BUP

A Bottom Up Parser

Tim Finin and Bonnie Lynn Webber

Computer and Information Science
The Moore School
University of Pennsylvania
Philadelphia, PA

report MS-CIS-83-18

2 November 1984

Abstract

BUP is a bottom up syntactic analyser that can be used in several ways: as a simple bottom up parser for context free languages, as a bottom up parser for extended phrase structure grammars (equivalent in power to an ATN), or as a transducer for either. BUP accepts a grammar and lexicon specified as rules. It will then analyze input strings according to those rules, recording its findings in a ***chart*** and producing all analyses in parallel. Rules can be displayed, added or deleted incrementally. BUP provides a small set of tools for building structures and specifying tests.

key words and phrases: parsing, chart parsing, bottom-up parsing, natural language processing, computational linguistics.

1. Introduction

BUP is a bottom up syntactic analyser that can be used in several ways: as a simple bottom up parser for context free languages, as a bottom up parser for extended phrase structure grammars (equivalent in power to an ATN), or as a transducer for either. BUP accepts a grammar and lexicon specified as rules (Section 3). It will then analyze input strings according to those rules, recording its findings in a ***chart*** and producing all analyses in parallel. Rules can be added and deleted incrementally (Section 5). Section 4 describes tools BUP provides for building structures and specifying tests. Section 6 gives some examples of BUP grammars.

1.1. Accessing BUP

The Franz Lisp version of BUP is in *UPENN::[finin.bup]bup.l.* Example grammars can be found in *UPENN::[finin.bup]grammars.l.*

You can load BUP into Maryland Lisp on the MSCF's Univac 1100 via:

```
@ADD FININ*BUP.BUPPER
```

The sample grammars described in the last section of this note can be loaded via:

```
@ADD FININ*BUP.GRAMMARS
```

2. Theory of Operation

The parsing stategy embodied in BUP is similar to that described in [pratt73] and [pratt75]. For a good general treatment of parsing, see [winograd]. etc. etc. etc.

3. Rule Structure

BUP rules are represented as lists which have at least two components corresponding to the ***Left Hand Side*** (LHS) and ***Right Hand Side*** (RHS) of a standard CF rule. This is true whether the rule is to represent a phrase structure rule like S ⇒ NP VP, written (S (NP VP)), or a lexical entry like N ⇒ dog, written (N dog). In the VAX implementation, the RHS of a rule may be either a list or an atom. However, on the 1100, it must be a list, even if it only contains a single element - e.g., a rule like VP ⇒ V would be written (VP (V)). If no rule has more than two components, BUP will act like a simple bottom up CF parser. Figure 1 shows a simple context free grammar for a trivial subset of English in the common format and in BUP's format.

BUP includes a special facility for dealing with "unknown words". BUP allows the value of the reserved atom "wordSymbol", initially set to <word>, as a RHS term. The value of wordSymbol is a wild-card which will match any LISP expression. Normally, it would be used along with a ***test*** attribute on a rule,

Figure 3-1: a simple CFG grammar

```
S   ⇒  NP VP          V   ⇒  chased
NP  ⇒  DET N          V   ⇒  ate
NP  ⇒  N              N   ⇒  man
NP  ⇒  NP PP          N   ⇒  woman
VP  ⇒  V              N   ⇒  dog
VP  ⇒  V NP           N   ⇒  cat
VP  ⇒  VP PP          DET ⇒  the
                         DET ⇒  a

(addrules

  (S (NP VP))            (V chased)
  (NP (DET N))           (V ate)
  (NP (N))               (N man)
  (NP (NP PP))           (N woman)
  (VP (V))               (N dog)
  (VP (V NP))            (N cat)
  (VP (VP PP))           (DET the)
                         (DET a)
 )
```

a facility which we will describe later. The following rule, for example, can be used to recognize numbers:

```
(NUMBER <word> #1 -if (numberp #1))
```

This rule will have a structural match with any single LISP expression, and will apply whenever the derived structure associated with that expression (i.e., #1) is a number.

BUP rules may have, in addition, one to four optional components. The first optional component allows arbitrary ***derived structures*** to be built, making BUP into a ***transducer***. For example, BUP can be used to output a ***functional description*** of its input (i.e., in terms of roles like "subject", "object", "theme", etc.), a ***case analysis*** description of its input (i.e., in terms of rules like "agent", "benefactor", "instrument", etc.) or a logical analysis (i.e., in terms of predicates, quantifiers, variable bindings, etc.).

This first component, if present, would appear right after the rule's RHS. We sometimes refer to it as the "generator". It can be any LISP expression. If the rule matches, the generator expression will be evaluated, and the resulting derived structure will be bound to the LHS term just found. (Note that this derived structure ***coexists*** with the syntactic structure associated with the term: it does not replace it.)

While a generator expression can be any LISP form, special functions are provided which support a compositional approach to derived structure. These functions allow access to information associated with each of the RHS constituents used in instantiating the rule. Thus, the structure generated for the LHS term can depend upon the structures built for its RHS terms (and, as will be seen, their ***features*** as well). These access functions are described in Section 4.1.

In addition to the generator expression, there are three other optional components of a rule: a ***test*** expression, an expression which evaluates to the ***features*** to be associated with the new LHS term, and an expression which evaluates to a ***weight*** to be associated with it. They can be specified in any order: however, the test expression should be preceded by either "-t" or "-if"; the features expression should be preceded by "-f", and the weight expression, by "-w". For example,

```
    (V chased 'chase -f '(transitive))
    (VP (V NP) -if (feat 1 'transitive) -f (list (feat 1 'number)))
```

Here, in addition to its LHS and RHS components, the first rule specifies both a derived structure for its new LHS term V (i.e., the atom "chase") and features for it - i.e., "(transitive)". The second rule specifies both a test and features: it does not explicitly specify a derived structure to be associated with the new VP. As we shall see, BUP provides ***default procedures*** for building a term's derived structure, as well as its features and its weight.

As for what these three remaining optional components represent, the ***test*** specifies additional conditions that must hold (over and beyond the structural match) in order for the rule to apply. If the ***test*** evaluates to the value of the reserved atom FAILURE (initially set to NIL), the rule fails. It is the presence of this component that turns the grammar from simply CF to an extended phrase structure grammar (PSG) with Turing Machine power. For example, the second rule above would not apply unless the verb (V) that matched had the feature "transitive". The test can be any LISP expression, but again access functions (described in Section 4.1) are provided for specifying which, if any, structures or features the test depends on.

(Actually, if any of these four optional components evaluates to FAILURE, the rule will not apply. This is not theoretically motivated, but rather found to be convenient. FAILURE is discussed more in Section 4.1.)

The ***features*** component is an arbitrary LISP expression that evaluates to the list of features to be associated with the newly built LHS term. A feature is either a ***feature name*** like "transitive" or a ***feature-value pair*** like "(number plural)". Like the derived structure component, there is a default procedure for combining the features associated with the RHS terms in the rule to produce a set of features associated with the LHS.

Finally, the ***weight*** component is also an arbitrary LISP expression that evaluates to a weight to be associated with the new LHS term. If it is not specified explicitly, the system will use a default procedure. Since BUP produces all possible analyses in parallel, these weights cannot be used for deciding what path to follow (as they might be if the analyses were being proposed sequentially): Rather, they can be used for rank-ordering the analyses after they've been done.

As noted above, BUP provides default procedures for computing a derived structure, features and/or weight for a LHS term ***if*** such expressions are not specified explicitly ***and*** the constituents matching the RHS terms have their own derived structures, features and/or weights. These procedures are specified as the values of the reserved atoms

defaultRuleSemantics	(for generating derived structures)
defaultRuleFeatures	
defaultRuleWeights	

The first constructs a list consisting of the LHS term followed, in order, by the derived structure associated with each of its RHS terms. For example, consider the rule (VP (V NP)), which does not explicitly specify a derived structure. Suppose V is associated with the derived structure (Relation CHASE) and NP is associated with the derived structure (NP DOGS). Since the rule does not specify a generator, the defaultRuleSemantics procedure will produce (VP (Relation CHASE)(NP DOGS)).

The defaultRuleFeatures procedure is straightforward: it appends together the set of features associated with each of the RHS terms. The defaultRuleWeights procedure produces an average of the weights associated with each of the RHS terms. Since we have not made great use of weights, this may change if and when we do. (N.B. There is also a defaultRuleTest procedure, whose value is T. One might want to change this, but it's not obvious why.)

4. Expressing Tests, Derived Features & Derived Structures

4.1. Access Functions

BUP provides a set of functions for accessing the syntactic structure, derived structure, features and/or weight associated with the constituents matching the RHS terms. These access functions can be used for expressing additional criteria that must be met for a rule to match (i.e., its ***test***) or for expressing the derived structure, features and/or weight to be associated with the new LHS term. These functions are:

```
(syn <N>)                              access to syntactic structure
(sem <N>)                              access to derived structure
(feat <N>)                             |
(feat <N> <FEAT>)                      |- access to features
(feat <N> <FEAT> <FEATVAL>)            |
(weight <N>)                           access to weight
```

The argument <N> must be either -1, 0 or a positive integer. If <N> is a positive integer, the function will return information for the Nth RHS term. The functions behave somewhat differently for N=0. The value of (syn 0) is a list consisting of the LHS term followed by the syntactic structure of each RHS term. The value of (sem 0) is a list consisting of the LHS term followed by the derived structure associated with each RHS term. The value of (feat 0) is a list of the feature set of each RHS term, and the value of (weight 0) is simply a list of numbers, the weight of each RHS term. N=-1 is defined the same for all four functions: the value is simply the LHS term.

The function ***feat*** has two additional forms: if a second argument <FEAT> is given, what is accessed is the <FEAT> feature of the Nth RHS term, and what is returned is its value. If a third argument <FEATVAL> is given as well, ***feat*** acts as a test of whether the value of the Nth RHS term's FEAT feature is FEATVAL. (N.B. "sem" may be abbreviated "#", in which case the expression need not be in parentheses - i.e., (sem 1) and #1 are equivalent.

These syn, sem, feat and weight functions may be used in accessing information, say for specifying tests or for passing on features/weights. For example, consider again the rule

```
(VP (V NP) -if (feat 1 'transitive) -f (list (feat 1 'number)))
```

This rule uses ***feat*** in two places: first, to test that the first term on the RHS - V - has the feature "transitive", and second, to associate V's number feature and its value (e.g., (number pl)) with the new VP. Notice that if the verb isn't transitive and the test (feat 1 'transitive) evaluates to the value of FAILURE - i.e., NIL - the rule will fail to match.

4.2. Structure Building

In addition to their use in specifying tests, features and weights, the above functions may be used for building new derived structures. Structure building makes use of the backquote feature intrinsic to dialects of Lisp in the Common Lisp family (e.g. MacLisp, FranzLisp LispMachineLisp, ZetaLisp, NIL Lisp) and imported to Maryland Lisp on the 1100. In the Common lisp implementations, the baxkquote character is "'" and in the Maryland Lisp implementation it is the caret or *up arrow* symbol "^". Within a backquoted list, all expressions taken as literals (e.g are quoted) except those preceded by the symbols "," or ",.". A Comma (,) by itself causes the value of the next expression to be ***inserted into*** the list. Used in conjuction with ".", comma causes the value to be ***spliced in***. For example, suppose the value of A is (1 2 3) and the value of B is (4 5 6). Then

```
^(A ,A B ,B)      has the value    (A (1 2 3) B (4 5 6))
^(A ,.A B ,.B)    has the value    (A 1 2 3 B 4 5 6)
^(A ,(car A) B ,(car B)) has the value  (A 1 B 4)
```

To show the use of access functions together with structure building, consider the following four-part rule:

```
(VP (V NP NP) ^(,#1 (OBJECT ,#3) (RECIPIENT ,#2)) -if (feat 1 'dative))
```

Suppose this rule matches the syntactic structure assigned to "gave the poor dog a bone". Suppose also that the derived structure assigned to the verb "gave" is simply "gave", that assigned to "the poor dog" is (dog (MODIFIER poor) the), and that assigned to "a bone" is (bone a), then the derived structure built for the new VP will be

```
(gave (OBJECT (bone a)) (RECIPIENT (dog (MODIFIER poor) the))))
```

5. Interactive Functions

BUP proves a number of functions for maintaining the grammar (by adding and deleting rules), exploring the results of parsing a sentence (i.e. examining the chart) and tracing the application of some or all of the grammar's rules.

5.1. Specifying Patterns

Several of the interactive functions described in this section (in particular, rules for showing, deleting and tracing rules) take a ***rule pattern*** as argument. A rule pattern is an atom or a 2-tuple in which the atoms "?" and "*" take on special significance. In matching a rule against a pattern, a "?" can match any single element and a "*" can match a (possibly empty) sequence of elements. (Only the LHS and RHS of a rule are described in a rule pattern. Hence a pattern will either be the atom * for all rules or a 2-tuple.) Some examples are:

```
(PP ?)          matches all rules which build PPs
(? (* PP *))    matches all rules which use PPs
*                       matches all rules
```

5.2. Function Descriptions

The following gives a brief description of the functions used to define grammars and parse sentences with them. Arguments which are evaluated are shown as quoted.

(PARSE w1 w2 ... wn)
: Parse the sentence with words w1...wn and print all possible analyses, where an analysis consists of a syntactic structure and a possibly empty derived structure, feature set and weight.

(PARSE)
: Re-parse the last sentence - i.e., the value of the reserved atom "oldSentence".

(PARSE1 's1 'c1)
: Parse s1 as a c1, where s1 evaluates to a list of words and c1 evaluates to a non-terminal symbol. For example,

```
(PARSE1 '(the man) 'NP)         parses (the man) as an NP
(PARSE1 '(the man eats) 'S)     == (PARSE the man eats)
```

(ADDRULES r1 r2 ... rn)
: Add the rules r1, r2 ... rn to the current grammar.

(DELETERULES p1)
: Delete all rules which match the pattern p1 from the current grammar. For example,

```
(DELETERULES (PP ?))       delete all rules which build PP's
(DELETERULES (? (* PP *))) delete all rules using PP's.
(DELETERULES *)               delete all rules in the grammar.
```

(RULES 'p)
: Returns a list of rules matching pattern p.

(SHOWRULES p1)
: Print (on the terminal) all rules which match the pattern p1. (SHOWRULES *) will, of course, print all rules in the grammar.

(SHOWTERMINALS)
: Display the lexicon - i.e., the list of all terminal symbols used in the grammar

(SHOWNONTERMINALS)
Display the non-terminal symbols used in the grammar.

(TRACERULES p1)
Shows the application of each rule matching pattern p1, as it is used in parsing. A message of the following form is printed:

```
[i,j] A ⇒ B
```

where i is the starting position, j is the final position, A is the type of constituent found and B is the structure generated for it. Positions fall between words, i.e. position 0 is just before the first word, position 1 is between the first and second words, etc.

(UNTRACERULES p1)
turns off the trace from those rules matching pattern p1.

(SHOWCHART from to s1 s2 ... sn)
Displays a portion of the chart built up in parsing the last sentence. The initial and final positions are given by FROM and TO. These can be integers or the atom "?", which matches any position. The Si name the constituent types we want displayed.

```
(SHOWCHART)          - shows all entries.
(SHOWCHART 0 ?)      - shows all entries which begin at 0,
                       end anywhere and are of any type.
(SHOWCHART ? ? S NP) - shows all constituents of type S or NP,
                       regardless of where they begin or end.
```

(HELP topic1 topic2 ...)
Prints some helpful information on the given topics. Try (HELP *) to see a list of topics for which help is available.

(example0), (example1), (example2), (example3)
Define example grammars that one might experiment with. These four functions may be found on the file [tim.bup]grammars.l (on the VAX) and ---- (on the Univac 1100).

6. Example Grammars

The following figures define some simple grammars with examples of their operation.

7. Bibliography

[Pratt 73] Pratt, V.
LINGOL ...
In *Proc. 3ed International Joint Conference on Artificial Intelligence.* IJCAI, August, 1973.

[Pratt 75] Pratt, V.
LINGOL, A Progress Report.
In *Proc. 4th International Joint Conference on Artificial Intelligence.* IJCAI, August, 1975.

[Winograd 83] Winograd, T>.
Language as a Cognitive Process.
Addison-Wesley Publishing Co., Inc., 1983.

Figure 6-1: grammar 0 - trivial syntax

```
(addrules
  (S (NP VP))

  (NP (DET NP1))
  (NP (NP1))
  (NP1 (ADJ NP1))
  (NP1 (N))
  (NP1 (NP1 PP))

  (VP (V))
  (VP (V NP))
  (VP (V NP NP))
  (VP (VP PP))

  (PP (PREP NP))

  (DET the) (DET a)
  (V chased) (V ate) (V gave) (V liked) (V believes) (V knows)
  (V thinks) (V put) (N man) (N woman) (N dog) (N cat) (N fish)
  (N table) (N floor) (ADJ large) (ADJ hungry) (PREP on) (PREP near)
  (PREP in) (PREP to))

> (parse the man ate fish)

< ((S (NP (DET the) (NP1 (N man))) (VP (V ate) (NP (NP1 (N fish)))))
   nil
   nil
   1.0)
```

Figure 6-2: grammar 1 - simple syntax

```
(addrules
   (S (NP VP))

   (factiveS (that S))
   (factiveS (S))

   (NP (DET NP1))
   (NP (NP1))
   (NP1 (ADJ NP1))
   (NP1 (N NP1))
   (NP1 (N))
   (NP1 (NP1 PP))

   (VP (V) -if (feat 1 'intransitive))
   (VP (V NP) -if (feat 1 'transitive))
   (VP (V factiveS) -if (feat 1 'factive))
   (VP (V NP NP) -if (feat 1 'dative))
   (VP (VP PP))
   (VP (VP ADV))
   (VP (ADV VP))

   (PP (PREP NP))

   (DET the)
   (DET a)
   (V chased  -f '(transitive))
   (V ate  -f '(transitive intransitive))
   (V gave  -f '(transitive dative intransitive))
   (V liked -f '(transitive))
   (V believes -f '(transitive factive))
   (V knows -f '(transitive factive))
   (V thinks -f '(transitive factive))
   (V put) (N man) (N woman) (N dog) (N cat) (N fish) (N table)
   (N floor)(ADJ large) (ADJ hungry) (ADV quickly) (ADV quietly)
   (ADV very)(PREP on)(PREP near)(PREP in)(PREP to))

(parse the man ate fish)

((S (NP (DET the) (NP1 (N man))) (VP (V ate) (NP (NP1 (N fish)))))
 nil
 (transitive intransitive)
 1.0)
```

Figure 6-3: grammar 2 - more complex syntax

```
(addrules
   (S  (NP VP))

   (factiveS (that S))
   (factiveS (S))

   (RELS (that S/NP))
   (RELS (who S/NP))
   (RELS (S/NP))

   (S/NP (VP))
   (S/NP ( NP VP/NP))

   (NP (DET NP1))
   (NP (NP1))
   (NP1 (ADJ NP1))
   (NP1 (N NP1))
   (NP1 (N))
   (NP1 (NP1 PP))
   (NP1 (NP1 RELS))

   (VP (V) -if (feat 1 'intransitive))
   (VP (V NP) -if (feat 1 'transitive))
   (VP (V factiveS) -if (feat 1 'factive))
   (VP (V NP NP) -if (feat 1 'dative))
   (VP (VP PP))
   (VP (VP ADV))
   (VP (ADV VP))

   (VP/NP (V) -if (feat 1 'transitive))
   (VP/NP (V NP) -if (feat 1 'dative))
   (VP/NP (V NP) -if (feat 1 'dative))

   (PP (PREP NP))

   (DET the)
   (DET a)
   (V chased 'chased -f '(transitive))
   (V ate 'ate -f '(transitive intransitive))
   (V gave 'gave -f '(transitive dative intransitive))
   (V liked 'liked -f '(transitive))
   (V believes 'believes -f '(transitive factive))
   (V knows 'knows -f '(transitive factive))
   (V thinks 'thinks -f '(transitive factive))
   (V put) (N man) (N woman) (N dog) (N cat) (N fish) (N table)
   (N floor) (ADJ large) (ADJ hungry) (ADV quickly) (ADV quietly)
   (PREP on) (PREP near) (PREP in) (PREP to))
```

Figure 6-4: grammar 3 - simple semantics

```
(addrules
   (S  (NP VP) ^(,#2 (SUBJECT ,#1)))
   (factiveS (that S) #2)
   (factiveS (S) #1)
   (RELS (that S/NP) #2)
   (RELS (who S/NP) #2)
   (RELS (S/NP) #1)
   (S/NP (VP) ^(,.#1 (SUBJECT <trace>)) -if (feat 1 'intransitive))
   (S/NP ( NP VP/NP) ^(,.#2 (SUBJECT ,#1)))
   (NP (DET NP1) ^(,.#2 ,#1))
   (NP (NP1) #1)
   (NP1 (ADJ NP1)  ^(,.#2 (MODIFIER ,#1)))
   (NP1 (N NP1) ^(,.#2 (MODIFIER ,#1)))
   (NP1 (N) ^(,#1))
   (NP1 (NP1 PP) ^(,.#1 (MODIFIER ,#2)))
   (NP1 (NP1 RELS) ^(,.#1 (MODIFIER ,#2)))

   (VP (V) ^(,#1) -if (feat 1 'intransitive))
   (VP (V NP) ^(,#1 (OBJECT ,#2)) -if (feat 1 'transitive))
   (VP (V factiveS) ^(,#1 (OBJECT ,#2)) -if (feat 1 'factive))
   (VP (V NP NP) ^(,#1 (OBJECT ,#3) (RECIPIENT ,#2)) -if (feat 1 'dative))
   (VP (VP PP) ^(,.#1 (MODIFIER ,#2)))
   (VP (VP ADV) ^(,.#1 (MODIFIER ,#2)))
   (VP (ADV VP) ^(,.#2 (MODIFIER ,#1)))
   (VP/NP (V) ^(,#1 (OBJECT <trace>)) -if (feat 1 'transintive))
   (VP/NP (V NP) ^(,#1 (OBJECT <trace>) (RECIPIENT ,#2)) -if (feat 1 'dative))
   (VP/NP (V NP) ^(,#1 (OBJECT ,#2) (RECIPIENT <trace>)) -if (feat 1 'dative))
   (PP (PREP NP) ^(,#1 ,#2))

   (DET the)(DET a)
   (V chased 'chased -f '(transitive))
   (V ate 'ate -f '(transitive intransitive))
   (V gave 'gave -f '(transitive dative intransitive))
   (V liked 'liked -f '(transitive))
   (V believes 'believes -f '(transitive factive))
   (V knows 'knows -f '(transitive factive))
   (V thinks 'thinks -f '(transitive factive))
   (V put 'put) (N man 'man) (N woman 'woman) (N dog 'dog)
   (N cat 'cat) (N fish 'fish) (N table 'table) (N floor 'floor)
   (ADJ large 'large) (ADJ hungry 'hungry) (ADV quickly 'quickly)
   (ADV quietly 'quietly) (PREP on 'on)
   (PREP near 'near) (PREP in 'in) (PREP to 'to))

(parse the man ate fish)

((S (NP (DET the) (NP1 (N man))) (VP (V ate) (NP (NP1 (N fish)))))
 ((ate (OBJECT (fish))) (SUBJECT (man nil)))
 (transitive intransitive)
 1.0)
```

Figure 6-5: grammar 4 - a semantic grammar for data bases

```
(addrules
        (PROPERTY-NP (DET PROP) #2)
        (PROPERTY-NP (PROPERTY) #1)
        (PROPERTY age '(quote age))
        (PROPERTY spouse '(quote spouse))
        (PROPERTY height '(quote height))

        (DET the)
        (DET a)
        (DET an)

        (WH-WORD what)
        (WH-WORD who)
        (WH-WORD which)

        (POSSESIVE ('s))

        (OBJECT PATH (list 'get (cadr #1) (car #1)))
        (OBJECT john '(quote john))
        (OBJECT mary '(quote mary))
        (OBJECT NUMBER #1)
        (OBJECT NEW-WORD #1 -w -1)
        (OBJECT LISP-LIST #1)
        (LISP-LIST <word> #1 -if (not (atom #1)))

        (NUMBER <word> #1 -if (numberp #1))

        (PATH (PROPERTY-NP of OBJECT) (list #1 #3))
        (PATH (OBJECT POSSESIVE PROPERTY-NP) (list #3 #1))

        (S (WH-WORD is OBJECT) #3)
        (S (is OBJECT OBJECT) (list 'equal #2 #3))
        (S (PATH is OBJECT)
           (list 'putprop (cadr #1) #3 (car #1)))
        (S (OBJECT is PATH)
           (list 'putprop (cadr #3) #1 (car #3)))
        (S (does OBJECT have PROPERTY-NP) ^(yes-no (get ,#2 ,#4)))
        (S (eval LISP-LIST) ^(eval ',#2))
        (S LISP-LIST ^(eval ',#1))


        (S (<word> is DET PROPERTY-NP) ^(addrules (PROPERTY (,#1))))

        (S (<word> is DET object) ^(addrules (OBJECT (,#1)))))
```

Figure 6-6: A Sample Session

```
$ lisp
Franz Lisp, Opus 37

1.(load "[tim.bup]bup")
[fasl [tim.bup]bup.o]
"type (HELP) for (outdated!) helpful information"
t

2.()
NIL

3.(load "[tim.bup]grammars.l")
t

4.(example3)
loading example grammar 3 - semantics
Deleting all rules.[*list:154{84%}; fixnum:1{79%}; ut:34%]
t

5.(parse the cat ate fish)

((S (NP (DET the) (NP1 (N cat))) (VP (V ate) (NP (NP1 (N fish)))))
 ((ate (OBJECT (fish))) (SUBJECT (cat the)))
 (transitive intransitive)
 1.0)
t

6.(parse the cat ate the fish on the table)

((S (NP (DET the) (NP1 (N cat)))
    (VP (VP (V ate) (NP (DET the) (NP1 (N fish))))
        (PP (PREP on) (NP (DET the) (NP1 (N table))))))
 ((ate (OBJECT (fish the)) (MODIFIER (on (table the)))) (SUBJECT (cat the)))
 (transitive intransitive)
 1.0)

((S (NP (DET the) (NP1 (N cat)))
    (VP (V ate)
        (NP (DET the)
            (NP1 (NP1 (N fish))
                 (PP (PREP on) (NP (DET the) (NP1 (N table))))))))
 ((ate (OBJECT (fish (MODIFIER (on (table the))) the))) (SUBJECT (cat the)))
 (transitive intransitive)
 1.0)
t
```

Figure 6-7: A Sample Session, Tracing and Examining the Chart

```
7.(tracerules '(NP *))
(R9 R10)

8.(parse)

Sentence is: (the cat ate the fish on the table)
R10 [1,2] NP -> ((NP1 (N cat)) (cat) nil 1.0)
R9 [0,2] NP -> ((DET the) the nil 1.0)
R10 [1,3] NP -> ((NP1 (NP1 (N cat)) (RELS (S/NP (VP (V ate))))) ...
R9 [0,3] NP -> ((DET the) the nil 1.0)
R10 [4,5] NP -> ((NP1 (N fish)) (fish) nil 1.0)
R9 [3,5] NP -> ((DET the) the nil 1.0)
R10 [1,5] NP -> ((NP1 (NP1 (N cat)) (RELS (S/NP (VP (V ate) (NP (DET the) ...
R9 [0,5] NP -> ((DET the) the nil 1.0)
R10 [7,8] NP -> ((NP1 (N table)) (table) nil 1.0)
R9 [6,8] NP -> ((DET the) the nil 1.0)
R10 [1,8] NP -> ((NP1 (NP1 (N cat)) (RELS (S/NP (VP (VP (V ate) ...
R9 [0,8] NP -> ((DET the) the nil 1.0)
R10 [1,8] NP -> ((NP1 (NP1 (NP1 (N cat)) (RELS (S/NP (VP (V ate) ...
R9 [0,8] NP -> ((DET the) the nil 1.0)
R10 [4,8] NP -> ((NP1 (NP1 (N fish)) (PP (PREP on) (NP (DET the) (NP1 ...
R9 [3,8] NP -> ((DET the) the nil 1.0)
R10 [1,8] NP -> ((NP1 (NP1 (N cat)) (RELS (S/NP (VP (V ate) (NP (DET the) ...
R9 [0,8] NP -> ((DET the) the nil 1.0)

((S (NP (DET the) (NP1 (N cat)))
    (VP (VP (V ate) (NP (DET the) (NP1 (N fish))))
        (PP (PREP on) (NP (DET the) (NP1 (N table))))))
 ((ate (OBJECT (fish the)) (MODIFIER (on (table the)))) (SUBJECT (cat the)))
 (transitive intransitive)
 1.0)

((S (NP (DET the) (NP1 (N cat)))
    (VP (V ate)
        (NP (DET the)
            (NP1 (NP1 (N fish))
                 (PP (PREP on) (NP (DET the) (NP1 (N table)))))))
 ((ate (OBJECT (fish (MODIFIER (on (table the))) the))) (SUBJECT (cat the)))
 (transitive intransitive)
 1.0)
t

9.(showchart 1 2)

[1,2] ((NP (NP1 (N cat))) (cat) nil 1.0) -> NP
[1,2] ((NP1 (N cat)) (cat) nil 1.0) -> NP1
[1,2] ((N cat) cat nil 1.0) -> N
[1,2] (cat cat nil 1) -> cat
[1,2] (cat cat nil 1) -> <word>
nil

10.(showchart ? ? s)
nil

11.(showchart ? ? S)

[1,8] ((S (NP (NP1 (N cat))) (VP (V ate) (NP (DET the) (NP1 (NP1 ...
[0,8] ((S (NP (DET the) (NP1 (N cat))) (VP (V ate) (NP (DET the) ...
[1,8] ((S (NP (NP1 (N cat))) (VP (VP (V ate) (NP (DET the) (NP1 ...
[0,8] ((S (NP (DET the) (NP1 (N cat))) (VP (VP (V ate) (NP (DET  ...
[1,5] ((S (NP (NP1 (N cat))) (VP (V ate) (NP (DET the) (NP1 (N  ...
[0,5] ((S (NP (DET the) (NP1 (N cat))) (VP (V ate) (NP (DET the) ...
[1,3] ((S (NP (NP1 (N cat))) (VP (V ate))) ((ate) (SUBJECT (cat))) ...
[0,3] ((S (NP (DET the) (NP1 (N cat))) (VP (V ate))) ((ate) (SUBJECT ...
```