.
`_np` (simple transitive) - The dog sees *the fish*.
`_np_np` - The dog gave *the man the fish*.
`_vp:inf` - The dog wants *to cry*.
`_np_vp:inf` - The dog wants *the fish to go*.
`_vp:ing` - The dog keeps *hoping* for the fish.
`_np_vp:ing` - The dog saw *the fish swimming* in the water.
`_np_vp:base` - The dog saw the *fish swim*.

Add the idea of having prepositional phrases in three general classes:

`to` - e.g., to
`loc` (location) - e.g., in, on, under, beside, ...

`mot` (motion) - e.g., to, toward, from, along, ...

And we can see the following additional verb subcategorizations:

`_np_pp:to` - The dog gave *the fish to the man*.
`_pp:loc` - The dog is *in the water*.
`_np_pp:loc` - The dog put t*he fish in the water*.
`_pp:mot` - The dog went to the water.
`_np_pp:mot` - The dog took *the fish to the man*.
`_adjp` - The dog is *happy*.
`_np_adjp` - The man kept *the dog happy*.
`_s:that` - Jack believed *that the dog took the fish*.
`_s:for` - Jack hoped *for the dog to take the fish*.

Using A Lisp Interpreter for Grammar and Lexicon

To take advantage of the NLP ideas in the Allen text, we switch from a Prolog-based to a Lisp-based style of expressing and exercising grammars and lexicons. Allen provides an interpreter tool for the grammars and lexicons discussed in that text, which are all provided in electronic form in the directory `jallen/Parser1.1`. (You should copy the directory `jallen` to your own directory, for later use throughout this course.) This tool is invoked using the following Lisp commands (assuming you have started Adobe Common Lisp out of your subdirectory `jallen/Parser1.1`):

```
> (load "LOADP")
```

Now to load and run the grammar and lexicon in Chapter 4, type the command:

```
> (loadChapter4)
```

and you are ready to parse and examine the structure of sentences that are discussed there.
The lexicon design is discussed on pages 90-93 of Allen. Each entry in the lexicon is coded using a Lisp-like notation according to its category, root, person and number agreement, and (in the case of verbs) form and subcategorization features. For instance, the word `dog` appears in the lexicon as follows:

```
(dog (n (root DOG1) (agr 3s))
```

which says that `dog` is a noun (`n`), has root form `DOG1` (which identifies a particular use, or sense, of the word dog), and has third person singular (3s) agreement features. The word saw has two different entries in the lexicon, one of which is:

```
(saw (v (root SEE1) (VFORM past) (subcat _np) (agr ?a)))
```

This identifies the word as a verb (v), with root SEE1, verb form (tense) past, with subcategorization _np (taking a noun phrase as an object), and requiring number and person agreement (agr ?a).
The grammar also takes a Lisp-like form, as discussed on pp 94-98 of Allen. As in our Prolog grammars, these grammars carry agreement and subcategorization features, along with the definition of the grammatical classes and syntactic structure. For instance, the rule

```
(NP AGR ?a) --> (ART AGR ?a) (N AGR ?a)
```

describes a noun phrase with a particular person-number agreement feature as an article and a noun with the same agreement feature. This is encoded into the following interpretable Lisp expression:

```
((np) -2> (art (agr ?a)) (head (n (agr ?a))))
```

Here, the special convention -n> encodes the arrow of the nth grammar rule in a way that uniquely identifies that rule for the purpose of tracing.

A Simple Grammar and Lexicon with Verb Subcategorization Features

The grammar and lexicon shown in Figures 4.6 and 4.7 utilize the person, number, verb form, and verb subcategorization features summarized above. They are encoded in Lisp notation in the file jallen/Parser1.1/Grams/chapt4. Here is a complete listing of that lexicon encoding.

```
(setq *lexicon4-6*
  '((a (art (agr 3s) (root A1)))
    (be (v (root BE1) (vform bare) (subcat (? s _adjp _np)) (irreg-
pres +)
           (irreg-past +)))
    (cry (v (root CRY1) (vform bare) (subcat _none)))
    (dog (n (root DOG1) (agr 3s)))
    (fish (n (root FISH1) (agr (? a 3s 3p)) (IRREG-PL +)))
    (happy (adj (subcat _vp-inf) (root HAPPY1)))
    (he (pro (root HE1) (AGR 3s)))
    (is (v (root BE1) (VFORM pres) (SUBCAT (? s _adjp _np)) (AGR 3s)))
    (Jack (name (agr 3s) (root JACK1)))
    (man (n (root MAN1) (agr 3s)))
    (men (n (root MAN1) (agr 3p)))
    (saw (n (root SAW1) (agr 3s)))
    (saw (v (root SAW2) (vform bare) (subcat _np)))
    (saw (v (root SEE1) (VFORM past) (subcat _np) (agr ?a)))
    (see (v (root SEE1) (VFORM bare) (subcat _np) (irreg-past +)
         (en-pastprt +)))
    (seed (n (root SEED1) (AGR 3s)))
    (the (art (root THE1) (agr (? a 3s 3p))))
    (to (to (vform inf)))
    (want (v (root WANT1) (VFORM bare)
             (subcat (? s _np _vp-inf _np_vp-inf))))
    (was (v (root BE1) (VFORM past) (AGR (? a 1s 3s))
            (SUBCAT (? s _adjp _np))))
    (were (v (root BE1) (VFORM past) (AGR (? a 2s 1p 2p 3p))
             (SUBCAT (? s _adjp _np))))
    (+s (+S))
    (+ed (+ED))
    (+en (+EN))
    (+ing (+ING))))
```

Here is a complete listing of the encoded grammar.

```
(setq *grammar4-7*
      '((headfeatures (s agr) (vp vform agr) (np agr))
        ((s (inv -))
          -1>
             (np  (agr ?a)) (head (vp (vform (? v past pres)) (agr
?a))))
         ((np)
           -2>
               (art (agr ?a))  (head (n (agr ?a))))
        ((np)
           -3>
             (head (pro)))
        ((vp)
           -4>
              (head (v (subcat _none))))
        ((vp)
           -5>
              (head (v (subcat _np))) (np))
        ((vp)
            -6>
               (head (v (subcat _vp-inf))) (vp (vform inf)))
        ((vp)
           -7>
```

```
            (head (v (subcat _np_vp-inf))) (np) (vp (vform inf)))
      ((vp)
         -8>
             (head (v (subcat _adjp))) (adjp))
      ((vp (vform inf))
         -9>
            (head (to)) (vp (vform bare)))
      ((adjp)
         -10>
           (head (adj)))
      ((adjp)
         -11>
            (head (adj (subcat _vp-inf))) (vp (vform inf)))))
```

Below is a trace of the parse of the sentence "He wants to be happy." shown in Figure 4.9, using this grammar and lexicon. It is obtained by the Lisp function call `(BU-parse '(he want +s to be happy))` followed by the function call `(show-answers)`.

```
S57:<S ((INV -) (AGR 3S) (1 NP50) (2 VP56))> from 0 to 6 from rule -1>
  NP50:<NP ((AGR 3S) (1 PRO44))> from 0 to 1 from rule -3>
    PRO44:<PRO ((LEX HE) (ROOT HE1) (AGR 3S))> from 0 to 1 from rule
NIL
  VP56:<VP ((VFORM PRES) (AGR 3S) (1 V52)
           (2 VP55))> from 1 to 6 from rule -6>
    V52:<V ((AGR 3S) (VFORM PRES) (ROOT WANT1) (SUBCAT _VP-INF) (1
V45)
           (2 +S46))> from 1 to 3 from rule -LEX1>
      V45:<V ((LEX WANT) (ROOT WANT1) (VFORM BARE)
             (SUBCAT
              (? S6 _NP_VP-INF _VP-INF
               _NP)))> from 1 to 2 from rule NIL
      +S46:<+S ((LEX +S))> from 2 to 3 from rule NIL
    VP55:<VP ((VFORM INF) (AGR -) (1 TO47)
             (2 VP54))> from 3 to 6 from rule -9>
      TO47:<TO ((LEX TO) (VFORM INF))> from 3 to 4 from rule NIL
      VP54:<VP ((VFORM BARE) (AGR -) (1 V48)
               (2 ADJP53))> from 4 to 6 from rule -8>
        V48:<V ((LEX BE) (ROOT BE1) (VFORM BARE) (SUBCAT _ADJP)
               (IRREG-PRES +)
               (IRREG-PAST +))> from 4 to 5 from rule NIL
        ADJP53:<ADJP ((1 ADJ49))> from 5 to 6 from rule -10>
          ADJ49:<ADJ ((LEX HAPPY) (SUBCAT _VP-INF)
                     (ROOT HAPPY1))> from 5 to 6 from rule NIL
```

This trace should be compared with the parse tree shown in Figure 4.9 on page 97 of Allen. Note that different levels of indentation here correspond with different levels of subtree in that figure. It's a bit tedious to unravel, but a close examination of the indentation structure reveals the structure of the parse tree itself. All derived feature values are attached to each different node of the tree, along with the identifying number of the grammar rule that generated that node.

Additional Requirements and Features

Exercises

1. Augment `grammar4` so that it takes person into account as well as number, in determining the grammaticality of its sentences. Consider adding a new grammatical category, pronoun, that permits the words i, you, he, she, it, we, and they to the language. Also add a new feature to the lexicon, person, which identifies the person of a noun or verb as 1, 2, or 3 according to its form. For instance, the pronoun she would be defined as `pro(she, s, 3),` the pronoun you would be defined as `pro(you, _, 2)`, and so forth.
2. Exercise the lexicon and grammar given in the file `chapt4`, using all the sentences given as examples in the above notes. Be sure to perform morphological analysis manually, so that sentences like "The dog wants to cry" can be correctly parsed. I.e., type the parsing command as `(BU-parse '(the dog want +s to cry))` rather than `(BU-parse '(the dog wants to cry))`. Which of these sentences do not parse correctly and why?
3. Augment the grammar and lexicon in the file `chapt4` so that various forms of `give` are included in the lexicon, allowing sentences like "The dog gave the man the fish" to be successfully parsed.
4. (Optional) Augment the grammar and lexicon in the file `chapt4` so that all the sentences given as examples in the above discussion can be successfully parsed.

Referenes

Allen, Chapter 4
Matthews, Chapter 11

Syntax of Various Sentence Forms

Auxiliary Verbs

The main auxiliary verbs in English are `be` and `have`, the *modal* verbs (`do`, `can`, `may`, `will`, `might`, `should`, `must`, `need`, and `dare`) and the form of `be` used in the passive (e.g., `being seen`). The ordering of the auxiliaries in a verb phrase is fairly simple:

```
(Modal) (Have) (Be)
```

which means that each part is optional, but modals always precede Have and Be, and Have always precedes Be. Here are some examples, using `see` as the main verb:

```
could have been seen
could have seen
can see
have seen
was seen
```

To enforce agreement in form among the auxiliaries and their main verb, the COMPFORM attribute is added to their entries in the lexicon, and then enforced in their entries in the grammar. Here is an example lexicon entry for the auxiliary verb `could`:

```
(could (aux (modal +) (root COULD1) (vform (? v pres past)) (agr ?a) (COMPFORM
bare)))
```

This says that `could` is a modal, can be used with a verb in the present or past tense (e.g., `could see` or `could have seen`), requires person and number agreement, and has as complement the base (`bare`) form of the main verb when used alone with it. A grammar rule that incorporates auxiliaries is illustrated below:

```
VP --> (AUX COMPFORM ?v) (VP VFORM ?v)
```

which is encoded as follows for the Allen interpreter:

```
((vp) -2> (head (aux (compform ?v))) (vp (vform ?v)))
```

This allows a verb phrase to be formed out of an auxiliary verb (like `could`) and a main verb, provided that the COMPFORM attribute of the auxiliary is the same as the VFORM of the main verb (like `bare`, in the verb phrase `could be`).

More examples of auxiliary verb coding and use in the lexicon and grammar are illustrated in Allen, pp 123-127.

Here is a listing of the lexicon for auxiliary verbs, given as the file `chapt5` in the `jallen/Parser1.1` directory.

```
(setq *lexicon5-2*
  '((can (aux (modal +) (root CAN1) (vform pres) (agr ?a) (COMPFORM bare))
can1)
    (could (aux (modal +) (root COULD1) (vform (? v pres past)) (agr ?a)
           (COMPFORM bare)))
    (do (aux (modal +) (root DO1) (vform pres) (agr (? a 1s 2s 1p 2p 3p))
        (COMPFORM bare)))
    (does (aux (modal +) (root DO1) (vform pres) (agr 3s) (COMPFORM bare)))
    (did (aux (modal +) (root DO1) (vform past) (agr ?a) (COMPFORM bare)))
    (have (aux (vform bare) (root HAVE-AUX) (COMPFORM pastprt)))
    (have (aux (vform pres) (root HAVE-AUX) (agr (?a 1s 2s 1p 2p 3p))
          (COMPFORM pastprt)))
    (has (aux (vform pres) (root HAVE-AUX) (agr 3s) (COMPFORM pastprt)))
    (had (aux (vform past) (root HAVE-AUX) (agr ?a) (COMPFORM pastprt)))
    (having (aux (vform ing) (root HAVE-AUX) (COMPFORM pastprt)))
    (be (aux (root BE-AUX) (VFORM bare) (COMPFORM -)))
    (is (aux (root BE-AUX) (VFORM pres) (COMPFORM -) (AGR 3s)))
    (am (aux (root BE-AUX) (VFORM pres) (COMPFORM -) (AGR 1s)))
    (are (aux (root BE-AUX) (VFORM pres) (COMPFORM -) (AGR (?a 2s 1p 2p 3p))))
    (was (aux (root BE-AUX) (VFORM past) (AGR (? a 1s 3s)) (COMPFORM -)))
    (were (aux (root BE-AUX) (VFORM past) (AGR (? a 2s 1p 2p 3p))
          (COMPFORM -)))
    (been (aux (root BE-AUX) (VFORM pastprt) (COMPFORM -)))
    (being (aux (root BE-AUX) (VFORM ing) (COMPFORM -)))))
```

Auxiliaries: a Prolog View

Here is another treatment of auxiliaries, from a Prolog point of view. This will be useful later when we discuss the semantics of sentences using a Prolog style of representation. This discussion comes from Pereira, p 114ff.

In this discussion, we identify the idea of a *finite* verb as one which is a complete verb phrase, like "halts," "halted," "writes a program," "is halting," or "has been halting." A *nonfinite* verb refers to a verb's base form, such as "halt." We also distinguish *infinitival* forms, like "to halt," *present participles*, like "halting," and *past participles*, like "halted." Now the lexicon for verbs can be encoded in the following way:

```
iv(Form) --> [IV], {iv(IV, Form)}.
iv(halts, finite).
iv(halt, nonfinite).
iv(halting, present_participle).
iv(halted, past_participle).

aux(Form) --> [Aux], {aux(Aux, Form)}.
aux(could, finite/nonfinite).
aux(have, nonfinite/past_participle).
aux(has, finite/past_participle).
aux(been, past_participle/present_participle).
aux(be, nonfinite/present_participle).
```

Now verb phrases can be formed with or without auxiliaries, using the following rules:

```
vp(Form) --> iv(Form).
vp(Form) --> tv(Form), np.
vp(Form) --> aux(Form/Require), vp(Require).
```

The above rules cover various kinds of verb phrases. For instance, the auxiliary verb `been` is a past participle and combines with a present participle, and `have` is nonfinite and takes a past participle when used as an auxiliary.

Thus, "have been halting" is a verb phrase with the auxiliary `Form = nonfinite` (have) and `Require = past_participle` (been halting). Taking the next step, the verb phrase "been halting" responds to the same grammar rule, with the auxiliary `Form = past_participle` (been) and `Require = present_participle` (halting). Finally, the verb phrase "halting" is an intransitive verb (using the first rule) with `Form = present_participle`.

Passives

Most verbs that include an NP in their complement (i.e., are transitive) allow the passive form as well. For instance, the active verb phrase in the sentence

```
Jack can see the dog.
```

has a passive form in sentences like the following:

```
The dog was seen.
```

To account for passives forms, grammars utilize the notion of a "passive gap", which is just a placeholder for an object that would normally complement a transitive verb. To explain passive verb phrases like the one above, the grammar is augmented with rules like the following:

```
VP[+pass] --> AUX[be] VP[pastprt, main, +passgap]
VP{+passgap, +main] --> V[_np]
```

That is, a verb phrase that is passive can be formed using the auxiliary `be`, followed by a verb phrase whose main verb is a past participle and then a passive gap. A verb phrase that is a main verb and has a passive gap can be any transitive verb (i.e., any verb that has the feature `_np`). These two rules are encoded as follows:

```
((vp (PASS +)) -5>
         (head (aux (root BE-AUX))) (vp (vform pastprt) (MAIN +) (PASSGAP +)))
((vp (PASSGAP +)  (MAIN +)) -8>
         (head (v (subcat _np))))
```

Here is an encoding of the complete grammar shown in Figure 5.3 (page 127) of Allen, including rules like the ones discussed above. This grammar defines several different verb phrase structures, each allowing different combinations of auxiliary verbs and passive forms.

```
(setq *grammar5-3*
      '((headfeatures
         (s vform agr)
         (vp vform agr)
         (np agr))
        ((s (inv -))
         -1>
         (np (agr ?a)) (head (vp (vform (? v pres past)) (agr ?a))))
        ((vp)
         -2>
         (head (aux (compform ?v))) (vp (vform ?v)))
        ((vp)
         -3>
         (head (aux (root BE-AUX))) (vp (vform ing) (MAIN +)))
        ((vp)
         -4>
         (head (aux (root BE-AUX))) (vp (vform ing) (PASS +)))
        ((vp (PASS +))
         -5>
         (head (aux (root BE-AUX))) (vp (vform pastprt) (MAIN +) (PASSGAP +)))
        ((vp (PASSGAP -) (MAIN +))
         -6>
         (head (v (subcat _none))))
        ((vp (PASSGAP -)  (MAIN +))
         -7>
         (head (v (subcat _np))) (np))
        ((vp (PASSGAP +)  (MAIN +))
         -8>
         (head (v (subcat _np))))
        ((np)
          -9>
          (art (agr ?a)) (head (n (agr ?a))))
         ((np)
          -10>
          (head (name)))
         ((np)
          -11>
          (head (pro)))))
```

Below is a parse of the sentence "The dog was seen" which shows the roles of the various grammar rules for handling passive gaps. A simplified tree diagram of this parse is shown in Figure 5.4 of Allen (page 128).

```
REL181:<REL ((GAP -) (1 S180))> from 0 to 3 from rule -R5>
```

```
  S180:<S ((GAP <NP ((SEM ?SEM177) (AGR ?AGR176))>) (WH -) (INV -)
           (VFORM PAST) (AGR 3S) (1 NP175)
           (2 VP179))> from 0 to 3 from rule -5-8-1>
    NP175:<NP ((GAP -) (WH -) (AGR 3S) (1 DET173)
               (2 CNP174))> from 0 to 2 from rule -5-7-2>
      DET173:<DET ((GAP -) (AGR 3S)
                   (1 ART167))> from 0 to 1 from rule -5-7-5>
        ART167:<ART ((LEX THE) (ROOT THE1)
                     (AGR (? A5 3P 3S)))> from 0 to 1 from rule NIL
      CNP174:<CNP ((GAP -) (AGR 3S)
                   (1 N168))> from 1 to 2 from rule -5-7-3>
        N168:<N ((LEX DOG) (ROOT DOG1)
                 (AGR 3S))> from 1 to 2 from rule NIL
    VP179:<VP ((GAP <NP ((SEM ?SEM177) (AGR ?AGR176))>) (VFORM PAST)
               (AGR 3S) (1 V170)
               (2 GAP178))> from 2 to 3 from rule -5-8-7>
      V170:<V ((LEX WAS) (ROOT BE1) (VFORM PAST) (AGR (? A7 3S 1S))
               (SUBCAT _NP))> from 2 to 3 from rule NIL
      GAP178:<NP ((EMPTY +) (GAP <NP ((SEM ?SEM177) (AGR ?AGR176))>)
                  (SEM ?SEM177)
                  (AGR ?AGR176))> from 3 to 3 from rule NP-GAP-INTRO
V191:<V ((VFORM PASTPRT) (ROOT SEE1) (SUBCAT _NP) (1 V171)
         (2 +EN172))> from 3 to 5 from rule -LEX5>
  V171:<V ((LEX SEE) (ROOT SEE1) (VFORM BARE) (SUBCAT _NP)
           (IRREG-PAST +) (EN-PASTPRT +))> from 3 to 4 from rule NIL
  +EN172:<+EN ((LEX +EN))> from 4 to 5 from rule NIL
REL196:<REL ((GAP -) (1 VP195))> from 3 to 5 from rule -R6>
  VP195:<VP ((GAP <NP ((SEM ?SEM193) (AGR ?AGR192))>) (VFORM PASTPRT)
             (AGR -) (1 V191) (2 GAP194))> from 3 to 5 from rule -5-8-7>
    V191:<V ((VFORM PASTPRT) (ROOT SEE1) (SUBCAT _NP) (1 V171)
             (2 +EN172))> from 3 to 5 from rule -LEX5>
      V171:<V ((LEX SEE) (ROOT SEE1) (VFORM BARE) (SUBCAT _NP)
               (IRREG-PAST +) (EN-PASTPRT +))> from 3 to 4 from rule NIL
      +EN172:<+EN ((LEX +EN))> from 4 to 5 from rule NIL
    GAP194:<NP ((EMPTY +) (GAP <NP ((SEM ?SEM193) (AGR ?AGR192))>)
                (SEM ?SEM193)
                (AGR ?AGR192))> from 5 to 5 from rule NP-GAP-INTRO
```

Gaps and Movement in Sentences

So far we have looked at simple declarative sentences. Effective grammars also need to handle a variety of other sentential forms in which some part of the simple declarative form has moved to a new position. Here are four types of movement identified by Allen (attributed to Baker, 1989):

wh-movement: move a wh-term to the beginning of a sentence to form a wh-question. E.g., "Which dogs did he see?"
topicalization: move a constituent to the beginning of a sentene for emphases. E.g., "That dog he never liked."

Adverb preposing: move an adverb to the beginning of a sentence. E.g., "Tomorrow, he will see the dog."
Extraposition: move certain NP complements to the end of the sentence. E.g., "A book was written about evolution."

Handling Questions and Relative Clauses

The notion of a "gap" is a more general one than that which is used to handle passive forms. It is very useful these kinds of sentences as well. Let's look at how gaps can be used to handle movement in certain kinds of questions, like the following:

```
Which dogs did he see?
```

Here, the gap follows the verb phrase, and the word "Which" is sometimes called a "filler" for the gap (that is, a word that gives license to the existence of a gap following a transitive verb). Often words such as `which` (e.g., who, what, where, etc.; sometimes called the "wh-words") are also used at the head of relative clauses, such as in

```
The dogs which he saw returned.
```

So the coding of words like which in the lexicon must allow these different uses. The feature WH is used for this purpose. Here is an encoding of the word `which` in the lexicon that distinguishes its use in a question from its use in a relative clause (see Allen Figure 5.6, page 135 for more discussion of these examples).

```
    (which (qdet (WH q) (root WHICH) (agr (? a 3s 3p))))
    (which (pro (WH r) (root WHICH) (agr (? a 3s 3p))))
```

The first encoding says that `which` can be used to introduce wh-questions, and the second says that it can be used to introduce relative clauses. Some corresponding grammar rules that can be used with the first of these two uses are as follows (the entire grammar is given in the file `lisp/jallen/Parser1.1/Grams/chap5`):

```
   ((s) -5-8-3>
        (np (wh q) (gap -) (agr ?a))
        (head (s (inv +) (gap (% np (agr  ?a))))))
   ((s (inv +) (wh ?w) (gap ?g)) -5-8-2>
        (head (aux (compform ?s) (agr ?a)))
        (np (wh ?w) (agr ?a) (gap -))
        (vp (vform ?s) (gap ?g)))
   ((np (wh ?w)) -5-7-2>
        (det (wh ?w) (agr ?a)) (head (cnp (agr ?a))))
   ((det (wh ?w)) -5-7-7>
        (head (qdet (wh ?w))))
```

The first rule says that a sentence can be constructed using a noun phrase of the WH variety (e.g., "which dogs") and a head of the inverted s variety (e.g., "did he see"). The second rule shows how an inverted s can be defined with a gapped vp. The third and fourth rules tell more about the structure of a np of the WH variety; that it can be a det of the qdet variety (e.g., which) followed by a complementary noun phrase ("dogs"). Agreement also appears in appropriate places, as does the location of the gap (e.g., following the transitive verb "see").

A full parse of the sentence "Which dogs did he see" appears below,. This corresponds to the chart parse shown and discussed on page 141 of Allen.

```
S218:<S ((VFORM PAST) (AGR 3S) (1 NP210)
         (2 S217))> from 0 to 6 from rule -5-8-3>
  NP210:<NP ((GAP -) (WH Q) (AGR 3P) (1 DET204)
             (2 CNP209))> from 0 to 3 from rule -5-7-2>
    DET204:<DET ((GAP -) (WH Q) (AGR 3P)
                 (1 QDET198))> from 0 to 1 from rule -5-7-7>
      QDET198:<QDET ((LEX WHICH) (WH Q) (ROOT WHICH)
                     (AGR (? A23 3P 3S)))> from 0 to 1 from rule NIL
    CNP209:<CNP ((GAP -) (AGR 3P)
                 (1 N208))> from 1 to 3 from rule -5-7-3>
      N208:<N ((AGR 3P) (ROOT DOG1) (1 N199)
               (2 +S200))> from 1 to 3 from rule -LEX7>
        N199:<N ((LEX DOG) (ROOT DOG1)
                 (AGR 3S))> from 1 to 2 from rule NIL
        +S200:<+S ((LEX +S))> from 2 to 3 from rule NIL
  S217:<S ((GAP <NP ((SEM ?SEM214) (AGR 3P))>) (WH -) (INV +)
           (VFORM PAST) (AGR 3S) (1 AUX201) (2 NP211)
           (3 VP216))> from 3 to 6 from rule -5-8-2>
    AUX201:<AUX ((LEX DID) (MODAL +) (ROOT DO1) (VFORM PAST) (AGR 3S)
                 (COMPFORM BARE))> from 3 to 4 from rule NIL
    NP211:<NP ((GAP -) (WH -) (POSS -) (AGR 3S)
               (1 PRO202))> from 4 to 5 from rule -5-7-1>
      PRO202:<PRO ((LEX HE) (ROOT HE1)
                   (AGR 3S))> from 4 to 5 from rule NIL
    VP216:<VP ((GAP <NP ((SEM ?SEM214) (AGR 3P))>) (VFORM BARE) (AGR -)
               (1 V203) (2 GAP215))> from 5 to 6 from rule -5-8-7>
      V203:<V ((LEX SEE) (ROOT SEE1) (VFORM BARE) (SUBCAT _NP)
               (IRREG-PAST +) (EN-PASTPRT +))> from 5 to 6 from rule NIL
      GAP215:<NP ((EMPTY +) (GAP <NP ((SEM ?SEM214) (AGR 3P))>)
                  (SEM ?SEM214)
                  (AGR 3P))> from 6 to 6 from rule NP-GAP-INTRO
```

Handling Gaps and Questions in Prolog

Prolog provides similar support for representing questions and other filler-gap situations in sentences. The simplest situation occurs with subject-auxiliary inversion, forming a yes-no question. For example, the sentence "the program could halt" can be turned into the question "could the program halt" by inverting the subject and the auxiliary verb could. This is characterized in the following grammatical rule (continung the grammar begun in the previous Prolog discussion).

```
sinv --> aux(finite/Required), np, vp(Required).
```

That is, an inverted sentence is formed by an auxiliary verb, followed by a noun phrase and a verb phrase that reflects the Required part of the auxiliary verb phrase.

Recall that a *gap* is part of a phrase missing from its usual location, and a *filler* is another phrase that stands for the missiing one. For instance, in "terry read every book that bertrand wrote", the filler is "that" and the gap occurs after the verb "wrote" which normally takes a noun phrase as an object. In Prolog, a gap is realized by omitting a noun phrase:

```
np(gap(np)) --> [].
```

Now a verb phrase that admits a gap can be formed from a transitive verb and a possibly-missing noun phrase:

```
vp(GapInfo) --> tv, np(GapInfo).
s(GapInfo) --> np(nogap), vp(GapInfo).
rel --> relpron, s(gap(np)).
```

That is, a relative clause is a relative pronoun followed by a sentence with a gap, as in "that bertrand wrote."

Wh-questions can be handled in Prolog using similar strategies. Questions like "who loves mary" and "who does mary love" are handled using the following rules, respectively:

```
q --> whpron, vp(nogap).
q --> whpron, sinv(gap(np)).
sinv(GapInfo) --> aux, np(nogap), vp(GapInfo).
```

How good are NLP systems in practice?

(Alshawi, 5) CLE (Core Language Engine, 1992) Structure: a comprehensive NLP system for English; four stages -- lexical analysis, morphology, syntactic alalysis, and semantic analysis. Additional disambiguation and contextual interpretation carried out in other phases -- sortal filtering, quatiifier scoping, reference and ellipsis resolution, and plausibility checking.

CLE performance in 1992. A sample of 1000 sentences taken at random from the Lancaster Oslo Bergen corpus of printed British English. Of these, 634 were analyzed successfully by the CLE -- that is parsed and produced at least one logical form for meaning. 67% of the 634 were estimated to be valid meaning representations.

Exercises

1. Consider the Prolog rules for auxiliary verbs. Show how the verb phrase "could have been halting" is parsed usng these rules.
2. Parse the question "who does mary love" using the Prolog grammars for questions and gaps that are given in this section.
3. Augment the Prolog grammatical rules with sufficient lexical and syntactic definitions that will allow them to be equivalent to the Lisp-based grammar discussed in this section. Run your grammar with the sentence "which dogs did he see" and similar sentences to check that your augmented grammar is correct.

References

Allen, Chapter 5
Matthews, Chapter 11
Alshawi, Hayan, *The Core Language Engine*, SRI International, 1992.

Statistical Methods for Ambiguity Resolution

Statistical methods have been developed (E.g., Charniak, 1993) that help select part of speech, word sense, and grammatical structure for syntactically ambiguous sentences. Here, we explore statistical methods for part of speech tagging and the use of probabilistic grammars in parsing.

Part of speech tagging

Statistical methods for reducing ambiguity in sentences have been developed by using large corpora (e.g., the Brown Corpus or the COBUILD Corpus) about word usage. The words and sentences in these corpora are pre-tagged with the parts of speech, grammatical structure, and frequencies of usage for words and sentences taken from a broad sample of written language. *Part of speech tagging* is the process of selecting the most likely part of speech from among the alternatives for each word in a sentence. In practice, only about 10% of the distinct words in a million-word cprpus have two or more parts of speech, as illustrated by the following summary for the complete Brown Corpus (DeRose 1988):

Number of Tags	Number of distinct words	Percent of total
1	35,340	89.6%
2	3,760	9.5
3	264	0.6
4	61	0.1
5	12	0.03
6	2	0.005
7	1	0.0025
Total 2-7	4,100	10.4%

The one word in the Brown corpus that has 7 different part of speech tags is "still." Presumably, those tags include at least *n*, *v*, *adj*, *adv*, and *conj*. In practice, there are a small number of different sets of tags (or "tagsets") in use today. For example, the tagset used by the Brown corpus has 87 different tags. In the following discussion, we shall use a very small 4-tag set, {*n*, *v*, *det*, *prep*} in order to maintain clarity of the ideas behind the statistical approach to part of speech tagging.

Given *n* or *v* as the two part of speech tags (*T*), for the word *W = flies*, the outcome is partially governed by looking at the probability of each choice, based on the frequency of occurrence of the given word in a large corpus of representative text.

Compare *prob(T = v | W = flies),* which reads "the probability of a verb (*v*) choice, given that the word is "flies," with *prob(T = n | W = flies).*

In statistics, the following is true for two independent events X and Y:

prob(X | Y) = prob(X & Y) / prob(Y).

That is, the probability of event X occurring, after knowing that Y has occurred, is the quotient of the probability of events X and Y both occurring and the probability of Y occurring alone. Thus,

prob(T = n | W = flies) = prob(T = n & W = flies) / prob(W = flies)

How are these probabilities estimated? The collected experience of using these words in a large corpus of known (and pre-tagged) text, is used as an estimator for these probabilities.

Suppose, for example, that we have a corpus of 1,273,000 words (approximately the size of the Brown corpus) which has 1000 occurrences of the word *flies,* 400 pre-tagged as nouns (*n*) and 600 pre-tagged as verbs (*v*). Then *prob(W = flies),* the probability that a randomly-selected word in a new text is *flies*, is 1000/1,273,000 = 0.0008. Similarly, the probability that the word is *flies* and it is a noun (*n*) or a verb (*v*) can be estimated by:

prob(T = n & W = flies) = 400/1,273,000 = 0.0003
prob(T = v & W = flies) = 600/1,273,000 = 0.0005

So *prob(T = v | W = flies) = prob(T = v & W = flies) / prob(W = flies)* = 0.0005/0.0008 = 0.625. In other words, the prediction that an arbitrary occurrence of the word *flies* is a verb will be correct 62.5% of the time.

The process of part of speech tagging for an entire sentence with *t* words relies not only on the relative frequencies of the sentence's words in a pre-tagged corpus, but also on the part(s) of speech of all the *other* words in the sentence. Let $T_1 ... T_n$ be a series of parts of speech for an *n*-word sentence with words $W_1 ... W_n$. To assign the most likely parts of speech for these words using statistical techniques, we want to find values for $T_1 ... T_n$ that maximizes the following:

$$prob(T_1...T_n \mid W_1...W_n) = \\ prob(T_1...T_n) * prob(W_1...W_n \mid T_1...T_n) / prob(W_1...W_n) \qquad (1)$$

For example, for the text "Fruit flies like a banana." we want to find part of speech tags $T_1 ... T_5$ for this sequence of n=5 words that maximizes:

$$prob(T_1...T_5 \mid \text{fruit flies like a banana}) = \\ prob(T_1...T_5) * prob(fruit\ flies\ like\ a\ banana \mid T_1...T_5) / prob(fruit\ flies\ like\ a\ banana)$$

where each part of speech tag T_i comes from the set {*n*, *v*, *prep*, *det*}.

Since this calculation would be very complex (it would require too much data), researchers use the following (reasonably good) estimate for $prob(T_1...T_n \mid W_1...W_n)$, using the following approximations:

$$prob(T_1...T_n) = \|_i prob(T_i \mid T_{i-1}) \qquad \text{for } i=1,...,n$$
$$prob(W_1...W_n \mid T_1...T_n) = \|_i prob(W_i \mid T_i) \qquad \text{for } i=1,...,n$$

(Read $\forall = \|_i$" as "approximates the product over *i*.") This particular method of estimation is called a *bigram* model, since each part of speech tag T_i is dependent only on the previous word's part of speech choice, T_{i-1}. In practice, both bigram and *trigram* models are used. Now, the problem is simplified to finding a series of choices $T_1 ... T_n$ that maximizes the following:

$$prob(T_1...T_n \mid W_1...W_n) = \|_i prob(T_i \mid T_{i-1}) * \|_i prob(W_i \mid T_i) \qquad \text{for } i=1,...,n \qquad (2)$$

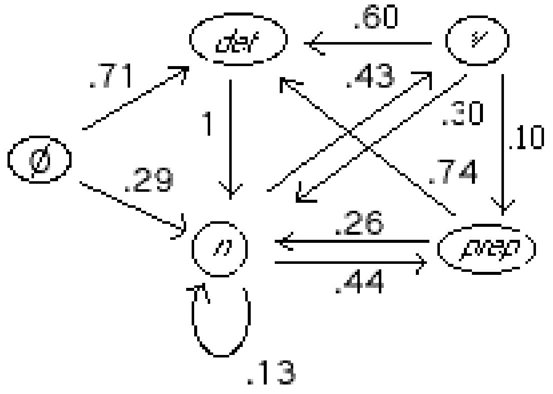
which is a much less complex calculation than equation (1) above. Notice here also that the denominator in (1) is dropped, since its value would be constant over all sequences of tags $T_1 ... T_n$.

For our example, we want to maximize the following:

$$prob(T_1...T_5 \mid fruit\ flies\ like\ a\ banana) = \|_i prob(T_i \mid T_{i-1}) * \|_i prob(Wi \mid Ti) \qquad \text{for i=1,...,5}$$

$$= prob(T_1 \mid \neg) * prob(T_2 \mid T_1) * prob(T_3 \mid T_2) * prob(T_4 \mid T_3) * prob(T_5 \mid T_4) * \\ prob(fruit \mid T_1) * prob(flies \mid T_2) * prob(like \mid T_3) * prob(a \mid T_4) * prob(banana \mid T_5)$$

Here, the symbol ¬ denotes the beginning of the sentence. These individual probabilities are readily computed from the frequencies found for the pre-tagged sentences in the corpus. That is, the probabilities *prob*(T_i | T_{i-1}) for each adjacent pair of parts of speech choice {*n*, *v*, *det*, *prep*}can be estimated by counting frequencies in the corpus and writing the result in the form of a "Markov chain," such as the one shown below (adapted from Allen, p 200):



This expresses, for instance, that the probability of having a verb (*v*) immediately following a noun (*n*) in a sentence is *prob(v | n)* = 0.43. In other words, 43% of the occurrences of a noun in the corpus are immediately followed by a verb.

With this information, we can estimate *prob(n n v det n)*, for instance, as:

$$\begin{aligned} prob(n\ n\ v\ det\ n) &= prob(n \mid Ø) * prob(n \mid n) * prob(v \mid n) * prob(det \mid v) * prob(n \mid det) \\ &= .29 * .13 * .43 * .6 * 1 \\ &= 0.00973 \end{aligned}$$

The corpus also provides estimates for the individual probabilites for each word's possible part of speech. Here is some sample data derived from a small corpus (Allen, 1995) for a few words of interest to us:

prob(a | det) = .036	*prob(a | n) = .001*	*prob(man | n) = .035*
prob(banana | n) = .076	*prob(like | v) = .1*	*prob(man | v) = .007*
prob(flies | n) = .025	*prob(like | prep) = .065*	*prob(hill | n) = .083*
prob(flies | v) = .07	*prob(like | n) = .013*	*prob(telescope | n) = .075*
prob(fruit | n) = .061	*prob(the | det) = .045*	*prob(on | prep) = .09*
prob(fruit | v) = .005	*prob(woman | n) = .076*	*prob(with | prep) = .09*

Now we can use equation (2) to calculate the likelihood of the sentence "fruit flies like a banana" being tagged with the parts of speech *n n v det n* as follows:

$$\begin{aligned}
&prob(n\ n\ v\ det\ n \mid fruit\ flies\ like\ a\ banana) \\
&\quad = prob(n\ n\ v\ det\ n) * prob(fruit \mid n) * prob(flies \mid n) * prob(like \mid v) * prob(a \mid det) * \\
&\qquad prob(banana \mid n) \\
&\quad = 0.00973 * 0.061 * .025 * .1 * .036 * .076 \\
&\quad = 0.00973 * 4.17 * 10^{-7} \\
&\quad = 4.06 * 10^{-9}
\end{aligned}$$

To find the *most likely* sequence of tags for a sentence, we need to maximize this calculation over all possible tag sequences for the sentence. This could be a complex computation if the "brute force" approach is used. That is, for an *n*-word sentence and *p* parts of speech, there are p^n different taggings. Even for our small sentence of *n*=5 words and p=4 different parts of speech {*n*, *v*, *det*, *prep*}, the number of calcuations would be 4^5 = 1,024. However, given our particular sentence, and the fact that several different taggings have 0 probability, there are at most 2*2*3*2*1 = 24 different non-zero calculations, as shown below:

fruit	*flies*	*like*	*a*	*banana*
n	*n*	*n*	*det*	*n*
v	*v*	*v*	*n*	
		prep		

Moreover, this number is reduced further by the fact that some of the transitions in the Markov chain are zero, such as *prob(v | v)* = 0. There are, practically speaking, only the following 8 different taggings for our sentence that give nonzero probabilities:

fruit flies like a banana

tagging = [n, n, n, n, n]
probability = 1.24795e-013

tagging = [n, n, v, n, n]
probability = 7.32753e-012

tagging = [n, n, v, det, n]
probability = 4.05833e-009

tagging = [n, n, prep, n, n]
probability = 4.22384e-012

tagging = [n, n, prep, det, n]
probability = 3.32909e-009

tagging = [n, v, n, n, n]
probability = 2.66722e-012

tagging = [n, v, prep, n, n]
probability = 8.89074e-012

tagging = [n, v, prep, det, n]
probability = 7.00738e-009

The probabilities calculated here are computed by the Prolog program `grammar11.pl`, whose implicit nondeterminism assists in the tedious search for alternative taggings and the accompanying calculations outlined above. Among these, the best alternatives suggest that *fruit flies* is a *n n* sequence or a *n v* sequence, leading to the following "most probable" taggings for the entire sentence.

> *prob(n n v det n | fruit flies like a banana)*
> *prob(n v prep det n | fruit flies like a banana)*

Furthermore, there is an algorithm called the *Viterbi algorithm*, which provides a strategy by which most of the non-zero alternatives that lead to nonzero probabilities can also be avoided. That is, the tagging with the maximum probability can be developed directly in a single pass on the words in the sentence, and not require the above calculation for all the 24 combinations of part of speech taggings listed above. (For the details of the Viterbi algorithm, see Allen, p 203).

The general results of this tagging method are quite positive (Allen p 203): "Researchers consistently report part of speech sentence tagging with 95% or better accuracy using trigram models."

Probabilistic grammars

Markov models can be extended to grammar rules to help govern choices among alternatives when the sentence is syntactically ambiguous (that is, there are two or more different parse trees for the sentence). This technique is called *probabilistic grammars*.

A probabilistic grammar has a probability associated with each rule, based on its frequency of use in the parsed version of a corpus. For example, the following probabilistic grammar's rules have their probabilities derived from a small corpus (Fig 7.17 in Allen).

Grammar rule	*prob*

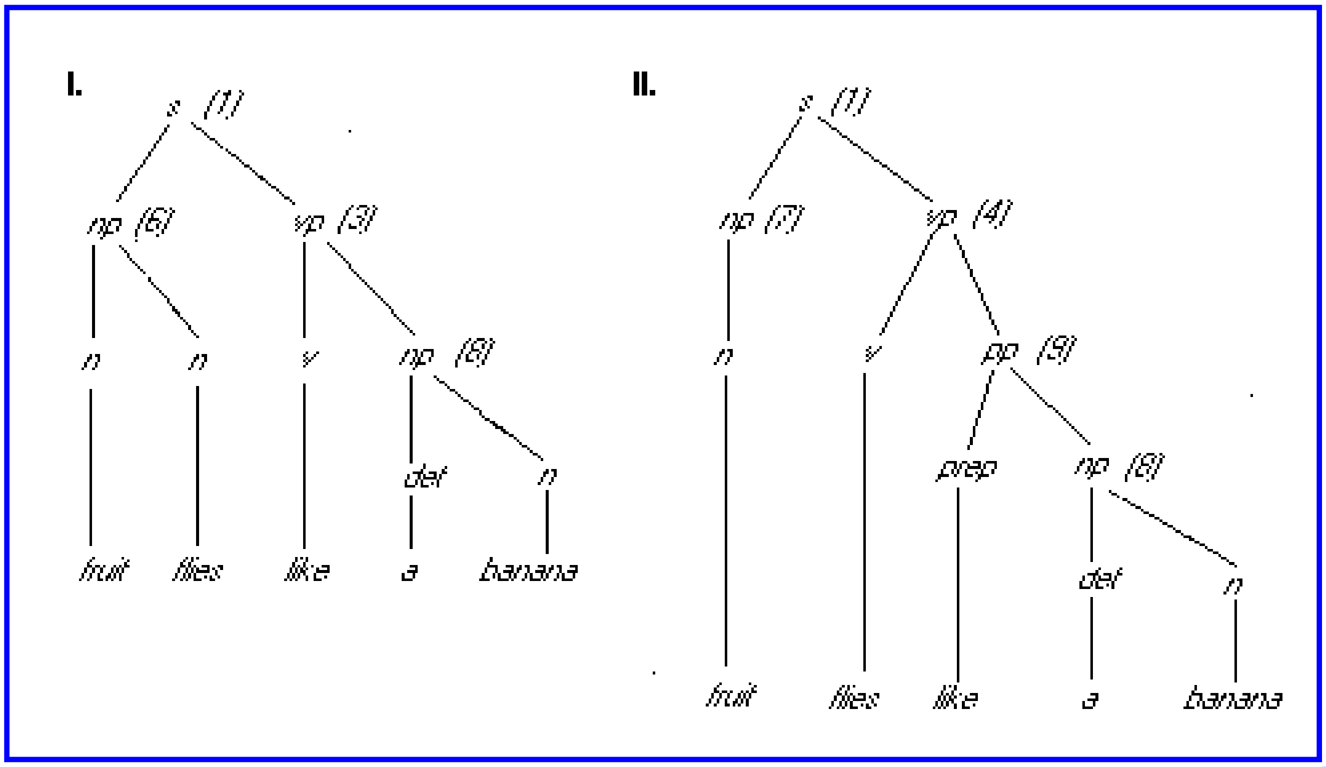
1. *s* --> *np vp*	1
2. *vp* --> *v np pp*	0.386
3. --> *v np*	0.393
4. --> *v pp*	0.22
5. *np* --> *np pp*	0.24
6. --> *n n*	0.09
7. --> *n*	0.14
8. --> *det n*	0.55
9. *pp* --> *p np*	1

For a given phrase like *a banana* and grammatical category (*np*), we look at all the possible subtrees that can be generated for that phrase, assigning appropriate parts of speech to the words. For example,

$$\begin{aligned} &prob(a\ banana \mid np) \\ &\quad = prob(rule\ 8 \mid np) * prob(a \mid det) * prob(banana \mid n) \\ &\quad\quad + prob(rule\ 6 \mid np) * prob(a \mid n) * prob(banana \mid n) \\ &\quad = .55 * .036 * .076 + .09 * .001 * .076 \\ &\quad = 1.51 * 10^{-3} \end{aligned}$$

(Note that only rules 6 and 8 come into play here, since they are the only ones that generate two-word noun phrases.) Here, we are using the same conditional probabilities for individual words, like *prob(banana | n),* that we used in the previous section for part of speech tagging.

For an entire sentence, we generate all of its possible parse trees, and then compute the probability that each of those trees will occur, using the above strategy. The preferred parse tree is the one whose probability is the maximum. Consider, for example, the two parse trees for the sentence *fruit flies like a banana* that are generated by the above grammar:



These trees are annotated with the numbers of the grammar rules that are used at each level. Thus, to determine which is the more probable parse for this sentence, we compute the following for each tree:

prob(fruit flies like a banana | s)

and choose the maximum value from these two computations. Here is a summary of the necessary calculations:

I. *prob(fruit flies like a banana | s)*
*= prob(rule 1 | s) * prob(fruit flies | np) * prob(like a banana | vp)*
*= 1 * prob(rule 6 | np) * prob(fruit | n) * prob(flies | n) **
*prob(rule 3 | vp) * prob(like | v) * prob(a banana | np)*
$= .09 * .061 * .025 * .393 * .1 * 1.51*10^{-3}$
$= 8.14 * 10^{-9}$

II. *prob(fruit flies like a banana | s)*
*= prob(rule 1 | s) * prob(fruit | np) * prob(flies like a banana | vp)*
*= 1 * prob(rule 7 | np) * prob(fruit | n) **
*prob(rule 4 | vp) * prob(flies | v) * prob(like a banana | pp)*
*= prob(rule 7 | np) * prob(fruit | n) **
*prob(rule 4 | vp) * prob(flies | v) * prob(rule 9 | pp) * prob(like | prep) **
prob(a banana | np)
$= .14 * .061 * .22 * .07 * 1 * .065 * 1.51*10^{-3}$
$= 12.9 * 10^{-9}$

Here, we do not show the details of the calculation of *$1.51*10^{-3}$* for *prob(a banana | np)* that was developed above. By these calculations, parse tree II is slightly favored over parse tree I. This result suggests that this sentence would mean that fruit flies in the same way that a banana flies. Adding semantic considerations, we might find that the more plausible parse would have "fruit flies" as a noun phrase and "like" as a verb.

According to Allen and others (p 213), probabilistic parsing gives the correct result (parse tree) about half the time, which is better than pure guess-work but not great in general. For instance, the results we obtained by parsing "fruit flies like a banana," from the sample grammar and probabilities delivers the less intuitive result. Thus, we should consider more robust approaches to semantics if we want to be able to parse sentences like this one correctly. Statistical methods that don't take semantic features into account have limited value in complex NLP situations.

Exercises

1. Complete the calculation of *prob(n v prep det n | fruit flies like a banana)*, given the probabilities used in the discussion above. What is the resulting probability? Is this greater than that computed for *prob(n n v det n | fruit flies like a banana)*? Does this result agree with your intuition about which is the more reasonable interpretation for this ambiguous sentence?
2. Revise the grammar given in the discussion above by changing the probabilities for rules 2, 3, and 4 to .32, .33, and .35. What effect does this change have on the selection of the best parse tree for the sentence *Fruit flies like a banana*?
3. Change the following probabilities in the Markov chain to the following:
   prob(n | v) = .53
   prob(det | v) = .32
   prob(prep | v) = .15
   Now recalculate the part of speech taggings for the sentence *Fruit flies like a banana*. What now is the best part of speech tagging for the words in this sentence?
7. Consider the sentence *The woman saw the man with the telescope*. Compute the statistically best parse for this sentence, given the two alternative attachments of the prepositional phrase *with the telescope.*
8. Consider the sentence *The woman saw the man on the hill with the telescope.* Using the above grammar, how many different parse trees does this sentence have? Compute the statistically best parse for this sentence.
9. (Optional extra credit) Consider the problems of part of speech tagging and probabilistic parsing using a Prolog program that implements the grammar and lexicon discussed here. What would be needed in the Prolog program to automatically calculate the part of speech taggings and select the best one? What would be needed in the Prolog program to automatically calculate the parse tree probabilities and select the best one? Design such a program.

References

1. Charniak, E., *Statistical Language Learning*, MIT Press (1993).
2. Allen, J., *Natural Language Understanding*, Chapter 7.
3. DeRose, S.J., "Grammatical category disambiguation by statistical optimization," *Computational Linguistics* 14, 31-39 (1988).
4. Jurafsky, D. and J. Martin, *Speech and Language Processing*, Prentice-Hall (2000).

Semantics - Meaning Representation in NLP

The entire purpose of a natural language is to facilitate the exchange of ideas among people about the world in which they live. These ideas converge to form the "meaning" of an utterance or text in the form of a series of sentences. The meaning of a text is called its *semantics*. A fully adequate natural language semantics would require a complete theory of how people think and communicate ideas. Since such a theory is not immediately available, natural langauge researchers have designed more modest and pragmatic approaches to natural language semantics.

One such approach uses the so-called "logical form," which is a representation of meaning based on the familiar predicate and lambda calculi. In this section, we present this approach to meaning and explore the degree to which it can represent ideas expressed in natural language sentences. We use Prolog as a practical medium for demonstrating the viability of this approach. We use the lexicon and syntactic structures parsed in the previous sections as a basis for testing the strengths and limitations of logical forms for meaning representation.

Logic and Logical Forms

The Meaning of Simple Objects and Events

Quantifiers and the Meaning of Determiners

The Meaning of Modifiers

Adjectives

Relative Clauses

Plurals, Cardinality, and Mass Nouns

Negation

Question-Answering

Yes/No

How Many and Which

Who and What

Discourse Referents, Anaphora, Definite Reference (the)

Word Sense Disambiguation

Ontological Methods

Statistical Methods

The semantics, or meaning, of an expression in natural language can be abstractly represented as a *logical form*. Once an expression has been fully parsed and its syntactic ambiguities resolved, its meaning should be uniquely represented in logical form. Conversely, a logical form may have several equivalent syntactic representations. Semantic analysis of natural language expressions and generation of their logical forms is the subject of this chapter.

Consider the sentence "The ball is red." Its logical form can be represented by `red(ball101)`. This same logical form simultaneously represents a variety of syntactic expressions of the same idea, like "Red is the ball." and "Le bal est rouge."

The most challenging issue that stands between a syntactic utterance and its logical form is ambiguity; lexical ambiguity, syntactic ambiguity, and semantic ambiguity. Lexical ambiguity arises with words that have many senses. For example, the word "go" has at least the following different senses, or meanings:

move
depart
pass
vanish
reach
extend
set out

Compounding the situation, a word may have different senses in different parts of speech. The word "flies" has at least two senses as a noun (insects, fly balls) and at least two more as a verb (goes fast, goes through the air).

To assist with the categorization and representation of word meanings, an *ontology* is often used. An ontology is a broad classification scheme for objects, originally proposed by Aristotle. Many of the original classifications are still in use today; here is a typical list:

action	position
event	state
substance	affection
quantity	ideas
quality	concepts
relation	plans
place	situation
time	

Case grammars have been proposed to further clarify relations between actions and objects. So-called *case roles* can be defined to link certain kinds of verbs and various objects. These roles include:

agent
theme
instrument

For example, in "John broke the window with the hammer," a case grammar would identify John as the agent, the window as the theme, and the hammer as the instrument. More examples of case roles and their use are given in Allen, p 248-9.

Logical Forms and Lambda Calculus

The language of logical forms is a combination of the first order predicate calculus (FOPC) and the lambda calculus. The predicate calculus includes unary, binary, and n-ary predicates, such as:

Lisp notation	Prolog notation	Sentence represented
`(dog1 fido1)`	`dog1(fido1)`	"Fido is a dog"
`(loves1 sue1 jack1)`	`loves1(sue1, jack1)`	"Sue loves Jack"
`(broke1 john1 window1 hammer1)`	`broke1(john1, window1, hammer1)`	"John broke the window with a hammer"

The third example shows how the semantic information transmitted in a case grammar can be represented as a predicate.

Affixing a numeral to the items in these predicates designates that in the semantic representation of an idea, we are talking about a particular instance, or interpretation, of an action or object. For instance, `loves1` denotes a particular interpretation of "love."

Logical forms can be constructed from predicates and other logical forms using the operators & (and), `=>` (implies), and the quantifiers `all` and `exists`. In English, there are other useful quantifiers beyond these two, such as `many`, `a few`, `most`, and `some`. For example, the sentence "Most dogs bark" has the logical form

```
(most d1: (& (dog1 d1) (barks1 d1)))
```

Finally, the lambda calculus is useful in the semantic representation of natural language ideas. If `p(x)` is a logical form, then the expression `\x.p(x)` defines a function with bound variable `x`. *Beta-reduction* is the formal notion of applying a function to an argument. For instance, `(\x.p(x))a` applies the function `\x.p(x)` to the argument `a`, leaving `p(a)`.

Semantic Rules for Context Free Grammars

One way to generate semantic representations for sentences is to associate with each grammar rule an associated step that defines the logical form that relates to each syntactic category. Consider simple grammars with S, NP, VP, and TV categories.

1. If the grammar rule is S --> NP VP and the logical forms for NP and VP are `NP'` and `VP'` respectively, then the logical form `S'` for S is `VP'(NP')`. For example, in the sentence "bertrand wrote principia" suppose that

```
NP' = bertrand and VP' = \x.wrote(x, principia)
```

Then the logical form `S'` is the result of Beta reduction:

```
(\x.wrote(x, principia))bertrand = wrote(bertrand, principia)
```

2. If the grammar rule is VP --> TV NP and the logical forms for TV and NP are `TV'` and `NP'` respectively, then the logical form `VP'` for VP is `TV'(NP')`. For example, in the phrase "wrote principia," if

```
TV' = \y.\x.wrote(x, y) and NP' = principia
```

Then the logical form `VP' = \x.wrote(x, principia)` after Beta reduction.

Prolog Representation

To accommodate the limitations of the ASCII chareacter set, the following conventions are used in Prolog to represent logical forms and lambda expressions.

Expression	Prolog convention	
(forall x: p(x))	`all(X, p(X))`	(recall that Prolog variables are capitalized)
(exists x: p(x))	`exists(X, p(X))`	
and	`&`	
implies	`=>`	
\x.p(x)	`X^p(X)`	
(\x.p(x)) y	`reduce(X^p(X), Y, LF)`	

The Beta reduction `reduce` is defined by the Prolog rule `reduce(Arg^Exp, Arg, Exp)`. For example,

```
reduce(X^halts(X), shrdlu, LF)
```

gives the answer `LF = halts(shrdlu)`.

Semantics of a Simple Grammar

Using these ideas, we can write Prolog rules with semantics as follows:

```
s(S) --> np(NP), vp(VP),   {reduce(VP, NP, S)}.
np(NP) --> det(D), n(N),   {reduce(D, N, NP)}.
np(NP) --> n(NP).
vp(VP) --> tv(TV), np(NP), {reduce(TV, NP, VP)}.
vp(VP) --> iv(VP).
```

In the first rule, VP has the lambda expression and NP has the subject. In fact, the references to `reduce` can be removed from these rules, and their effects can be inserted directly where they will take place. That is, the following set of rules

```
s(S) --> np(NP), vp(NP^S).
np(NP) --> det(N^NP), n(N).
np(NP) --> n(NP).
vp(VP) --> tv(NP^VP), np(NP).
vp(VP) --> iv(VP).
```

captures the same semantics as the original set. This is called *partial execution*.

We encode lexical entries for nouns and verbs as follows:

```
n(shrdlu) --> [shrdlu].             % proper nouns represent
themselves
n(terry) --> [terry].
n(X^ `program(X)) --> [program].     % this encoding produces the "is-
a" property
tv(Y^X^ `wrote(X, Y)) --> [wrote].
iv(X^ `halts(X)) --> [halts].
```

Note that nouns which represent classes of objects are encoded with a logical form that permits specific proper nouns to become members of that class. For instance, the logical form `program(shrdlu)` may result from the parsing of `"shrdlu is a program."` Transitive verbs, like `wrote`, generate 2-variable lambda expressions, while intransitive verbs generate 1-variable lambda expressions. With this grammar, we can parse sentences and directly generate encodings of logical forms. For example, the call

```
s(LF, [terry, wrote, shrdlu], [])
```

yields the result `LF = wrote(terry, shrdlu)`, in a sequence of steps shown by the following partial trace (failing steps are omitted from this trace):

```
?- s(LF, [terry, wrote, shrdlu], []).
   Call:  (  7) s(_G276, [terry, wrote, shrdlu], []) ? creep
   Call:  (  8)   np(_L131, [terry, wrote, shrdlu], _L132) ? creep
   Call:  (  9)      n(_L131, [terry, wrote, shrdlu], _L132) ? creep
   Exit:  (  9)      n(terry, [terry, wrote, shrdlu], [wrote, shrdlu]) ? creep
   Exit:  (  8)    np(terry, [terry, wrote, shrdlu], [wrote, shrdlu]) ? creep
   Call:  (  8)    vp(terry^_G276, [wrote, shrdlu], []) ? creep
   Call:  (  9)      tv(_G382^terry^_G276, [wrote, shrdlu], _L171) ? creep
   Exit:  (  9)      tv(_G382^terry^wrote(terry, _G382), [wrote, shrdlu], [shrdlu]) ? creep
   Call:  (  9)      np(_G382, [shrdlu], []) ? creep
   Call:  ( 10)          n(_G382, [shrdlu], []) ? creep
   Exit:  ( 10)          n(shrdlu, [shrdlu], []) ? creep
   Exit:  (  9)      np(shrdlu, [shrdlu], []) ? creep
   Exit:  (  8)    vp(terry^wrote(terry, shrdlu), [wrote, shrdlu], []) ? creep
   Exit:  (  7) s(wrote(terry, shrdlu), [terry, wrote, shrdlu], []) ? creep

LF = wrote(terry, shrdlu)
```

Quantified Noun Phrases

The logical forms that represent quantified noun phrases, like "every program" in the sentence "every program halts," should carry appropriate quantifiers. To accomplish this, determiners like "every" should be encoded in the lexicon in a way that anticipates the addition of a quantifier to the logical form being constructed. Here is a definition for the determiner "every."

```
det((X^P)^(X^Q)^all(X, P=>Q)) --> [every].
```

The Prolog logical form here is an encoding of the lambda expression \p.\q.(all(x, p(x) => q(x))). Note here that the variable x is bound in each of the expressions p and q.

This definition suggests that when "every" is used as a determiner in a sentence, the semantic representation of that sentence is a lambda expression with two expressions (P and Q) identifying properties of an object X, such that P(X) implies Q(X). For instance, in "every program halts," the expression P=>Q is instantiated with `program(X)` and `halts(X)` for the object `X`, so we have the following logical form as a meaning representation:

```
\p.\q.(all(x, p(x) => q(x)) (\y.halts(y)) (\z.program(z))
      =  \q.(all(x, program(x) => q(x)) (\y.halts(y)) )
      =  all(x, program(x) => \y.halts(y) (x) )
      =  all(x, program(x) => halts(x) )
```

Here is a partial trace for this parse.

```
?- s(LF, [every, program, halts], []).
   Call:  (  7) s(_G276, [every, program, halts], [])
   Call:  (  8)    np(_L131, [every, program, halts], _L132)
   Call:  (  9)       det(_G379^_G380, [every, program, halts], _L146)
   Call:  ( 10)          'C'([every, program, halts], _L159, _L160)
   Exit:  ( 10)          'C'([every, program, halts], every, [program, halts])
   Call:  ( 10)           det(every, _G379^_G380)
   Exit:  ( 10)           det(every,  (_G382^_G383)^ (_G382^_G389)^all(_G382, _G383=>_G389))
   Call:  ( 10)           _L146=[program, halts]
   Exit:  ( 10)           [program, halts]=[program, halts]
   Exit:  (  9)        det( (_G382^_G383)^ (_G382^_G389)^all(_G382, _G383=>_G389),
                                                        [every, program, halts], [program, halts])
   Call:  (  9)        n(_G382^_G383, [program, halts], _L132)
   Exit:  (  9)         n(_G382^ `program(_G382), [program, halts], [halts])
   Exit:  (  8)    np( (_G382^_G389)^all(_G382, `program(_G382)=>_G389),
                                                        [every, program, halts], [halts])
   Call:  (  8)    vp( ( (_G382^_G389)^all(_G382, `program(_G382)=>_G389))^_G276, [halts], [])
   Call:  (  9)        iv( ( (_G382^_G389)^all(_G382, `program(_G382)=>_G389))^_G276, [halts], [])
   Exit:  (  9)        iv( ( (_G382^_G389)^all(_G382, `program(_G382)=>_G389))^
                                    `halts( (_G382^_G389)^all(_G382, `program(_G382)=>_G389)),
                                           [halts], [])
   Exit:  ( 8)    vp( ( (_G382^_G389)^all(_G382, `program(_G382)=>_G389))^
                            `halts( (_G382^_G389)^all(_G382, `program(_G382)=>_G389)),
                                           [halts], [])
   Exit:  ( 7) s(`halts( (_G382^_G389)^all(_G382, `program(_G382)=>_G389)),
                                           [every, program, halts], [])
```

LF = `halts( (_G382^_G389)^all(_G382, `program(_G382)=>_G389))

Note that the logical form derived in this parse is not the same as the one developed above. To see that they are equivalent, note that LF is of the form `halts( \z.\x.all(x, program(x) => z(x) ), and `halts has the logical form \y.halts(y).
Beta-reducing these two, we get:

```
(\z.\x.all(x, program(x) => z(x)) (\y.halts(y))
 =  \x.all(x, program(x) => \y.halts(y) (x))
 =  \x.all(x, program(x) => halts(x))
```

Semantics of Filler-Gap Dependencies

Gaps are analyzed like proper nouns except they supply a variable as the second argument.

```
np((X^S)^S, gap(np, X)) --> [].
```

That is, a relative clause meaning is a property, which is conjoined with a noun meaning (another property).

```
optrel((X^S1) ^ (X ^ (S1&S2))) --> relpron, vp(X^S2, nogap).
optrel((X^S1) ^ (X ^ (S1&S2))) --> relpron, s(S2, gap(np, X)).
```

For example, the clauses "professor who wrote principia" and "that bertrand wrote" represented by the following logical forms:

```
M^(professor(M) & wrote(M, principia))
B^wrote(bertrand, B)
```

A question can be interpreted as a property that is true of the answer to the question. For yes-no questions, we want the property "yes" conveyed if the condition in the inverted sentence is satisfied:

```
sinv(S, GapInfo) --> aux, np(VP^S, nogap), vp(VP, GapInfo).
q(yes^S) --> sinv(S, nogap).
```

For complement questions, we want the property to be that given by the VP.

```
q(V) --> whpron, vp(VP, nogap).
q(X^S) --> whpron, sinv(S, gap(np, X)).
```

Exercises

1. Add to the lexicon an appropriate encoding for the determiner "a", so that it can be used in sentences like "terry wrote a program". Hand-trace the application of the Prolog rules given in this section with this sentence and show the intermediate logical forms that lead to its logical form representation, exists(x, program(x) => wrote(terry, X)).
2. Assuming the grammatical rules found in this section, find appropriate semantic representations for the following statements:
   a. terry wrote a program that halts
   b. a program halts
   c. terry wrote every program that halts
3. Give an example of a yes-no question and a complement question to which the rules in the last section can apply. For each example, show the intermediate steps in deriving the logical form for the question. Assume there are sufficient definitions in the lexicon for common words, like "who", "did", and so forth.
4. Look at program 4.2 on p 102 of Pereira. Using a trace, show the intermediate steps in the parse of the sentence "every student wrote a program."

References

Allen, chapter 9
Pereira, chapter 4

Application: A Question-Answering System

This program is an adaptation of a question-answering system developed in Pereira and Shieber (pp 149-157). This is a simple program that demonstrates the main semantic ideas that can be represented by logical forms, quantifiers, and the lambda calculus. The complete program is given in the attached file.

This discussion illustrates how the program is used, what it can do, and some key points about how it represents, stores, and retrieves information that it has "learned" as it interacts with the user.

Sample Dialog

Here is a dialog that occurs with the program `pereiraTalk` (in the directory `~allen/public_html/prolog`) after typing consult(pereiraTalk). at the ?- prompt:

```
?- run.
>> principia is a book
Asserted "book(principia)."
>> bertrand wrote every book
Asserted "writes(bertrand, _G332):-book(_G332)."
>> what did bertrand write
principia.
>> what did allen write
no.
>> who wrote every book
Error: "too difficult."
>> who wrote principia
bertrand.
>> bruce read every book that bertrand wrote
Asserted "reads(bruce, _G379):-book(_G379), writes(bertrand, _G379)."
>> shrdlu is a book
Asserted "book(shrdlu)."
>> what did bruce read
principia, shrdlu.
>>
```

As is evident, the program stores information about what has been asserted and uses that information to answer questions, provided that it can parse the assertions and questions that are entered.

Successful parsing depends on syntactic correctness. Parsable assertions result in the message "Asserted: C" where C is a Prolog *rule* or *fact*. Not shown in this excange is the program's adding this fact to the database with which it has started (facts and rules defining the basic grammar and lexicon), and the program's translation of the original assertion into a logical form, which is its semantic representation.

Parsable questions are answered by a message "yes" (if the question is a yes/no question and the answer is true according to the facts and rules stored in the database during the processing of prior assertions), "no" (if the answer is not so justified), or a list of all values that answer the question (if the question is not a yes/no question).

The Overall Strategy

The overall strategy for the program is simple, and given by the following code: give a prompt, read a sentence (assertion or query), parse it and discover its meaning as a logical form, convert that logical form into a Prolog clause, generate and display a reply, and then repeat these steps. If the sentence cannot be successfully parsed (grammatical or lexicon error) or if a Prolog clause cannot be generated from the logical form, then the message `Error: "too difficult"` appears for that sentence.

```
run :-
        write('>> '),           % prompt the user
        read_sent(Words),       % read a sentence
        talk(Words, Reply),     % process it with TALK
        print_reply(Reply),     % generate a reply
        run.                    % pocess more sentences

%%% talk(Sentence, Reply)
%%%
%%%     Sentence ==> sentence to form a reply to
%%%     Reply    <== appropriate reply to the sentence

talk(Sentence, Reply) :-                     % parse the sentence
         parse(Sentence, LF, Type),
        % convert the FOL logical form into a Horn
        % clause, if possible
         clausify(LF, Clause, FreeVars), !,
        % concoct a reply, based on the clause and
        % whether sentence was a query or assertion
         reply(Type, FreeVars, Clause, Reply).

talk(Sentence, error('too difficult')).
```

The Lexicon

Six different forms for each verb are stored in the lexicon: the nonfinite form, the third person singular form, the past tense, the past participle, the present participle, and the logical form.

The logical form for an intransitive verb is an encoding of the lambda expression \x.verb(x), where x is a placeholder for the subject. For instance, "halts" has a logical form that encodes the expression \x.halts(x), or `X^ 'halts(X)`.

```
iv(nonfinite,       LF) --> [IV], {iv(IV, _, _, _, _, LF)}.
iv(finite,          LF) --> [IV], {iv(_, IV, _, _, _, LF)}.
iv(finite,          LF) --> [IV], {iv(_, _, IV, _, _, LF)}.
iv(past_participle, LF) --> [IV], {iv(_, _, _, IV, _, LF)}.
iv(pres_participle, LF) --> [IV], {iv(_, _, _, _, IV, LF)}.
iv(halt, halts, halted, halted, halting,   X^ `halts(X)).
iv(run,  runs,  ran,    run,    running,   X^ `runs(X)).
```

Transitive verbs have as a logical form an encoding of the lambda expression \x.\y.verb(x, y), where x is the subject of the verb and y is the object. For instance, "speaks" is an encoding of \x.\y.speaks(x, y), or `X^Y^ 'speaks(X, Y)`.

```
tv(nonfinite,   LF)         --> [TV],   {tv(TV, _, _, _, _, LF)}.
tv(finite,      LF)         --> [TV],   {tv(_, TV, _, _, _, LF)}.
tv(finite,      LF)         --> [TV],   {tv(_, _, TV, _, _, LF)}.
tv(past_participle,     LF) --> [TV],   {tv(_, _, _, TV, _, LF)}.
tv(pres_participle,     LF) --> [TV],   {tv(_, _, _, _, TV, LF)}.
tv(write,   writes,   wrote,     written,   writing,    X^Y^ `writes(X,Y)).
tv(read,    reads,    read,      read,      reading,    X^Y^ `reads(X,Y)).
tv(speak,   speaks,   spoke,     spoken,    speaking,   X^Y^ `speaks(X,Y)).
tv(meet,    meets,    met,       met,       meeting,    X^Y^ `meets(X,Y)).
tv(concern, concerns, concerned, concerned, concerning, X^Y^ `concerns(X,Y)).
tv(run,     runs,     ran,       run,       running,    X^Y^ `runs(X,Y)).
```

Auxiliary verbs (to, does, did, ...), relative pronouns (that, who, whom), wh-words (who, whom, what) are enoded in their usual way.

Determiners are encoded with logical forms that reflect their meaning. For example, if we assert "Bertrand wrote every book", the system needs a logical form that will say, in effect, that "for every X , if X is a book then bertrand wrote X." The logical form for "every" in the lexicon has that structure:

det(every, (X^S1) ^ (X^S2) ^ all(X, S1=>S2)).

Similarly, the logical form for determiners "a" and "some" are coded as follows:

det(a,   (X^S1) ^ (X^S2) ^ exists(X, S1&S2)).
det(some,   (X^S1) ^ (X^S2) ^ exists(X, S1&S2)).

There are two different types of nouns, those which represent classifications of objects or people, and proper nouns (which represent specific objects and people themselves -- e.g., principia). The first type of noun has a logical form which allows proper nouns to become members of that classification. For instance, the classification "author" has the logical form \x.author(x), which is encoded `X^ `author(X)`. The second type has itself as its logical form.

```
n(author,     X^ `author(X)).
pn(principia, principia).
```

The Grammar

The grammar for this program parses simple declarative sentences, sentences with relative clauses, inverted sentences, and various kinds of questions, using familiar strategies from our earlier studies of syntax. What's new here is that each grammar rule generates a logical form as a byproduct (rather than a portion of a parse tree).

```
s(S, GapInfo)   --> np(VP^S, nogap), vp(finite, VP, GapInfo).

sinv(S, GapInfo) --> aux(finite/Form, VP1^VP2),
                     np(VP2^S, nogap), vp(Form, VP1, GapInfo).

q(S => `answer(X))   --> whpron, vp(finite, X^S, nogap).
q(S => `answer(X))   --> whpron, sinv(S, gap(np, X)).
q(S => `answer(yes)) --> sinv(S, nogap).
q(S => `answer(yes)) --> [is],  np((X^S0)^S, nogap),
                                np((X^true)^exists(X,S0&true), nogap).

np(NP, nogap) --> det(N2^NP), n(N1), optrel(N1^N2).
np(NP, nogap) --> pn(NP).
np((X^S)^S, gap(np, X)) --> [].
```

```
vp(Form, X^S, GapInfo)  --> tv(Form, X^VP), np(VP^S, GapInfo).
vp(Form, VP, nogap)     --> iv(Form, VP).
vp(Form1, VP2, GapInfo) --> aux(Formrlogina1/Form2, VP1^VP2),
                            vp(Form2, VP1, GapInfo).
vp(Form1, VP2, GapInfo) --> rov(Form1/Form2, NP^VP1^VP2),
                            np(NP, GapInfo), vp(Form2, VP1, nogap).
vp(Form2, VP2, GapInfo) --> rov(Form1/Form2, NP^VP1^VP2),
                            np(NP, nogap), vp(Form1, VP1, GapInfo).
vp(finite, X^S, GapInfo) --> [is], np((X^P)^exists(X,S&P), GapInfo).

optrel((X^S1)^(X^(S1&S2))) --> relpron, vp(finite, X^S2, nogap).
optrel((X^S1)^(X^(S1&S2))) --> relpron, s(S2, gap(np, X)).
optrel(N^N) --> [].
```

Parsing Example: Assertions

Parsing an assertion proceeds in two steps: developing a logical form for the meaning of the assertion and then adding that logical form (converted to Prolog) to the database of facts and rules. The first step is initiated by:

parse(Sentence, LF, assertion) :- s(LF, nogap, Sentence, []).

The reply to an assertion is made simply be reporting to the user the Prolog fact or rule that was generated from the logical form after the paring of the sentence. The reply function also adds this fact or rule to the Prolog program, so that it can participate in answering future queries. This is accomplished by the Prolog function "assert", which takes any Prolog fact or rule and adds it dynamically to the database.

```
reply(assertion, _FreeVars, Assertion, asserted(Assertion)) :- assert(Assertion), !.
```

Parsing Example: Queries

Parsing a query requires that an answer be developed alongside the logical form. If the sentence is a query, then the productions for function q all return a logical form LF = "`Q => answer(A)`". This logical form is then used to develop an answer, which will be true exactly when the database of facts and rules can justify the truth of A. The first call in this process is:

parse(Sentence, LF, query) :- q(LF, Sentence, []).

For example, the parse for the sentence "who wrote principia" wlll return the logical form

LF = `writes(X, principia)=> `answer(X)

The reply to a query is made by first finding a set of all the answers that satisfy the query, replying with that set (if it exists), or "no" if it doesn't.

```
reply(query, FreeVars, (answer(Answer):-Condition), Reply) :-
        (setof(Answer, FreeVars^Condition, Answers) -> Reply = answer(Answers) ;
                                (Answer = yes -> Reply = answer([no]) ;
                                            Reply = answer([none]))), !.
```

This can be read as an ordinary "if-then-else" statement, where the arrows -> denote "then" and the semicolons (;) denote "or else". So the first line generates the Reply if such a set is found, and the second generates the Reply if the answer is "yes," and otherwise the third line generates the Reply.

The `setof` function builds an ordered set of instances of Answer, called Answers (in the form of a list without duplicates) that satisfies the goal FreeVars^Condition. For instance, if the goal is answer(X) := writes(X, principia), then the call is setof(X, FreeVars^writes(X, principia), Answers). If the fact writes(bertrand, principia) is in the database at the time, then setof will return Answers = [bertrand]. Otherwise, setof will fail.

Clausify: Converting Logical Forms to Prolog Clauses

The function clausify converts a logical form to an equivalent Prolog fact or rule, whenever it can. That is, given a logical form FOL, clausify generates a Clause which is the equivalent Prolog fact or rule. A third argument to the function, `FreeVars`, is a list of the variables that are free in the antecedent of the clause.

```
clausify(FOL, Clause, FreeVars)
```

Universally quantified logical forms are clausified by stripping the quantifier from the clausified version of the logical form. For instance, `clausify(all(B, book(B) & wrote(bertrand, B)), Clause, FreeVars)` leaves Clause = `book(B), wrote(bertrand, B)` and FreeVars = `[B]`. This makes sense, since both forms are logically equivalent to "B is a book if and only if bertrand wrote B."

```
clausify(all(X,F0),F,[X|V]) :- clausify(F0,F,V).
```

For implications, the consequent C0 is clausifed as the literal C and the antecedent A0 is clausified as A. Then the Prolog clause C :- A, itself an implication, is formed out of these results. The list V of free variables is passed along in this process.

```
clausify(A0=>C0,(C:-A),V) :- clausify_literal(C0,C), clausify_antecedent(A0,A,V).
```

Literals are left unchanged by clausify, though an empty free variable list is generated.

clausify(C0,C,[]) :- clausify_literal(C0,C).

The function clausify_antecedent(FOL, Clause, FreeVars) leaves literals unchanged (except the literal marker is removed). For conjunctions, clausify-antecedent clausifies each conjunct separately and then combines them with a comma to form the right-hand side of a Prolog rule. For existentials, the quantifier is stripped and the free variable is added to the free variable list.

clausify_antecedent(L0,L,[]) :- clausify_literal(L0,L).

```
clausify_antecedent(E0&F0, (E,F), V) :- clausify_antecedent(E0,E,V0),
                                        clausify_antecedent(F0,F,V1),
                                        conc(V0,V1,V).
clausify_antecedent(exists(X,F0), F, [X|V]) :- clausify_antecedent(F0,F,V).

clausify_literal(`L, L).
```

```
%%%  TALK Program, adapted from Pereira & Shieber (1987).

:- op(500,xfy,'&').
:- op(510,xfy,'=>').
:- op(100,fx,'`').

run :-
        write('>> '),           % prompt the user
        read_sent(Words),       % read a sentence
        talk(Words, Reply),     % process it with TALK
        print_reply(Reply),     % generate a reply
        run.                    % pocess more sentences

%%% talk(Sentence, Reply)
%%%
%%%     Sentence ==> sentence to form a reply to
%%%     Reply    <== appropriate reply to the sentence

talk(Sentence, Reply) :-                    % parse the sentence
         parse(Sentence, LF, Type),
       % convert the FOL logical form into a Horn
       % clause, if possible
         clausify(LF, Clause, FreeVars), !,
       % concoct a reply, based on the clause and
       % whether sentence was a query or assertion
         reply(Type, FreeVars, Clause, Reply).

% No parse was found; sentence is too difficult for the grammar
% or the lexicon.
talk(_Sentence, error('too difficult')).


%%% parse(sentence, LF, Type)
%%%
%%% Sentence    ==> sentence to parse
%%%     LF      <== logical form (in FOL) of sentence
%%%     Type    <== type of Sentence
%%%             (query or assertion)

% Parsing an assertion: a finite sentence without gaps.
parse(Sentence, LF, assertion) :- s(LF, nogap, Sentence, []).

% Parsing a query: a question.
parse(Sentence, LF, query) :- q(LF, Sentence, []).


/* Nonterminal names:
   q       Question
   sinv    INVerted Sentence
   s       noninverted Sentence
   np      Noun Phrase
   vp      Verb Phrase
   iv      Intransitive Verb
   tv      Transitive Verb
   aux     AUXiliary verb
   rov     subject_Object Raising Verb
   optrel  OPTional RELative clause
   relpron RELative PRONoun
   whpron  WH PRONoun
   det     DETerminer
   n       Noun
   pn      Proper Noun

  Typical order of and values for arguments:

  1.    verb form:
     (main verbs) finite, nonfinite, etc.
     (auxiliaries and raising verbs) Form1-Form2
         where Forml is form of embedded VP
               Form2 is form of verb itself
  2.    FOL   logical form
  3.    gap   information:
              nogap or gap(Nonterm, Var)
                 where Nonterm is nonterminal for gap
                 Var is the LF variable that
                     the filler will bind
*/

%%% Declarative Sentences

s(S, GapInfo)   --> np(VP^S, nogap), vp(finite, VP, GapInfo).

%%% Inverted Sentences

sinv(S, GapInfo) --> aux(finite/Form, VP1^VP2),
                     np(VP2^S, nogap), vp(Form, VP1, GapInfo).

%%%     Questions

q(S => `answer(X))   --> whpron, vp(finite, X^S, nogap).
q(S => `answer(X))   --> whpron, sinv(S, gap(np, X)).
q(S => `answer(yes)) --> sinv(S, nogap).
q(S => `answer(yes)) --> [is],  np((X^S0)^S, nogap),
                  np((X^true)^exists(X,S0&true), nogap).

%%% Noun Phrases

np(NP, nogap) --> det(N2^NP), n(N1), optrel(N1^N2).
np(NP, nogap) --> pn(NP).
np((X^S)^S, gap(np, X)) --> [].

%%% Verb Phrases

vp(Form, X^S, GapInfo)  --> tv(Form, X^VP), np(VP^S, GapInfo).
vp(Form, VP, nogap)      --> iv(Form, VP).
vp(Form1, VP2, GapInfo) --> aux(Form1/Form2, VP1^VP2),
                            vp(Form2, VP1, GapInfo).
vp(Form1, VP2, GapInfo) --> rov(Form1/Form2, NP^VP1^VP2),
                            np(NP, GapInfo), vp(Form2, VP1, nogap).
vp(Form2, VP2, GapInfo) --> rov(Form1/Form2, NP^VP1^VP2),
                            np(NP, nogap), vp(Form1, VP1, GapInfo).
vp(finite, X^S, GapInfo) --> [is], np((X^P)^exists(X,S&P), GapInfo).

%%% Relative Clauses

optrel((X^S1)^(X^(S1&S2))) --> relpron, vp(finite, X^S2, nogap).
optrel((X^S1)^(X^(S1&S2))) --> relpron, s(S2, gap(np, X)).
optrel(N^N) --> [].

% Dictionary

% Verb  entry arguments:
%       1.      nonfinite form of the verb
%       2.      third person singular present tense form of the verb
%       3.      past tense form of the verb
%       4.      past participle form of the verb
%       5.      pres participle form of the verb
%       6.      logical form of the verb

iv(nonfinite,       LF) --> [IV], {iv(IV, _, _, _, _, LF)}.
iv(finite,          LF) --> [IV], {iv(_, IV, _, _, _, LF)}.
iv(finite,          LF) --> [IV], {iv(_, _, IV, _, _, LF)}.
iv(past_participle, LF) --> [IV], {iv(_, _, _, IV, _, LF)}.
iv(pres_participle, LF) --> [IV], {iv(_, _, _, _, IV, LF)}.
iv(halt, halts, halted, halted, halting,   X^ `halts(X)).
iv(run,  runs,  ran,    run,    running,   X^ `runs(X)).

tv(nonfinite,   LF)         --> [TV],   {tv(TV, _, _, _, _, LF)}.
tv(finite,      LF)         --> [TV],   {tv(_, TV, _, _, _, LF)}.
tv(finite,      LF)         --> [TV],   {tv(_, _, TV, _, _, LF)}.
tv(past_participle,     LF) --> [TV],   {tv(_, _, _, TV, _, LF)}.
tv(pres_participle,     LF) --> [TV],   {tv(_, _, _, _, TV, LF)}.

tv(write,   writes,   wrote,     written,   writing,    X^Y^ `writes(X,Y)).
tv(read,    reads,    read,      read,      reading,    X^Y^ `reads(X,Y)).
tv(speak,   speaks,   spoke,     spoken,    speaking,   X^Y^ `speaks(X,Y)).
tv(meet,    meets,    met,       met,       meeting,    X^Y^ `meets(X,Y)).
tv(concern, concerns, concerned, concerned, concerning, X^Y^ `concerns(X,Y)).
tv(run,     runs,     ran,       run,       running,    X^Y^ `runs(X,Y)).

rov(nonfinite   /Requires,    LF) --> [ROV], {rov(ROV, _, _, _, _, LF,
Requires)}.
rov(finite      /Requires,    LF) --> [ROV], {rov(_, ROV, _, _, _, LF,
Requires)}.
rov(finite      /Requires,    LF) --> [ROV], {rov(_, _, ROV, _, _, LF,
Requires)}.
```

```
rov(past_participle/Requires, LF) --> [ROV], {rov(_, _, _, ROV, _, LF, Requires)}.
rov(pres_participle/Requires, LF) --> [ROV], {rov(_, _, _, _, ROV, LF, Requires)}.
rov(want,   wants,    wanted,    wanted,    wanting,
     % semantics is partial execution of
     % NP ^ VP ^ Y ^ NP( X^want(Y,X,VP(X))
      ((X^ `want(Y, X, Comp))^S) ^ (X^Comp) ^ Y ^ S,
     % form of VP required:
      infinitival).

aux(Form, LF) --> [Aux], {aux(Aux, Form, LF)}.
aux(to,    infinitival/nonfinite,    VP^ VP).
aux(does,  finite/nonfinite,         VP^ VP).
aux(did,   finite/nonfinite,         VP^ VP).
aux(could, finite/nonfinite,         VP^ VP).
aux(have,  nonfinite/past_participle, VP^ VP).
aux(has,   finite/past_participle,   VP^ VP).
aux(been,  past_participle/present_participle, VP^ VP).
aux(be,    nonfinite/present_participle,       VP^ VP).

relpron       --> [RP], {relpron(RP)}.
relpron(that).
relpron(who).
relpron(whom).

whpron         --> [WH], {whpron(WH)}.
whpron(who).
whpron(whom).
whpron(what).

det(LF)   --> [D], {det(D, LF)}.
det(every,  (X^S1) ^ (X^S2) ^ all(X, S1=>S2)).
det(a,      (X^S1) ^ (X^S2) ^ exists(X, S1&S2)).
det(some,   (X^S1) ^ (X^S2) ^ exists(X, S1&S2)).

n(LF)     --> [N], {n(N, LF)}.
n(author,     X^ `author(X)).
n(book,       X^ `book(X)).
n(professor,  X^ `professor(X)).
n(program,    X^ `program(X)).
n(programmer, X^ `programmer(X)).
n(student,    X^ `student(X)).
n(person,     X^ `person(X)).
n(language,   X^ `language(X)).

pn((E^S)^S) --> [PN], {pn(PN, E)}.
pn(allen,    allen).
pn(bruce,    bruce).
pn(bertrand, bertrand).
pn(terry,    terry).
pn(bill,     bill).
pn(david,    david).
pn(kathy,    kathy).
pn(behshad,  behshad).
pn(shane,    shane).
pn(principia, principia).
pn(shrdlu,   shrdlu).
pn(prolog,   prolog).
pn(english,  english).
pn(chinese,  chinese).
pn(korean,   korean).
pn(swahili,  swahili).

%%%  Clausifier

%%%  clausify(FOL, Clause, FreeVars)
%%%
%%%  FOL      ==> FOL expression to be converted to clause form
%%%  Clause   <== clause form of FOL expression
%%%  FreeVars <== free variables in clause

% Universals: variable is left implicitly scoped.
clausify(all(X,F0),F,[X|V]) :- clausify(F0,F,V).

% Implications: consequent must be a literal,
%               antecedent is clausified specially.
clausify(A0=>C0,(C:-A),V) :- clausify_literal(C0,C),
                             clausify_antecedent(A0,A,V).

% Literals: left unchanged (except literal marker is removed).
clausify(C0,C,[]) :- clausify_literal(C0,C).

% Note that conjunctions and existentials are
% disallowed, since they can't form Horn clauses.

%%% clausify_antecedent(FOL, Clause, FreeVars)
%%%
%%%     FOL      ==> FOL expression to be converted to clause form
%%%     Clause   <== clause form of FOL expression
%%%     FreeVars ==> list of free variables in clause

% Literals: left  unchanged (except literal marker is removed).
clausify_antecedent(L0,L,[]) :- clausify_literal(L0,L).

% Conjunctions: each conjunct is clausified separately.
clausify_antecedent(E0&F0, (E,F), V) :-
        clausify_antecedent(E0,E,V0),
        clausify_antecedent(F0,F,V1),
        conc(V0,V1,V).

% Existentials: variable is left implicitly scoped.
clausify_antecedent(exists(X,F0), F, [X|V]) :-
        clausify_antecedent(F0,F,V).
```

```
%%%  clausify_literal(Literal, Clause)
%%%
%%%      Literal ==> FOL literal to be converted
%%%                  to clause form
%%%      Clause  <== clause form of FOL expression

% Literal is left unchanged (except literal marker is removed).
clausify_literal(`L, L).

% Auxiliary Predicates

conc([], List, List).
conc([Element|Rest], List, [Element|LongRest]) :-
                        conc(Rest, List, LongRest).

%%% read_sent(Words)
%%% input ==> series of words terminated by a new line character
%%% Words <== list of the words
read_sent(Words) :- get0(Char), read_sent(Char, Words).

read_sent(C, []) :- newline(C), !.
read_sent(C, Words) :- space(C), !, get0(Char),
                        read_sent(Char, Words).
read_sent(C, [Word|Words]) :- read_word(C, Chars, Next),
                        name(Word, Chars),
                        read_sent(Next, Words).

read_word(C, [], C) :- space(C), !.
read_word(C, [], C) :- newline(C), !.
read_word(C, [C|Chars], Last) :- get0(Next),
                        read_word(Next, Chars, Last).

newline(10).
space(32).

%%% reply(Type, FreeVars, Clause, Reply)
%%%
%%%     Type    ==>     the constant "query" or "assertion"
%%%                            depending on whether clause should
%%%                            be interpreted as a query or assertion
%%%     FreeVars        ==>     the free variables (to be
%%%                            interpreted existentially) in the clause
%%% Clause ==> the clause being replied to
%%% Reply  <== the reply
%%%
%%% If the clause is interpreted as an assertion,
%%% the predicate has a side effect of asserting
%%% the clause to the database.

%Replying to a query.
reply(query, FreeVars,
     (answer(Answer):-Condition), Reply) :-
```

```
   % find all the answers that satisfy the query,
   % replying with that set if it exists, or "no"
   % or "none" if it doesn't.
     (setof(Answer, FreeVars^Condition, Answers)
         -> Reply = answer(Answers)
         ;  (Answer = yes
               -> Reply = answer([no])
               ;  Reply = answer([none]))), !.

% Replying to an assertion.
% assert the assertion and tell user what we asserted
reply(assertion, _FreeVars, Assertion, asserted(Assertion)) :-
        assert(Assertion), !.

% Replying to some other type of sentence.
reply(_Type, _FreeVars, _Clause, error('unknown type')).


%%% print_reply(Reply)
%%%
%%% Reply ==> reply generated by reply predicate
%%%           that is to be printed to the standard output.

print_reply(error(ErrorType)) :-
     write('Error: "'), write(ErrorType), write('."'), nl.

print_reply(asserted(Assertion)) :-
     write('Asserted "'), write(Assertion), write('."'), nl.

print_reply(answer(Answers)) :- print_answers(Answers).

%%%     print_answer(Answers)
%%%
%%%     Answers ==> nonempty list of answers to be printed
%%%                 to the standard output separated by commas.

print_answers([Answer]) :- !, write(Answer), write('.'), nl.

print_answers([Answer|Rest]) :- write(Answer), write(', '),
                                print_reply(answer(Rest)).
```

The ALE Homepage

This page contains descriptions of the Attribute-Logic Engine (ALE), Version 3.2, a freeware logic programming and grammar parsing and generation system. This includes information on obtaining the system, user's guide, graphical interfaces, and grammars.

What's New

- January, 2003: Bug fixed in SWI port of ALE 3.2.1 - compilation of grammar source would fail in some circumstances.

- December, 2001: ALE 3.2.1 is available. This version patches ALE and its source-level debugger so that they work with SICStus 3.8.6 and SWI 4.0.

- June, 1999: ALE 3.2 is available. In this version:
  - **ALE is faster (again)**, both at compile-time and run-time,
  - **A new parsing compilation algorithm (Empty-First-Daughter closure)**, which:
    - corrects a long-standing problem in ALE with combining empty categories - EFD closure closes the phrase structure rules of a grammar under prefixes of empty category daughters, so any permutation of empty categories can, in principle, be combined to form a new empty category;
    - works around a problem that non-ISO-compatible Prologs, including SICStus Prolog, have with asserted predicates that results in leftmost empty category daughters not being able to combine with their own outputs;
    - allows parsers to establish a precondition that rules only need to be closed with non-empty leftmost daughters at run-time - this allows ALE, at each step in its right-to-left pass throught the string, to copy all of the edges in the internal database back onto the heap before they can be used again, and thus reduces edge copying to a constant 2 copies per edge for non-empty edges (edges with different left and right nodes). Keeping a copy of the chart on the heap also allows for more sophisticated indexing strategies that would otherwise be overwhelmed by the cost of copying the edge before matching. This puts to rest the misconception that Prolog-based parsers are necessarily inefficient because of copying overhead.
  - **Shallow cuts (if-then-else predicates)** have been added to the definite clause language
  - **Faster extensionalisation code**, particularly with grammars that have few or no extensional types,
  - **Faster subsumption checking code** for chart edges,
  - **ALE Source-level Debugger 3.0,** which has been integrated with the new SICStus 3.7 source-level debugger,

- More compile-time error and warning messages,
- Several bug corrections,
- An updated user's manual,
- An SWI Prolog port.

- September, 1998: ALE 3.1 is available. In this version:
  - **ALE is now substantially faster**. Extensively revised code, making better use of shallow cuts, first-argument indexing, and other tricks of the Prolog trade,
  - **A new, faster, term-expansion-based compiler**. ALE now uses only one zero-byte intermediate file to compile its intermediate code. Intermediate code is saved for future use by saving the Prolog database itself,
  - **ALE Source-level Debugger 2.0**, which now supports semantic-head-driven generation,
  - **Lexicon compile/consult options** (lex_compile/0, lex_consult/0). Consulting the lexicon substantially reduces compilation time, with negligible consequences at run-time for most grammars,
  - **New commands to modify the lexicon incrementally** (update_lex/1, retract_lex/1). Also export_words/2, to write the words of a lexicon to a stream,
  - **New parsing commands** to filter solutions through a description and for batch parsing (rec/2,5, rec_best/2, rec_list/2,3),
  - **Prolog character escapes** can be used in conjunction with ALE,
  - Several bug corrections,
  - An updated user's manual,
  - An updated SWI Prolog port.

Contents

Overview

ALE 3.2, a freeware system written in Prolog by Bob Carpenter, and Gerald Penn integrates phrase structure parsing, semantic-head-driven generation and constraint logic programming with typed feature structures as terms. This generalizes both the feature structures of PATR-II and the terms of Prolog II to allow type inheritance and appropriateness specifications for features and values. Arbitrary constraints may be attached to types, and types may be declared as having extensional structural identity conditions. Grammars may also interleave unification steps with logic program goal calls (as can be done in DCGs), thus allowing parsing to be interleaved with other system components. ALE was developed with an eye toward Head-Driven Phrase Structure Grammar (HPSG), but it can also execute PATR-II grammars, definite clause grammars (DCGs), Prolog, Prolog-II, and LOGIN programs, etc. With suitable coding, it can also execute several aspects of Lexical-Functional Grammar (LFG).

The terms involved in ALE grammars and logic programs are specified using a typed extension of Rounds-Kasper attribute-value logic, which includes variables, full disjunction, inequations, and functional descriptions. Programs are then compiled into low-level Prolog instructions corresponding to the basic operations of the typed Rounds-Kapser logic. There is a strong type discipline enforced on descriptions, allowing many errors to be detected at compile-time.

The logic programming, parsing, and generating systems may be used independently or together. Features of the logic programming system include negation, disjunction and cuts. It has last call optimization, but does not perform any argument indexing.

The phrase structure system employs a bottom-up, all-paths dynamic chart parser. A general lexical rule component is provided, including procedural attachment and general methods for orthographic transformations using pattern matching or Prolog. Empty categories are permitted in the grammar. A mini-interpreter is included for stepping through the parsing process. Both the phrase structure and logic programming components of the system allow parametric macros to be defined and freely employed in descriptions. The language allows hooks to general Prolog routines, allowing the grammars and programs to be embedded in Prolog, and thus also in C and Unix. Parser performance is similar to that of the logic programming system.

The generation component also uses the phrase structure system plus a user-defined semantics definite clause that identifies semantically relevant information in feature structures. It uses an adaptation of the algorithm presented in:

Shieber, S., van Noord, G., Moore, R., and Pereira, F. (1990). Semantic-head-driven Generation. *Computational Linguistics*, 16.

This algorithm is very well suited to large-scale HPSG generation, since it avoids the non-termination problems inherent to top-down processing of lexicocentric theories, but at the same time, does not require the semantic contribution of every daughter in a grammar rule to subsume some portion of the semantic contribution of the mother, as, for example, is the case with Earley-based strategies. Some example glue code is provided to show how to attach two SICStus Prolog processes, one running a parser and one running a generator, to build a simple machine translation with ALE.

The source-level debugger can be used to trace nearly every operation in ALE's built-in parser, generator and definite clause resolver down to the level of feature structure unification. Based upon the procedural box model of control flow, it supports breakpoints, skipping, and leashing, and is compatible with ALE's built-in chart interpreter for navigating a parsing chart, as well as SICStus Prolog's own source-level debugger for on-line debugging of Prolog hooks. When used with its XEmacs interface, the buffer will indicate the current line being processed during execution. In a future release, the ALE source-level debugger will be extended to operations below the level of feature structure unification.

Documentation

Complete documentation for ALE 3.2.x (running to over 130 pages, with examples of everything, programming advice, and sample grammars) is available in postscript and LaTeX format in the general ALE release.

This document is also available in HTML format by clicking:

> ALE 3.2 User's Guide, HTML Version (coming soon!)

ALE is based on the typed attribute-value logic and associated grammar and constraint logic programming model developed in:

> Bob Carpenter(1992) *The Logic of Typed Feature Structures*. Cambridge Tracts in Theoretical Computer Science 32. Cambridge University Press.

System Requirements

ALE was developed with SICStus Prolog 3.8.6, but only requires that if the source-level debugger is used. For cyclic feature structures, at least version 2.1#8 is required. ALE itself is rather compact and takes up about 300K when compiled. The size of grammar that can be compiled and string that can be parsed will be determined by the size and structure of the grammar itself.

The efficiency of the system depends on first-argument indexing, last-call optimization, and for the chart parser, the indexing of dynamic clauses. The system will be very slow if it is interpreted rather than compiled.

The ALE source-level debugger requires SICStus Prolog 3.8.6. Its XEmacs interface requires XEmacs 20.3 or higher.

The SWI port of ALE can be run on SWI Prolog 4.0 or higher.

The Quinuts port of ALE 3.0 was the last port of ALE made to Quintus Prolog. Its behaviour with cyclic feature structures is unpredictable; and there are some hard restrictions on the number of variables in compiled clauses that can make the development of grammars with large descriptions in rules, lexical rules or the lexicon very difficult.

Obtaining ALE

ALE and its documentation are available on the GNU Lesser General Public License. The components of the ALE system are available at this address:

> http://www.cs.toronto.edu/~gpenn/ale/files/

The contents of the directory are as follows; the grammars are described in the ALE Grammars Section below.

ale.pl

> The source code for ALE 3.2.1, for use with SICStus Prolog 3.8.6.

ale.swi.pl

The SWI port of ALE 3.2.1, for use with SWI Prolog 4.0 or higher. To download SWI Prolog, choose Unix (version 4.0.11) or Windows (version 4.0.9).

aleguide.ps.gz

User's guide in compressed postscript format for US letter paper

aleguideA4.ps.gz

User's guide in compressed postscript format for A4 paper

debugger.tar.gz

The ALE source-level debugger and XEmacs interface, for use with SICStus 3.8.6 and XEmacs 20.3 or higher.

glue.tar.gz

Example glue code files to attach two SICStus processes running ALE together to build a simple machine translation system.

guide.tex

> User's guide in LaTeX 2e format; also available online (soon) as ALE 3.2 User's Guide, HTML Version

figs.tar.gz

Some encapsulated postscript figures that you will need to latex guide.tex yourself, along with the style file, epsf.sty

ale31.pl

The previous version of ALE (SICStus).

ale.qui.pl
The Quintus port of ALE 3.0.
ale31.swi.pl
The previous version of ALE, ported to SWI Prolog.
debugger20.tar.gz
The previous version of the source-level debugger and XEmacs interface, for use with SICStus 3.0.6, and XEmacs 19.16 or higher.

ALE Grammars

Sample Grammars

The source code of the system is distributed with a number of sample grammars, which are available at this address:

http://www.cs.toronto.edu/~gpenn/ale/files/grammars

The grammars, along with a description and their file names are as follows.

Metrical Phonology Syllabification Grammar (syllab.pl)
A small grammar for generating syllable structures of English words by coding sonority contours, the maximal onset principle, and phonotactic constraints via monostratal logical descriptions. (For more on constraint-based phonology, see the Computational Morphology and Phonology Page .)
Categorial Grammar with Cooper Storage (cg.pl)
A small unification categorial grammar with forward and backward slashes illustrating the use of procedural attachments.
HPSG 2.0 (hpsg.pl)
A fairly comprehensive implementation of a Head-Driven Phrase Structure Grammar for English, following (Pollard and Sag 1994:Chapters 1--5, 7, and 8). This grammar has changed one of its feature names as of ALE 3.1, to work around a problem with SWI Prolog's operator parsing, and now contains discontiguity declarations. For the purposes of benchmarking, it behaves identically to the old hpsg.pl.
Sample Grammar from "Semantic-Head-Driven Generation" (gengram.pl)
The example grammar from the *Computational Linguistics* article of Shieber et al., adapted to the logic of typed feature structures.
Zebra Puzzle (baby.pl)
Not a grammar per se, but rather a direct coding of a simplified version of the Zebra Puzzle, a simple logic puzzle stated with constraints, for illustrating the power of pure constraint resolution.

Third-Party User Interfaces

HDrug

The HDrug system, developed by Gertjan van Noord, is a package for developing logic grammars implemented on top of SICStus Prolog 3.0 or later, and Tcl/Tk. It has been tested on HP-UX, Linux and Sun platforms. Information is available for it on the:

HDrug Homepage

HDrug has extensive features for performance evaluation, including plotting of statistical information. It contains methods for viewing parses as they are created, modifying them, and comparing them with other parses. It also contains an on-line manual.

The HDrug release includes a number of utilities for grammar formalisms other than ALE, including tree adjoining grammar, definite clause grammars, categorial grammar, head-driven generation grammar, and extraposition grammars. Also interesting is Gertjan's joint work with Gosse Bouma on delayed evaluation for lexical rules in the context of an HPSG grammar for Dutch.

The Pleuk Grammar Development Environment

For those using SICStus 2.1#9 running under X windows, the Pleuk grammar development shell has been adapted for ALE. Pleuk is described in the following technical report.

Calder, Jo (1993) Graphical Interaction with Constraint-Based Grammars. In *Proceedings of the First Conference of the Pacific Association for Computational Linguistics*(PACLING '93), Vancouver, BC, Canada. pps. 160--168.

Pleuk provides a graphical user interface, facilities for maintaining and testing corpora, and an interactive, incremental derivation checker. Pleuk is available free of charge from:

ftp://ftp.cogsci.ed.ac.uk/pub/pleuk

The file README contains instructions for downloading the system. Pleuk has been ported to Sun SPARCs SunOS 4.* and HP-UX. For more information, send e-mail to pleuk@cogsci.ed.ac.uk. Pleuk was developed by Jo Calder and Chris Brew of the Human Communication Research Centre (HCRC) at the University of Edinburgh, Kevin Humphreys of the Centre for Cognitive Science at the University of Edinburgh, and Mike Reape, of the Computer Science Department, Trinity College, Dublin.

Emacs Interface

Olivier Laurens is distributing an Emacs-based interface to ALE. It is described in a technical report (in compressed postscript format):

Laurens, Olivier (1995) An Emacs User Interface for ALE. Technical Report CSS-IS TR 95-07, School of Computing Science, Simon Fraser University, Burnaby, BC, Canada, June 1995. ftp://fas.sfu.ca/pub/cs/nl/emacs-ale/tr95-07.ps.Z(20pps.)

It is also available free of charge for research purposes via ftp as a compressed, tarred directory:

ftp://fas.sfu.ca/pub/cs/nl/emacs-ale/ale_emacs_1.tar.Z

Teaching Materials

- Course on HPSG Grammars and Typed Feature Formalisms

December 11--14, 1995
Human Communication Research Centre, University of Edinburgh

- Course Notes on HPSG in ALE, created by Colin Matheson, Centre for Cognitive Science , University of Edinburgh
- Computational Morphology: An Introduction to ALE-RA, created by Colin Matheson, Centre for Cognitive Science, University of Edinburgh. Based on the ALE-RA system, created by Tomaz Erjavec.

Mailing List and Feedback

Please send all bug reports to gpenn+@cmu.edu.

If you'd like to be put on the ALE mailing list and be informed of updates, new grammars, and so on, send e-mail to Gerald Penn at gpenn+@cmu.edu.

If you would like to post to the ALE Mailing list, it can be found at:

+dist+~gpenn/dl/ale.dl@andrew.cmu.edu

You can alias this list to something more readable, or just click on the address above to send a mail if your browser supports forms.

Please let us know if you have any comments, suggestions, or questions. We'd be especially interested to hear what you are doing with ALE. As an added bonus, if you would like, we can link your project into this page so that other people can find out about it; just send us a description and a URL if you have one.

ALE Users' Project Reports

- Frederik Fouvry's report on Dutch HPSG and his thesis.
- Ion Androutsopoulos' thesis on natural language interfaces to temporal databases, including prototype source code.

About ALE

ALE was developed in the Computational Linguistics Program and the Language Technologies Institute at Carnegie Mellon University by Bob Carpenter and Gerald Penn. The semantic-head-driven generator is based on a generator written for ALE by Octav Popescu.

Gerald Penn, gpenn+@cmu.edu, October 2003