CL-PATR Reference Manual

Technical Note 456

February 13, 1989

By: Stuart M. Shieber, Computer Scientist
Artificial Intelligence Center
Computer and Information Sciences Division
and
Center for the Study of Language and Information

APPROVED FOR PUBLIC RELEASE:
DISTRIBUTION UNLIMITED

The preparation of this paper and the research reported herein were supported in part by a contract with the Nippon Telegraph and Telephone Corporation, and in part by a gift from The System Development Foundation.

333 Ravenswood Ave. • Menlo Park, CA 94025
(415) 326-6200 • TWX: 910-373-2046 • Telex: 334-486

CL-PATR Reference Manual

Version 1.0

Stuart M. Shieber

Artificial Intelligence Center
and
Center for the Study of Language and Information
SRI International

February 13, 1989

Copyright © Stuart M. Shieber
Software Unpublished Copyright © SRI International
All Rights Reserved
CL-PATR and PATR are Trademarks of SRI International
July, 1988

Contents

Chapter 1

Introduction

The PATR-II grammar formalism has been developed over the last few years at SRI International as a grammar formalism for codifying fragments of natural language. Historically, PATR-II developed from the PATR (Parse, Annotate, and TRanslate) formalism [12]. The formalism has been used over the last several years in projects building natural-language-processing systems to express syntactic and semantic constraints on natural-language expressions.[1]

CL-PATR (Common LISP PATR) is the latest implementation of the PATR-II formalism developed at SRI. The CL-PATR implementation was motivated by a desire for a simple, reasonably efficient, and highly portable implementation of PATR-II. Because various implementations of PATR-II over the years had begun to diverge in the formalism actually used, we have developed a standard for the PATR-II language, called S-PATR; this is the version of the formalism actually implemented in the CL-PATR system. S-PATR contains various extensions to previous PATR formalisms that have been found in the past to be desirable. Written in pure Common LISP [4], CL-PATR has already been ported to Symbolics, Sun, Macintosh, and IBM PC-RT computers.

1.1 Prerequisites

This manual is intended to be used for reference by users of the CL-PATR system, and is therefore not self-contained. Readers are assumed to have some familiarity with the PATR formalism and its use for expressing grammars, as well as an understanding of chart parsing methods. The introductory text on unification-based grammars [15] should provide sufficient background on the former; the latter is covered in natural-language processing textbooks [1; 18].

The implementation itself has been ported to various computers. Hardware environment should be one of the following:

- Symbolics 3600 LISP machine running Genera 7.2 or later.

- Sun workstation running Lucid Common LISP and Gnu EMACS.

[1] The preparation of this paper and the research reported herein were supported in part by a contract with the Nippon Telegraph and Telephone Corporation, and in part by a gift from the System Development Foundation.

- Apple Macintosh computer, preferably a Macintosh II, running Allegro's Coral Common LISP version 1.2 or later.

- IBM PC-RT computer running A/IX and Lucid Common LISP.

1.2 Functionality of CL-PATR

The CL-PATR system allows users to manipulate, test, debug, and modify grammars for fragments of natural language written in the S-PATR standard for the PATR-II formalism.

1.2.1 Grammar Manipulation

A grammar is written as a group of computer files containing grammatical and lexical information about an object language. These files are read and digested by the CL-PATR system in preparation for testing, debugging, and modifying the grammars.

To this end, CL-PATR provides for listing a directory of the various grammar files it can reference, reading a particular grammar file or set of files, installing a compiled version of the files, and listing the various grammar components such as rules and lexical entries.

1.2.2 Grammar Testing

A grammar can be tested by using the system to analyze object language sentences according to the grammatical information, and checking the results of the analysis. Alternatively, testing by generating sentences admitted by the grammar can be performed.

To this end, CL-PATR provides for parsing and generating of sentences.

1.2.3 Grammar Debugging

Aiding in the location of grammar errors are facilities for tracing the grammar rules, browsing the analysis and generation results, and manually performing analysis and generation steps.

To this end, CL-PATR provides for printing parse tree representations of a phrase, listing the contents of the chart, displaying the various results of an analysis, and manually extending one chart edge with another.

1.2.4 Grammar Modification

Modifying grammars is aided by a facility for making incremental changes to a grammar in order to update it without rereading the entire modified grammar.

To this end, CL-PATR provides for updating the grammar with new grammatical information.

1.2.5 User Interfaces

The basic user interface, designed for maximum portability, is a simple line-oriented command interface. The interface has been slightly augmented for the Macintosh computer. A separate graphical interface is supplied for Symbolics 3600 LISP machines.

1.3 Summary of Contents

This manual covers two essentially independent topics: the structure of the S-PATR formalism and the functionality of the CL-PATR implementation of that formalism.

We start by presenting the details of the S-PATR standard for the PATR-II grammar formalism (Chapter 2), followed by some sample grammar files (Chapter 3). Chapter 4 discusses the changes and augmentations to the S-PATR standard that CL-PATR allows for in grammar files.

The remainder of the manual is concerned with the implementation itself, rather than the formalism. Instructions for installing the system on various computers is given in Chapter 5. (Symbolics users may want to use the system files given in Chapter 13 when installing the system.)

An overview of the functionality of CL-PATR available through its various interfaces takes up Chapter 6. This chapter presents the basic structure of interaction with the system around which all the interfaces are constructed. The portable command interface is then described in Chapter 7, followed by a sample run of the system under this interface (Chapter 8). For users of Symbolics 3600 computers, the graphical interface for that machine is presented in Chapter 9, again followed by snapshots of a sample run (Chapter 10). Discussion of interface enhancements for the Macintosh and Sun computers occupies Chapter 11. Finally, the implementation methods used in the parsers and generators in CL-PATR are discussed in Chapter 12.

Readers familiar with the PATR formalism may want to skip immediately to Chapters 6 and 7 to get an idea of what using the system is like, returning to the earlier chapters for reference.

> **Advanced Topics Are Marked.** Throughout the manual, advanced topics are offset and marked with an identifying comment in bold face, as this paragraph itself is.

> **Typographical Conventions.** Italics are used both for introducing new terms and for emphasis. Material that is input to or output from a computer by keyboard—including file names, menu items, and similar mouse-sensitive elements—is printed in a typewriter font. All quote mark applications use double quotes, except that single quotes will be used when mentioning, as opposed to using, words or phrases. Because the topic of this document is the manipulation of text by formal grammars, the typographical convention of moving trailing punctuation within quote marks is not followed, as it blurs important textual distinctions.

Chapter 2

The S-PATR Grammar Formalism

Currently, there are several versions of programs interpreting languages derived from the PATR formalism. In addition to the CL-PATR system, they include (in roughly chronological order):

System	*Language*	*Hardware*	*Developer*
Z-PATR	ZETALISP	Symbolics 3600	Shieber
D-PATR	INTERLISP	Xerox Dandelion	Karttunen
mini-PATR	Prolog	SUN 2	Pereira, Shieber
P-PATR	Prolog	SUN 2	Hirsh
C-PATR	Common LISP	HP Bobcat	Jones, Moore, Shaio

Many of these systems use slightly different versions of the language and make incompatible assumptions concerning interpretation. A PATR standard would provide a benchmark against which systems could be checked, and would allow sharing of grammars and possibly implementation code. Furthermore, it would constitute a starting point from which extensions and changes can be designed, analyzed and debated.

This chapter constitutes a draft specification for a standard version of the PATR computer language. We have made the following general design choices:

- The syntax is designed so as to be easily convertible to an LL(1) grammar (by left factorization), so that simple parsing techniques, such as recursive descent, can be used to read a grammar.

- The syntax is based on ease of human, rather than computer, readability. For this reason, we eschew notations motivated by particulars of individual programming languages, e.g., LISP-like parenthesized notations.

- Certain notational conventions are taken from other programming languages, particularly Prolog, LISP and ML.

The standard PATR language described here will henceforth be referred to as S-PATR (pronounced "spatter"). A S-PATR *interpreter* or *compiler* is a program using S-PATR as its input language.

2.1 Background

The S-PATR language comprises a grammar formalism for codifying fragments of natural languages. To avoid confusion we will refer to the S-PATR language as the *metalanguage* and any language fragment being encoded in a S-PATR grammar system as an *object language*. The language in which a particular S-PATR interpreter or compiler is written will be referred to as the *native language*.

The reader is assumed to be familiar with previous work on the PATR formalism. A brief review of PATR—to remind the reader of certain terminology used in the discussion below—is presented here.

The PATR formalism describes object-language expressions by associating them with *feature structures* (also *directed graphs* or *DGs*). DGs come in two varieties, *atomic* DGs with no structure except for their *name*, and *compound* DGs which are structured as a set of pairs composed of a *feature* (drawn from a finite set of features) and another DG.

DGs are associated with expressions indirectly. Rather than giving the graph structure itself, we give a [partial] description of the graph structure in terms of mutual constraints among several DGs. For primitive expressions of the object language (i.e., words), the set of DGs among which the constraints are required to hold is comprised of only one DG, that DG associated with the word. For complex expressions (phrases), the set of DGs among which the constraints are required to hold includes the DGs associated with the whole phrase and all of its immediate subphrases.

In S-PATR, primitive expressions are assigned DGs with the Word statement and compound expressions are assigned DGs inductively with the Rule statement. Furthermore, commonly used sets of constraints can be defined as macros with the Macro and Stem statements. These statements provide the core of the ability to define object language fragments in S-PATR; other statements allow configuring certain parsing or grammar parameters and distributing grammatical information among other files.

For example, the following are statements in the S-PATR language taken with modification from the sample grammar in Chapter 3. We will refer to these examples in the discussion below.

```
;;; A sample rule
Rule 'sentence formation'

    S -> NP VP:

        Head(S, VP)                     ;VP is the head of S
        <S head> = [form: finite]       ;S must be finite
        <VP subcat> = (NP | ()).        ;VP must be saturated
                                        ;except for the subject

;;; A sample lexical entry
Word 'Crockett':
    ProperNoun.

;;; Two macro definitions used by the above examples
Macro ProperNoun:
    <cat> = 'NP'
    <head trans> = <word>.

Macro Head (Parent, HeadChild):
        <Parent head> = <HeadChild head>.
```

2.2 Token Structure and Lexical Issues

The unit of lexical analysis in S-PATR is the *token*. Because the S-PATR language has as its subject matter natural-language text, in which case distinctions can be significant, the tokens of S-PATR are case-sensitive. For instance, the tokens `US` and `us` are distinct.[1]

Tokens are used for a variety of purposes in S-PATR. We list them here, with some examples taken from the example statements above.

- Metalanguage keywords: `Rule`, `->`, `Macro`, etc.
- Atomic DGs: `finite`, `'NP'`
- Features: `head`, `form`, etc.
- Object language words and other strings: `'Crockett'`, `'sentence formation'`
- Macro names: `ProperNoun`, `Head`
- Names for referring to DGs: `S`, `NP`, `VP`, `Parent`, etc.

Unlike many programming languages, we do not for the most part distinguish these uses of tokens syntactically, but rather, by the context in which they appear. The only distinction made syntactically is between two classes of tokens—*identifiers* and *handles*—about which more will be said later. One important side effect of distinguishing the uses of tokens by context is that there are no reserved words in the S-PATR language; even keywords can be used in other ways. For instance, although the token `string` is a keyword in the metalanguage, we can also use the same token, in a different context, to represent the object-language word 'string'. This is important, since reserved words would contradict the ability of the metalanguage to talk about arbitrary fragments of natural language.

2.2.1 Character classes

Tokens are composed of alphanumeric and special characters. The following characters are considered alphabetic: `a-z`, `+` (plus sign), `A-Z`, `*` (asterisk). The first half of these characters are lower case (including plus sign) and the second half upper case (including asterisk).

The following characters are considered numeric: `0-9`.

The following characters are considered alphanumeric: all alphabetic and numeric characters plus `_` (underbar).

At least the following characters are considered special characters: `:` (colon), `;` (semicolon), `[ ]` (square brackets), `< >` (angle brackets), `()` (parentheses), `.` (period), `,` (comma), `=` (equals sign), and `-` (dash). The `~` (tilde) and `{ }` (braces) are reserved for extensions to allow negation and disjunction.

At least the following characters are considered whitespace: space, newline, tab, formfeed.

[1] This will require that implemented in Common LISP use a separate lexical analyzer from the Common LISP reader which always folds case.

2.2.2 Identifiers and handles

Tokens are divided syntactically into two classes: identifiers and handles. *Identifiers* are tokens composed of a sequence of non-whitespace characters whose first alphabetic character, if there is one, is not uppercase.[2] All other sequences of nonwhitespace characters, i.e., those in which there is a first alphabetic character which is uppercase, are *handles*. Intuitively, handles are used to refer to DGs, that is as names for, or handles on, DGs. Identifiers are used for all other purposes. Since S-PATR keywords are matched in a case-insensitive way, both identifiers and handles can be used for this purpose. Thus atoms are used in S-PATR for the following purposes:

- Metalanguage keywords
- Atomic DGs
- Features
- Words and strings

Handles are used for these purposes:

- Metalanguage keywords
- Macro names
- Names for complex DGs

Recall that the asterisk is an uppercase alphabetic and the plus sign a lowercase alphabetic. Thus tokens starting with the former are handles, the latter identifiers.

> **Pseudo-words and Plus Signs.** A *pseudo-word* is a word whose definition is not lexically associated with its spelling, but rather, with some class of words that cannot be easily encoded as separate lexical items (for instance, because the class is infinite). It is often useful to include such pseudo-words in the lexicon. The intended notational convention is that pseudo-words should be spelled with surrounding plus signs. For instance, the pseudo-word `+integer+` might be reserved for all integers in the input. Similarly, the pseudo-word `+unknown+` might be reserved for any otherwise unknown word. The set of pseudo-words recognized by an implementation remains implementation-dependent but the following are possibilities: `+integer+`, `+unknown+`, `+date+`, `+time+`.
>
> **Conventional Use of Asterisks.** The asterisk in handles is intended for conventional use in introduced handles and stem handles (discussed later).

Digits are considered full alphanumeric characters; thus, the sequence of characters `123abc` is one identifier, not two as under normal programming language definitions. This convention allows, for example, the use of `3sg` as an atomic DG specifier and `3Sg` as a macro name.

Special characters always form single character identifiers with the exception of the following two-character identifiers: `->` used as the arrow in the grammar rule syntax, and `()` used to mark the ends of lists.

[2] If there are no alphabetic characters, the token is therefore an identifier since the condition holds vacuously.

Identifiers that include special characters or whose first alphabetic character is uppercase can be specified by enclosing an arbitrary string in single quotes. For instance, `'*a=B#'` is a single identifier. Quoted identifiers may begin with an uppercase alphabetic as in the previous example. Whenever strings of arbitrary characters are needed in the language (e.g., as file names in the `Input` statement), identifiers are used because these quoted identifiers can include arbitrary text. Single quotes can be inserted in quoted identifiers by doubling them, e.g., `'don''t'`.

If a given handle includes one or more underbar characters, the handle composed of the sequence of characters before the first underbar is called the *category* of the given handle. If the given handle includes no underbar character, its category is the same as the entire handle. For instance, the category of the handle `VP_1` is the handle `VP`.

The *category name* of a handle is the identifier with the same spelling as the its category. Typically, such an identifier would have to be specified using the single quoting notation. For instance, the category name of the handle `VP_1` is the identifier which would be notated `'VP'`.

2.2.3 Comments

Comments are introduced by a semicolon (`;`) and continue to the end of the line. No nesting comment notation is provided. Comments are treated exactly like whitespace, and are allowed anywhere except in quoted identifiers.

> **Nesting Comment Notation.** An alternative to the standard: comments may be bracketed by the `#|` and `|#` comment delimiters. Nesting of these delimiters is allowed.

2.3 Specifying DGs and DG Constraints

Before discussing the syntax and semantics of S-PATR statements, we set out the notation for specifying DGs, and for specifying mutual constraints among several DGs.

2.3.1 Specifying DGs

We will use the following preterminal symbols in the Backus-Naur form (BNF) rules constituting the metagrammar of S-PATR:

- ⟨*identifier*⟩ for identifiers
- ⟨*handle*⟩ for handles

The metalanguage keywords appear as terminal symbols in the metagrammar. Note that these symbols match in a case-insensitive way. That is, although the token `Rule` is distinct from the token `RULE`, both may be used to specify the metalanguage keyword given as the terminal symbol `"Rule"` in the grammar.

The preterminals are introduced by the following rules:

⟨*dg-atom*⟩ `::=` ⟨*identifier*⟩

corresponding to the use of identifiers to specify atomic DGs,

```
⟨feature⟩ ::= ⟨identifier⟩
```

corresponding to the use of identifiers to specify features,

```
⟨macro0⟩ ::= ⟨handle⟩
⟨macro1⟩ ::= ⟨handle⟩
```

corresponding to the use of handles to specify macro names, and

```
⟨simple-dg-specifier⟩ ::= ⟨handle⟩
```

corresponding to the use of handles to specify DGs.

The final use of identifiers to specify words or other strings correspond to instances of the preterminal *⟨identifier⟩* elsewhere in the grammar.

Certain contexts define a set of *bound handles* and a notion of *default handle*. For instance, in the sample grammar rule above, the DG constraints are in an environment in which the bound handles are `S`, `NP` and `VP`. Bound handles are handles that have been associated with (bound to) some DG that is being constrained by the description. In certain environments, one of the bound handles may be designated as the default handle. For instance, in defining a macro, the name of the macro is the default handle.

The rule introducing an identifier as a feature can only be used if the identifier was declared as a feature in the `Features` profile statement. All other occurrences of identifiers are considered to be specifying atomic DGs. Similarly, handles used as macro names are identifiable because they must be defined in a `Macro` statement. All other handles are considered to be specifying complex DGs, either as bound handles or introduced handles (defined later).

The full syntax for DG specifiers is as follows:

```
⟨simple-dg-specifier⟩ ::= ⟨dg-atom⟩
⟨simple-dg-specifier⟩ ::= ⟨handle⟩
⟨simple-dg-specifier⟩ ::= ⟨path-spec⟩
⟨simple-dg-specifier⟩ ::= ⟨feature⟩ ":" ⟨simple-dg-specifier⟩
⟨simple-dg-specifier⟩ ::= ⟨macro0⟩
⟨simple-dg-specifier⟩ ::= ⟨macro1⟩ ⟨simple-dg-specifier⟩
⟨simple-dg-specifier⟩ ::= "[" ⟨simple-dg-list⟩ "]"
⟨simple-dg-specifier⟩ ::= "(" ⟨dg-specifier⟩ ")"
⟨path-spec⟩ ::= "<" ⟨simple-dg-specifier⟩ ⟨feature-list⟩ ">"
⟨path-spec⟩ ::= "<" ⟨feature-list⟩ ">"
⟨feature-list⟩ ::=
⟨feature-list⟩ ::= ⟨feature⟩ ⟨feature-list⟩
⟨simple-dg-list⟩ ::=
⟨simple-dg-list⟩ ::= ⟨simple-dg-specifier⟩ ⟨simple-dg-list⟩
⟨id-dg-specifier⟩ ::= ⟨simple-dg-specifier⟩ "=" ⟨id-dg-specifier⟩
⟨id-dg-specifier⟩ ::= ⟨simple-dg-specifier⟩
⟨dg-specifier⟩ ::= ⟨id-dg-specifier⟩
⟨dg-specifier⟩ ::= ⟨id-dg-specifier⟩ "," ⟨dg-specifier⟩
⟨dg-specifier⟩ ::= ⟨id-dg-specifier⟩ "|" ⟨dg-specifier⟩
⟨macro0⟩ ::= ⟨handle⟩
⟨macro1⟩ ::= ⟨handle⟩
⟨dg-atom⟩ ::= ⟨identifier⟩
⟨feature⟩ ::= ⟨identifier⟩
```

Names for DGs

A handle specifies a DG according to the particular environment in which it occurs.

⟨*simple-dg-specifier*⟩ ::= ⟨*handle*⟩

If the handle is a bound handle in the given environment, it specifies the DG with which it is associated by the construct in which it was bound.

If a handle is used which is not a bound handle in the given environment, it is implicitly bound to a new otherwise unconstrained DG. Such handles will be referred to as *introduced handles*, conventionally written with asterisks as the first and last characters in the handle. The scope for which the binding of the introduced handle is in force is the statement in which this occurrence of the handle occurs. An important issue concerns whether the binding of the introduced handle to a DG is in force within the DG being defined, that is, whether circularities are allowed. The contentious solution proposed here would impose no restriction of acyclicity.

Atomic DGs

An atomic DG can be specified by giving the name of the atomic DG as an identifier.

⟨*simple-dg-specifier*⟩ ::= ⟨*dg-atom*⟩
⟨*dg-atom*⟩ ::= ⟨*identifier*⟩

Paths

A DG can be specified by giving a path from a specifier to a sub-DG:

⟨*simple-dg-specifier*⟩ ::= ⟨*path-spec*⟩
⟨*path-spec*⟩ ::= "<" ⟨*simple-dg-specifier*⟩ ⟨*feature-list*⟩ ">"
⟨*feature-list*⟩ ::=
⟨*feature-list*⟩ ::= ⟨*feature*⟩ ⟨*feature-list*⟩
⟨*feature*⟩ ::= ⟨*identifier*⟩

This syntax specifies the DG reached by following the given features in the DG specified by the ⟨*simple-dg-specifier*⟩. The following convention allows a simpler structure in case the root of the path is specified by the default handle.

- **Default Handle Convention I:** If the DG specifier is the default handle, it can be removed.

⟨*path-spec*⟩ ::= "<" ⟨*feature-list*⟩ ">"

> **Detecting the Default Handle Convention.** Detecting when this convention is in force can be a bit tricky, since a ⟨*simple-dg-specifier*⟩ can itself begin with a feature. Thus the use of this convention cannot be keyed to the existence of a feature as the first token in a path. For instance, consider the path `<feat:val feat>` which should specify the same DG as `val`. Implementers might want to disallow certain of the ⟨*simple-dg-specifier*⟩ expansions as the first element of a path by redefining the grammar slightly, so as to make detection of this convention simpler.

Feature Values

A DG with a specific value for a feature can be specified by giving the feature and its designated value.

⟨*simple-dg-specifier*⟩ ::= ⟨*feature*⟩ ":" ⟨*simple-dg-specifier*⟩

For instance, the DG specified by `form: finite` has the atomic DG `finite` as the value of the feature form. This notation is intended primarily for use with the bracket notation for conjunction which is described next.

Conjunction

A DG can be listed explicitly using bracket notation to denote conjunction (unification) of several simple DG specifiers.

⟨*simple-dg-specifier*⟩ ::= "[" ⟨*simple-dg-list*⟩ "]"
⟨*simple-dg-list*⟩ ::=
⟨*simple-dg-list*⟩ ::= ⟨*simple-dg-specifier*⟩ ⟨*simple-dg-list*⟩

This notation for conjunction is quite general, but it is expected that it will be used primarily to explicitly specify some simple dgs, e.g,

```
[cat: 'V'
 agr: [person: third]]
```

Reentrancies

A DG can be specified with a side effect of constraining it to be identical to another DG. This is done with the following notation:

⟨*id-dg-specifier*⟩ ::= ⟨*simple-dg-specifier*⟩ "=" ⟨*id-dg-specifier*⟩
⟨*id-dg-specifier*⟩ ::= ⟨*simple-dg-specifier*⟩

This notation specifies the single DG specified by both DG specifiers. The side effect of identifying the specified DGs will guarantee that the same token DG is specified.

Identifications of this sort in conjunction with introduced handles can be used to specify a DG with a side effect of giving it a name for later reference. DGs so named can be referred to within the scope of their binding thereby giving a way to introduce reentrancy in DG specifiers. For example,

```
[a: [b: *1*=[c: s]]
 d: *1*]
```

notates a DG where the `b` and `d` features are token-identical. Note that by the scope definitions given for introduced roots, definition before use is not required. For instance, the same DG type can be specified with the notation

```
[d: *1*
 a: [b: *1*=[c: s]]]
```

Indeed, even this unexpected notation can be used to specify the same DG.

```
[a: *2*
 d: <*2* b>=[c: s]]
```

Lists

Lists can be encoded with DGs with the well-known technique of using two features `first` and `rest` to recursively connect list elements. For instance, the list containing elements `a`, `b`, and `c` can be encoded by:

```
[first: a
 rest:  [first: b
         rest:  [first: c
                 rest: ()]]]
```

where `()` is a special token used for marking the end of a list.

This encoding is so common that a special notation is introduced for it. The infix vertical bar notation (similar to infix dot in LISP)

```
x|y
```

is an alternate notation for

```
[first: x
 rest:  y]
```

and the infix comma notation

```
x,y
```

is an alternate notation for lists ending with `()`, e.g.,

```
[first: x
 rest: [first: y
        rest: ()]]
```

⟨*dg-specifier*⟩ ::= ⟨*id-dg-specifier*⟩
⟨*dg-specifier*⟩ ::= ⟨*id-dg-specifier*⟩ "," ⟨*dg-specifier*⟩
⟨*dg-specifier*⟩ ::= ⟨*id-dg-specifier*⟩ "|" ⟨*dg-specifier*⟩

Since elements of lists are ⟨*id-dg-specifier*⟩s, comma and bar have lower precedence than other connectives. Therefore, it will usually be necessary to parenthesize lists to get the proper bracketings.

In the sample rules, lists were used in several places. For instance, in the constraint `<VP syncat> = (NP | ())`, the verb phrase's syncat list was constrained to be identical to a list of length one whose first element is the DG specified by the handle `NP` (which will be the DG associated with the subject noun phrase).

Parenthesization

Because the precedences of the various operators used to specify DGs are sometimes inappropriate for a particular application, parentheses are allowed to override these precedences. Parenthesizing an expression specifying a DG does not change the DG specified.

```
⟨simple-dg-specifier⟩ ::= "(" ⟨dg-specifier⟩ ")"
```

Parenthesization was used in the sample statements in the macro application `Head(S, VP)` to override the default parenthesization `(Head S), VP`.

Macro applications

Application of a macro yields a DG. In general, macros are applied to zero or one DG specifiers, an invocation being notated with prefix notation.

```
⟨simple-dg-specifier⟩ ::= ⟨macro0⟩
⟨simple-dg-specifier⟩ ::= ⟨macro1⟩ ⟨simple-dg-specifier⟩
⟨macro0⟩ ::= ⟨handle⟩
⟨macro1⟩ ::= ⟨handle⟩
```

For instance, the following macro applications occurred in the sample statements above:

```
Head(S, VP)
ProperNoun
```

Note that in the first example, the macro `Head` is applied to a single argument which is a list, not two separate arguments.

The semantics for macro application is given in the section on `Macro` statements.

2.3.2 Specifying DG Constraints

When several given DGs are to be mutually constrained, the constraints are stated as a set of identities. The allowed identity format is as follows:

```
⟨dg-constraints⟩ ::=
⟨dg-constraints⟩ ::= ⟨dg-constraint⟩ ⟨dg-constraints⟩
⟨dg-constraint⟩ ::= ⟨simple-dg-specifier⟩ "=" ⟨dg-specifier⟩
```

For example, the sample statements above use the following constraints.

```
<S head> = [form: finite]
<VP syncat> = (NP | ())
<cat> = 'NP'
<head trans> = <word>
<Parent head> = <HeadChild head>
```

In addition, the following convention allows a simpler notation for constraining the default handle.

- **Default Handle Convention II:** If the left-hand side of the = is the default handle, the left-hand side and the = may be dropped.

  This convention allows for compatibility with previous versions where template names could be inserted by themselves in template definitions to mean that the template was to be unified directly as a part of the template being defined.

  ⟨*dg-constraint*⟩ ::= ⟨*simplc-dg-specifier*⟩

  Detecting the Default Handle Convention. Use of this convention can be detected by noting that the convention is being used whenever the lookahead token after the ⟨*simple-dg-specifier*⟩ that begins the ⟨*dg-constraint*⟩ is not an =. Note that this assumes that = is not being used as a name of a macro, for instance. If it is being so used, the macro invocation can be surrounded with parentheses.

For example, in the sample rules, the following constraints used this convention.

```
Head(S, VP)         (rather than S = Head(S, VP))
ProperNoun          (rather than * = ProperNoun)
```

2.3.3 Summary of Precedences

All operators are right associative. The precedence of operators, as defined by the BNF rules above, is summarized below (from highest to lowest):

1. `(...)  [...]  <...>`
2. ⟨*macro application*⟩
3. = (of DG specifiers)
4. , |
5. = (of DG constraints)

2.4 Structure of S-PATR Statements

2.4.1 General notes

The largest unit of expression in S-PATR is a *grammar system*. A grammar system is a set of files that together describe the grammar and lexicon of a single object language. Each file in the grammar system (a *grammar file*) contains zero or more S-PATR *statements*.

⟨*stmt-list*⟩ ::=
⟨*stmt-list*⟩ ::= ⟨*stmt-list*⟩ ⟨*stmt*⟩

It is useful to note that all statements are delimited at the end by a period. This fact can be used, for instance, to aid in error recovery during file parsing.

Five basic statement types are allowed.

⟨*stmt*⟩ ::= ⟨*grammar-rule*⟩
⟨*stmt*⟩ ::= ⟨*macro-defn*⟩
⟨*stmt*⟩ ::= ⟨*lexical-stmt*⟩
⟨*stmt*⟩ ::= ⟨*control-stmt*⟩
⟨*stmt*⟩ ::= ⟨*profile-stmt*⟩

2.4.2 Grammar Rules

Grammar rules are specified with the following statement:

⟨*grammar-rule*⟩ ::= "Rule" ⟨*identifier*⟩ ⟨*lhs*⟩ "->" ⟨*rhs*⟩ ":" ⟨*dg-constraints*⟩ "."
⟨*lhs*⟩ ::= ⟨*handle*⟩
⟨*rhs*⟩ ::= ⟨*handle-list*⟩

where

- ⟨*identifier*⟩ is a mnemonic to uniquely identify the rule. The implementation may make use of the uniqueness of identifiers, e.g. by indexing the rules by the identifier, or using the identifier as a short way of representing the rule to the user.
- ⟨*lhs*⟩ is a handle representing the parent constituent.
- ⟨*rhs*⟩ is a list of zero or more handles representing the child constituents in order. Since zero roots are allowed, epsilon rules can be stated directly.
- ⟨*dg-constraints*⟩ specifies the constraints that the rule invocation places on the left and right side handles. That is, the constraints are evaluated in an environment in which the left and right side roots are bound handles (associated with the DGs of the parent constituent and the child constituents respectively) and the left side handle is the default.

The interpretation of a grammar rule is as follows: A grammar rule R admits a sequence of pairings of strings s_i and DGs D_i for $0 \leq i \leq n$ if and only if:

- The right side of R has length n.
- $s_0 = s_1 \cdots s_n$.
- The rule constraints (augmented according to the conventions below) hold in an environment in which the left side handle is bound to D_0 and the right side handles are bound to D_1 through D_n in order and the left side handle is the default handle.

Various additional implicit constraints are imposed in addition to the explicit constraints listed in the grammar rule:

- **Category Convention:** For each left or right side handle H in the rule, if H has category C and C is a category (as specified in the `Categories` profile statement q.v.) then the following constraint is conventionally added to the DG constraints:

  ```
  < H p > = c
  ```

  where c is the category name of H and p is the category path (as defined in the `Category path` profile statement, q.v.)

- **Gap-Threading Convention:** To be decided. Possibly implemented as an external macro.

2.4.3 Lexical Structure Statements

The following discussion assumes a system with no morphological analyzer. The lexicon is specified through two types of statements, one for specifying stems of words, and one for the inflected forms.

Word definition

⟨*lexical-stmt*⟩ `::= "Word"` ⟨*identifier*⟩ `":"` ⟨*dg-constraints*⟩ `"."`

The interpretation of `Word` statements is as follows: A word w is associated with a DG D if and only if there is a `Word` statement associating w with a set of DG constraints and the constraints (augmented according to the conventions below) are satisfied in an environment in which the handle * is a handle bound to D and * is also the default handle.

This statement actually sets up the lexical pairing between the token and the specified DG. No macro is defined as in `Stem` statements, so other word definitions cannot use the DG specified here. Also the DG constraints are augmented according to the following convention.

- **Word Default Convention:** For a `Word` statement defining a word w, the implicit constraint

  ```
  < * word > = w
  ```

  is conventionally added.

Stem definition

Stems of words are specified with the `Stem` statement.

⟨*lexical-stmt*⟩ `::= "Stem"` ⟨*handle*⟩ `":"` ⟨*dg-constraints*⟩ `"."`

Actually, this statement does not associate a string with a DG (as the `Word` statement does) but merely provides a convenient way of defining a set of constraints that play a role in several words, and naming these constraints with a handle. As such it is merely syntactic sugar for a zero-argument macro definition.

A `Stem` statement with handle H and constraints D defines a macro of no arguments equivalent to the definition

Macro H: D'.

where D' is D augmented according to the following convention:

- **Stem Default Convention:** The implicit constraint

  < H stem > = c

  is conventionally added to D where c is the category name of H.

2.4.4 Macro Definitions

Macro statements are a generalization of both template and lexical rule definitions in earlier versions of PATR. Macros are defined with the following syntax:

⟨*macro-defn*⟩ ::= "Macro" ⟨*handle*⟩ ":" ⟨*dg-constraints*⟩ "."
⟨*macro-defn*⟩ ::= "Macro" ⟨*handle*⟩ ⟨*simple-dg-specifier*⟩ ":" ⟨*dg-constraints*⟩ "."

Interpretation of a macro invocation is as follows.

When a macro named M with no arguments and constraints E is invoked, it specifies a DG D such that the constraints E hold in an environment in which the handle M is bound to D and M is the default handle.

When a macro named M with one argument specified by *FArg* and constraints E is invoked by applying to a DG *AArg*, the invocation specifies a DG D such that the constraints E' hold in an environment in which the handle M is bound to D and M is the default handle and E' is E augmented with the constraint *FArg* = *AArg*.

This semantics allows a destructuring macro invocation style, because introduced roots in *FArg* have scope in E. This circumvents the need for multiple argument macros. For instance, a "two-argument" macro Head can be defined and invoked as in the sample rules at the beginning of this document.

> **Invoking a Macro Yields a New Token.** Note that the DG specified by two different invocations of a macro applied to the same argument will be type-identical to each other but token-distinct in general. For instance, consider the following constraints in an environment with a handle H and macro Foo.
>
> ```
> <H a> = Foo
> <H b> = Foo
> ```
>
> Although the two occurrences of H specify the same (token-identical) DG, the two references to the macro Foo specify token-distinct DGs. Thus, the following equality is not a consequence:
>
> ```
> <H a> = <H b>
> ```
>
> Although notationally this may seem confusing, it makes sense when considering the role of macros as abbreviatory devices, rather than as function specifiers. Consider

> the problems of implementing macros such that when called with type- or token-identical arguments, the token-identical result would be engendered.

Multiple definition of a macro name is disallowed, including defining zero- and one-argument macros with the same name.

> **Extending the Standard with Nondeterministic Macros.** An extension to the standard is the following: If the same macro name is defined with more than one macro definition, an invocation using that macro name chooses one of the definitions *nondeterminisitically*. This extension has far-reaching implications for the power, use, and implementation of and should be considered carefully before implementation. It basically makes macro definitions by themselves a Prolog-like language with a breadth-first solution strategy.
>
> As an example of the generality of macro definitions under this extension, the following definition could be used to nondeterministically propagate a slash feature to zero or one child constituents in a rule as in early GPSG.

```
Macro Slash_propagate (Parent, ()): .

Macro Slash_propagate (Parent, (Kid|Others)):
    Slash_propagate (Parent, Others).

Macro Slash_propagate (Parent, (Kid|Others)):
    <Parent slash> = <Kid slash>.
```

> It would be invoked as follows:

```
Rule 'ditransitive verb phrase'
    VP -> V NP PP:
        Slash_propagate (VP, (NP, PP)).
```

> **External Macros.** Macros provide a convenient hook to programs written in native code. A native code macro might be declared with an appropriate statement in a extension, such as
>
> ⟨*profile-stmt*⟩ ::= "External" "macro" ⟨*handle*⟩ "."
> ⟨*profile-stmt*⟩ ::= "External" "macro" ⟨*handle*⟩ ⟨*simple-dg-specifier*⟩ "."
>
> A file loaded with an `Evaluate` statement would provide the implementation of the macro as a function of the same number of arguments and returning a DG.

2.4.5 Control Statements

The S-PATR language includes two statements for controlling the process of interpreting grammar files.

Input of files

The `Input` control statement allows a grammar system to be split among several files. It specifies a file by giving its pathname as an identifier (probably a quoted identifier):

⟨*control-stmt*⟩ `::=` `"Input"` ⟨*identifier*⟩ `"."`

The interpretation of this statement is the same as the interpretation of the file specified by the pathname, that is, it works like `#include` in C. However, the file loaded must be parsable as a ⟨*stmt-list*⟩, that is, the grammar system may be split among several files only at statement boundaries.

Evaluation

The `Evaluate` statement allows the grammar to specify particular native language statements to run at grammar system installation time.

⟨*control-stmt*⟩ `::=` `"Evaluate"` ⟨*identifier*⟩ `"."`

The identifier is evaluated as a native language statement. This facility might be used, for instance, to load a program that calls the parser and run the program automatically. It might also be used (sparingly) to redefine certain functions of the implementation or to load code defining external macros.

2.4.6 Profile statements

Profile statements are used to tell the S-PATR interpreter the settings for various grammar-dependent parameters. All profile statements must precede any non-profile statements. The syntax of profile statements is typically the following: a keyword (or keywords) giving the parameter be set, followed by a colon, followed by the setting of the parameter followed by the delimiting period.

At least the following four profile statements must be accepted by the interpreter, but an implementation may choose to support others. Note that these statements are required in the sense that implementations must allow them, not that grammars must include them.

Features

The `Features` profile statement declares the set of identifiers that are to be interpreted as permissible features.

⟨*profile-stmt*⟩ `::=` `"Features"` `":"` ⟨*identifier-list*⟩ `"."`

This information is useful for error-checking and for the interpretation of the Default Handle Convention I.

In addition, it is recommended that when printing DGs, the printing routines attempt to respect the order in which the features are listed here, i.e., by sorting in the given order.

Because the features `word` and `stem` play a role in every grammar (as part of the Word Default Convention and the Stem Default Convention) these two features need not be declared in the `Features` statement of the grammar.

Start DG

The `Start` profile statement is used to specify the DG which serves as the "start symbol" of the grammar. The format is as follows:

> ⟨*profile-stmt*⟩ `::= "Start" ":"` ⟨*dg-constraints*⟩ `"."`

The DG constraints are in an environment where the only bound handle is the handle * which is bound to the start DG being defined. It is also the default handle. If this statement is left out, the start DG is assumed to be the left-hand side DG of the textually first grammar rule in the grammar system.

The following is a sample start DG definition stating that the start DG includes only finite sentences.

```
Start: <cat> = 'S'
       <head form> = finite.
```

Note the use of Default Handle Convention I.

Categories

The `Categories` profile statement is used to define the domain of applicability of the Category Convention. The format is as follows:

> ⟨*profile-stmt*⟩ `::= "Categories" ":"` ⟨*handle-list*⟩ `"."`
> ⟨*handle-list*⟩ `::=`
> ⟨*handle-list*⟩ `::=` ⟨*handle*⟩ ⟨*handle-list*⟩

The listed handles are considered to be categories for purposes of the Category Convention.

Category path

The `Category path` profile statement specifies where category information is put during interpretation by the Category Convention. The format is as follows:

> ⟨*profile-stmt*⟩ `::= "Category" "path" ":"` ⟨*path-spec*⟩ `"."`

The following four profile statements are likely to be needed, but are not required to be accepted by an interpreter. They are included here so that implementations that do include this type of information can do so compatibly.

Restrictor

The `Restrictor` profile statement allows the grammar system to tell an interpreter that uses restriction [14] what particular restrictor should be used in interpretation.

> ⟨*profile-stmt*⟩ `::= "Restrictor" ":"` ⟨*path-list*⟩ `"."`

Abbreviations

The following two profile parameters provide a conventional way to specify how category-valued dgs can be displayed in an abbreviated format.

⟨*profile-stmt*⟩ `::= "Abbreviation" "control" "string" ":"` ⟨*identifier*⟩ `"."`
⟨*profile-stmt*⟩ `::= "Abbreviation" ":"` ⟨*path-list*⟩ `"."`

The interpreter is free to use the information in the following way: a list of the atomic DG values at the end of each of the specified paths in the dg (as native language atoms) is built (if the value at the end of the path is non-atomic, nil is substituted); then, the control string is used in a call to the native language function `format` with the atom list as arguments.[3] For instance, the following might be a useful setting of these parameters:

```
Abbreviation control string:
              '~:[X~;~:*~a~][~@[~a~]~@[~a~]]'.
Abbreviation: <cat>
              <agreement number>
              <agreement person>.
```

This would result in the following abbreviations:

```
[cat: 'VP'
 agreement: [person: 3        ==>    VP[3sg]
             number: sg]]

[agreement: [person: 3        ==>    X[3sg]
             number: sg]]

[cat: 'NP'                    ==>    NP[sg]
 agreement: [number: sg]]
```

Semantics path

The `Semantics path` profile statement specifies where the semantic translation or logical form is built in the course of parsing, if the grammar does build such structures. The format is as follows:

⟨*profile-stmt*⟩ `::= "Semantics" "path" ":"` ⟨*path-spec*⟩ `"."`

This may be useful for using the interpreter as a front-end to a system interested in extracting the semantics of a phrase for further processing.

2.5 Extensions

Disjunction Braces are reserved for an extension to notate disjunction in a manner parallel to the use of square brackets for conjunction.

[3] ZETALISP, Common LISP and QUINTUS Prolog all provide a roughly compatible `format` facility.

⟨*simple-dg-specifier*⟩ ::= "{" ⟨*simple-dg-list*⟩ "}"

Negation The tilde notation is reserved for an extension to notate negation.

⟨*simple-dg-specifier*⟩ ::= "~" ⟨*simple-dg-specifier*⟩

Other possible extensions The following are further possible extensions.

- Profile statements for declaring whether or not to enforce conventions.
- Splice paths as in Karttunen.
- Regular paths as in Kaplan and Zaenen.
- Atomic inheritance as in LOGIN.
- Overwriting or default inheritance.
- Typing of DGs allowing only certain features in the domain of the DG.

Chapter 3

Sample Grammar Files

3.1 The Main File

```
;;;==================================================================
;;;                          Sample Grammar
;;;
;;; Compatible with the CL-PATR Common LISP Implementation of S-PATR
;;;==================================================================


Evaluate '(format t "~2%Loading CL-PATR Sample Grammar~2%")'.

Features: cat lex sense head
          subcat first rest
          form agreement person number gender
          trans pred arg1 arg2 arg3.

Categories: S NP VP V.

Category path: <cat>.

Start: <cat> = 'S'
       <head form> = finite.

Restrictor: <cat>
            <head form>.

Semantics path: <head trans>.

Abbreviation:
        <cat>
        <head agreement number>
        <head agreement person>.

Abbreviation control string: '~:[X~;~:*~a~]~@[[~a~@[~a~]]~]'.

Input 'sample.gram'.
Input 'sample.lex'.
```

3.2 The Grammar File

```
;;;=================================================================
;;;                           Sample Grammar
;;;=================================================================


Macro Head(Parent, Child):
        <Parent head> = <Child head>.


;;;=================================================================
;;;                           Grammar Rules
;;;=================================================================


Rule 'sentence formation'

        S -> NP VP:

                Head(S, VP)                     ;VP is the head of S
                <S head> = [form: finite]       ;S must be finite
                <VP subcat> = (NP | ()).        ;VP must be saturated
                                                ;except for the subject


Rule 'trivial verb phrase'

        VP -> V:

                Head(VP, V)                      ;V is the head of VP

                <VP subcat> = <V subcat>.        ;same subcategorization
```

```
Rule complements

        VP_parent -> VP_child X:

                Head(VP_parent, VP_child)        ;smaller VP is the head

                ;; Decompose subcategorization of the head into a
                ;; subject, a first postverbal complement and the rest
                ;; of the complements.  Note that we've found the first
                ;; postverbal complement (X).
                <VP_child subcat> = (*Subject*, X | *OtherComplements*)

                ;; We're still looking for the subject and the rest of
                ;; the complements.
                <VP_parent subcat> = (*Subject* | *OtherComplements*).

Rule 'adverbial modification'

        VP_parent -> VP_child AdvP:

                Head(VP_parent, AdvP)            ; Adverbial is the head

                ;; parent VP needs all the complements that the child does
                <VP_parent subcat> = <VP_child subcat>
                ;; child VP must be saturated except for subject
                <VP_child subcat> = ( *Subj* | () )
                ;; parent VP is modified
                <VP_parent modified> = yes
                ;; adverbial gets to look at child to form semantics
                <AdvP subcat> = VP_child.
```

3.3 The Lexicon File

```
;;;==================================================================
;;;                          Sample Grammar
;;;==================================================================


;;;==================================================================
;;;                             Lexicon
;;;==================================================================

;;;------------------------------------------------------------------
;;;                           Noun Phrases
;;;------------------------------------------------------------------
```

```
Macro NounP:
        <cat> = 'NP'
        <head trans> = <word>.

;;;                              Gender

Macro Masc:        <head agreement gender> = masculine.
Macro Fem:         <head agreement gender> = feminine.
Macro Neut:        <head agreement gender> = neuter.

;;;                              Number

Macro Sing:        <head agreement number> = singular.
Macro Plur:        <head agreement number> = plural.

;;;                              Person

Macro Person P:    <head agreement person> = P.

;;;                         Other Agreement

Macro 3Sing:       Sing
                   Person 3
                   <head agreement 3sg> = yes.

Macro Non3Sing:    <head agreement 3sg> = no.

;;;                        Subclasses of NPs

Macro ProperNoun:  NounP 3Sing.

Word 'Crockett':   Masc ProperNoun.
Word 'Sonny':      Masc ProperNoun.
Word 'Tubbs':      Masc ProperNoun.
Word 'Ricardo':    Masc ProperNoun.
Word 'Rico':       Masc ProperNoun.
Word 'Castillo':   Masc ProperNoun.
Word 'Gina':       Fem ProperNoun.
Word 'Kait':       Fem ProperNoun.
Word detectives:   Plur NounP.
Word criminals:    Plur NounP.


;;;------------------------------------------------------------------
;;;                            Verb Phrases
;;;------------------------------------------------------------------
```

```
Macro Verb:        <cat> = 'V'.

Macro MainVerb:    Verb
                   <head aux> = false
                   <head trans pred> = <stem>.

;;;                            Verb Forms

Macro Finite:      <head form> = finite.

Macro Nonfinite:   <head form> = nonfinite.

Macro Infinitival: <head form> = infinitival.

Macro PassiveParticiple:
                   <head form> = passiveParticiple.

;;;                            Agreement

Macro Agrees AgrFeatures:
                   ;; only finite verbs show agreement
                   Finite
                   <subcat first> = AgrFeatures.


;;;                     Subcategorization Frames

Macro Intransitive:
        MainVerb
        <subcat> = ( [cat: 'NP'
                      head: [trans: *SubjectMeaning*]]
                   | () )

        <head trans arg1> = *SubjectMeaning*.

Macro Transitive:
        MainVerb
        <subcat> = ( [cat: 'NP'
                      head: [trans: *SubjectMeaning*]]
                   , [cat: 'NP'
                      head: [trans: *ObjectMeaning*]] )

        <head trans arg1> = *SubjectMeaning*
        <head trans arg2> = *ObjectMeaning*.

Macro TakesS:
        MainVerb
        <subcat> = ( [cat: 'NP'
                      head: [trans: *SubjectMeaning*]]
                   , [cat: 'S'
                      head: [form: finite
                             trans: *SMeaning*]] )

        <head trans> = [arg1: *SubjectMeaning*
                        arg2: *SMeaning*].

Stem *say*:        TakesS.
Word says:         Agrees 3Sing *say*.
Word say:          Agrees Non3Sing *say*.
Word say:          Nonfinite *say*.
Word said:         Finite *say*.

Macro Raising:
        Verb
        <head trans> = [pred: <stem>
                        arg1: *PredMeaning*]
        <subcat> = ( *Subject* =
                     [cat: 'NP']
                   , [Infinitival
                      cat: 'VP'
                      head: [trans: *PredMeaning*]
                      modified: no
                      subcat: (*Subject* | () )]
                   ).

Macro Equi:
        Verb
        <head trans> = [pred: <stem>
                        arg1: *SubjectMeaning*
                        arg2: *PredMeaning*]
        <subcat> = ( *Subject* =
                     [cat: 'NP'
                      head: [trans: *SubjectMeaning*]]
                   , [Infinitival
                      cat: 'VP'
                      head: [trans: *PredMeaning*]
                      modified: no
                      subcat: (*Subject* | () )]
                   ).

Macro AuxVerb EmbeddedConstituent:
        Verb
        <head trans> = *PredMeaning*
        <subcat> = ( *Subject* =
                     [cat: 'NP']
                   , EmbeddedConstituent =
                     [cat: 'VP'
                      head: [trans: *PredMeaning*]
                      modified: no
                      subcat: (*Subject* | () )]
                   ).

Macro ReplaceForm(Input, Form):
        <cat> = <Input cat>
        <subcat> = <Input subcat>
        <modified> = <Input modified>
        <head form> = Form
        <head trans> = <Input head trans>
        <head agreement> = <Input head agreement>.

        Macro AgentlessPassive Active:
        ReplaceForm([cat: <Active cat>
                     subcat: <Active subcat rest>
                     head: <Active head>],
                    passiveParticiple).

Stem *fume*:      Intransitive.
Word fumes:       Agrees 3Sing *fume*.
Word fume:        Agrees Non3Sing *fume*.
Word fume:        Nonfinite *fume*.
Word fumed:       Finite *fume*.

Stem *ventilate*: Transitive.
Word ventilates:  Agrees 3Sing *ventilate*.
Word ventilate:   Agrees Non3Sing *ventilate*.
Word ventilate:   Nonfinite *ventilate*.
Word ventilated:  AgentlessPassive *ventilate*.
Word ventilated:  Finite *ventilate*.

Stem *kill*:      Transitive.
Word kills:       Agrees 3Sing *kill*.
Word kill:        Agrees Non3Sing *kill*.
Word kill:        Nonfinite *kill*.
Word killed:      AgentlessPassive *kill*.
Word killed:      Finite *kill*.

Stem *shoot*:     Transitive.
Word shoots:      Agrees 3Sing *shoot*.
Word shoot:       Agrees Non3Sing *shoot*.
Word shoot:       Nonfinite *shoot*.
Word shot:        AgentlessPassive *shoot*.
Word shot:        Finite *shoot*.

Stem *love*:      Transitive.
Word loves:       Agrees 3Sing *love*.
Word love:        Agrees Non3Sing *love*.
Word love:        Nonfinite *love*.
Word loved:       AgentlessPassive *love*.
Word loved:       Finite *love*.

Word is:          AuxVerb PassiveParticiple
                  Agrees 3Sing.
Word was:         AuxVerb PassiveParticiple
                  Agrees 3Sing.
Word were:        AuxVerb PassiveParticiple
                  Agrees Plur.

Word to:          AuxVerb Nonfinite
                  Infinitival.

Stem *seem*:      Raising Verb.
Word seems:       Agrees 3Sing *seem*.
Word seem:        Agrees Non3Sing *seem*.
Word seem:        Nonfinite *seem*.
Word seemed:      Finite *seem*.

Stem *want*:      Equi Verb.
Word wants:       Agrees 3Sing *want*.
Word want:        Agrees Non3Sing *want*.
Word want:        Nonfinite *want*.
Word wanted:      Finite *want*.

;;;------------------------------------------------------------------
;;;                               Adverbials
;;;------------------------------------------------------------------

Macro Adverb:     <cat> = 'AdvP'
                  <head form> = <subcat head form>
                  <head trans> = [pred: <word>
                                  arg1: <subcat head trans>].

Word passionately: Adverb.
Word quickly:      Adverb.
Word happily:      Adverb.
Word yesterday:    Adverb.
```

Chapter 4

The CL-PATR Version of S-PATR

CL-PATR implements an interpreter for the S-PATR grammar formalism described in Chapter 2. This chapter describes how CL-PATR makes implementation-dependent decisions and extensions to the specification in Chapter 2.

Pseudo-Words Implemented (Refer Section 2.2.2). The following pseudo-words are implemented: +unknown+ for unknown words and +number+ for integers in the input.

Disambiguating Identifiers (Refer Section 2.2.2). In parsing a grammar file, an attempt is made to disambiguate identifiers as to whether they are specifying an atomic DG, feature, or keyword. The following method is used:

1. If context disambiguates, the use dictated by context is assumed. If this dictates treating the identifier as a feature name, and the feature name was not declared in a Features statement (Section 2.4.6), the identifier is implicitly declared as a feature and the user is so notified.
2. Otherwise, the identifier is treated as the first one of the following which context allows: keyword, feature, atomic DG. To be treated as a keyword, the identifier must, of course, match the spelling of a contextually allowed keyword. To be treated as a feature, the macro need not be declared as a feature; it will be implicitly so declared as above. Any identifier can be treated as specifying an atomic DG.

Similarly, handles are disambiguated by a hierarchy as well. A handle is treated as the first one of the following which context allows: keyword, macro name, bound or introduced handle.

Warning: because of this disambiguation strategy, if you use a macro before defining it, no error will be reported. Rather, the intended macro name will be treated as an introduced handle.

Known Bug in the Grammar Reader. This disambiguation strategy leads to a bizarre scoping inconsistency. Since identifiers are treated as macros preferentially to bound handles, a macro definition that occurs before a statement with the same name as a bound handle, or that was installed before the incremental compilation of such a statement, makes the bound handle impossible to reference. Consequently, it is strongly advised that macro names not be used that occur elsewhere (even textually earlier) as handles. The fix to this misfeature is to make bound handles have higher precedence than macros.

No Nesting Comments (Refer Section 2.2.3). No nesting comment notation is implemented.

Default Handle Convention (Refer Section 2.3.1). The Default Handle Convention I is detected in full generality. No restriction is placed on the first elements of paths.

Gap Threading Unimplemented (Refer Section 2.4.2). The Gap-Threading Convention has not yet been implemented.

Multiple Macro Definitions Disallowed (Refer Section 2.4.4). Multiple definitions of macros are disallowed, although files with multiple definitions can be installed. The later definition will be in force, and a one-argument definition will take priority over a zero-argument definition. Users should not, however, rely on this behavior.

External Macros Unimplemented (Refer Section 2.4.4). External macros have not been implemented.

Profile Statements Augmented (Refer Section 2.4.6). All profile statements listed are implemented. The `Restrictor` statement is used by all parsers and generators based on the chart system architecture. The `Semantics` path is used by the bottom-up generator for installing the logical form. The `Abbreviation` statements are used in general for printing DGs in abbreviated form, as in printing edges.

An additional profile statement has been added to allow for the normalization of spelling of word and stem names, since they are frequently used as portions of the logical forms for phrases. The structure of the statement is

⟨*profile-stmt*⟩ `::=` `"Normalization"` `":"` ⟨*identifier-list*⟩ `"."`

where the identifiers in ⟨*identifier-list*⟩ are zero or more of `strip`, `lowercase`, or `uppercase`. If `strip` is included, special characters, i.e., underscore and asterisk are removed from the ends of the identifiers before using them as the atom values for the `word` or `stem` features as per the Word Default Convention and Stem Default Convention. If `lowercase` or `uppercase` are included, then the identifiers are forced to lowercase or uppercase, respectively.

In addition, the same normalization is applied to the atoms in S-expressions being generated from. The importance of the profile parameter, then, is to insure compatibility of the S-expression logical forms and lexical information.

The default setting for the parameter applies no normalization to atoms for compatability with the standard of Chapter 2. In general a setting of `strip` and `lowercase` is more appropriate, however.

Two additional profile statements allow the specification of different restrictors to be used for parsing and generation. If either of these are specified, they take precedence over a restrictor defined by a `Restrictor` statement for the specific process they apply to. The syntax is

```
⟨profile-stmt⟩ ::= "Generation" "restrictor" ":" ⟨path-list⟩ "."
⟨profile-stmt⟩ ::= "Parsing" "restrictor" ":" ⟨path-list⟩ "."
```

Extensions Unimplemented (Refer Section 2.5). None of the listed extensions have been implemented.

Chapter 5

Installing and Running CL-PATR

The CL-PATR system is distributed as a set of binary files, plus sources for a subset of the files. The files are organized under the following directory structure:

CL-PATR directory: Source and object code for CL-PATR, plus two subdirectories. The sources distributed include the following files:

`cl-patr-3600.lisp`	the main installation file for Symbolics 3600s
`cl-patr.lisp`	the main installation file for other computers
`metaparse.lisp`	metaparser for reading S-PATR files
`sh-red-p.lisp`	a particular parser built on the chart system architecture simulating a shift-reduce parser
`earley-p.lisp`	a particular parser built on the chart system architecture simulating an Earley parser
`topdown-g.lisp`	top-down backtracking generator
`earley-g.lisp`	bottom-up chart-based generator
`bridge.lisp`	converts S-expr LFs to DG encoding
`interface.lisp`	portable read-eval-print loop interface
`top-level.lisp`	user interface based on labeling REP loop
`mac-iface.lisp`	Macintosh-specific interface code
`gnu-iface.el`	GNU Emacs interface commands
`3600-iface.lisp`	Symbolics-3600-specific interface code

The subdirectories are:

`documentation` directory: Documentation of CL-PATR, in particular, this document in DVI format stored as `cl-patr.dvi`.

`grammars` directory: Sample grammars for use by CL-PATR including the sample grammar given in Appendix 3 is stored in the files `sample.patr`, `sample.gram`, and `sample.lex`, with appropriate sentences and LFs in `sample.data`. Variants of two other grammars taken from Lauri

Karttunen's D-PATR documentation are provided as toycgram and toypsggram. Both .patr and .data files are included.

ParEn directory: Source and object code for the driver subsystem of the ParEn parser environment system; this is auxiliary code developed at SRI International that is needed for CL-PATR to run. The driver subsystem of the ParEn parser environment system includes the following source files:

paren-3600.lisp	the main installation file for Symbolics 3600s
paren.lisp	the main installation file for other computers
def-parse.lisp	code for building parsers using the driver
lr-parse.lisp	parser driver for LR parsers

In general, the file cl-patr.lisp in the CL-PATR directory is configured to load the system from the distribution disk. This file must be modified, as per instructions in the file, to load the system from a different location such as a local hard disk or file server.

> **Compiling the Systems.** Full object code is distributed with the system, so users should not need to compile the code. However, users modifying the source, may find the following information useful. The file cl-patr.lisp is the main file that includes all the pertinent settings and functions for compiling and loading the system. In particular, it includes a function system-compile to recompile and load all files that have been modified since the last file compilation and a function system-load to load the most current or compiled versions of the files. All functions are in a separate patr package or other packages that inherit from lisp and user.
>
> Similarly, the package paren includes system-compile and system-load for compiling and loading the driver susbsystem of ParEn.

5.1 Installing the System on a Macintosh Computer

The system requires Allegro Common Lisp from Coral Software, version 1.2 or later. It is recommended that the system be run on Macintosh computers with at least 2 megabytes of memory. Code is distributed on a single 3.5 inch floppy disk. The contents of the disk can be copied to a hard disk or the system can be run directly from the distribution disk.

5.1.1 Running from the Distribution Disk

1. Open the disk icon and the folder CL-PATR. Double-click on the icon for the file init.lisp. Allegro Common Lisp should start up.

2. An error message will appear in the listener window. You should continue from this error with the <command>-/ key. The error is a result of CL-PATR installing a fix to Allegro Common Lisp version 1.0. The message should not appear if running Allegro CL version 1.2 or later.

3. From the Eval menu, choose the **Load** entry and load the file cl-patr.lisp, the main installation file for CL-PATR. A copyright message will be printed and a long series of files will be loaded. Finally, the interface window will be created and zoomed to cover the entire screen.

4. The system is now loaded and commands can be typed to the window.

If the underlying LISP encounters an error, it will enter a LISP break loop in the listener window. Typing `<command>-.` to the break loop will exit the break loop and move up one break level. Returning to the interface window allows reentering CL-PATR where it was stopped at the break. However, some extraneous characters may have been added at the end of the interface window; these can be safely deleted.

5.1.2 Running from the Hard Disk

1. Copy the contents of the distribution disk to the hard disk, preserving the directory structure.
2. Open the folder `CL-PATR`. Double-click on the icon for the file `init.lisp`. Allegro CL should start up.
3. An error message will appear in the listener window. You should continue from this error with the `<command>-/` key. The error is a result of CL-PATR installing a fix to Allegro CL version 1.0. The message should not appear if running Allegro CL version 1.2 or later.
4. Edit the file `cl-patr.lisp` as per the instructions in the file.
5. Continue from Step 3 of the instructions for running from the distribution disk.

5.2 Installing the System on a Sun Workstation

The system requires Lucid Common Lisp for the Sun workstations. Separate object code is available for Sun 2 and Sun 3 computers. In addition, GNU EMACS is needed if the extended interface is to be used. The EMACS interface assumes that the `shell.el` GNU code is available and that it is configured so that Lucid Lisp is loaded by default by `m-x run-lisp`.

After copying the files to the server disk from the distribution tape, and editing the file `cl-patr.lisp` as per the instructions in the file, run and load the system with the extended interface as per these steps.

1. Run GNU EMACS.
2. Load the file `gnu-iface.el` (with `m-x load-file`).
3. Execute the command `m-x run-patr`. A buffer for interacting with CL-PATR will be created and the system loaded. Special commands can be typed to this buffer.

Running the system with Lucid Lisp alone (without the EMACS extensions to the interface) proceeds as follows:

1. Run Lucid Lisp.
2. Load the file `cl-patr.lisp` with the `load` function. A whole series of other files will be loaded.

3. Call the function (`patr::patr t`). This starts up CL-PATR with its portable interface. Commands can be typed to the interface.

If the underlying Lisp encounters an error, it will enter a Lisp break loop. Typing `:a` to the break loop will exit the break loop all the way to the top level. Then calling the function (`patr::patr`) will restart CL-PATR where it stopped at the break.

5.3 Installing the System on an IBM-RT

The system requires Lucid Common Lisp for the IBM-RT workstation running IBM's A/IX operating system. Code is distributed on a single 5.25 inch floppy disk. The contents of the disk can be copied to the hard disk or the system can be run directly from the distribution disk.

5.3.1 Running from the Distribution Disk

1. Mount the disk as `/diskette0` with the A/IX command `mount /diskette0`.

2. Run Lucid Lisp.

3. Execute the Lisp call (`load "/diskette0/cl-patr/cl-patr.lisp"`).

4. Execute the Lisp call (`patr::patr t`).

5. The system is now loaded and running; commands can be typed to the prompt.

If the underlying LISP encounters an error, it will enter a LISP break loop. Typing `:a` to the break loop will exit the break loop all the way to the top level. Calling the function (`patr::patr`) will restart CL-PATR where it stopped at the break. The contents of the disk can be copied directly to the hard disk or the floppy disk can be used as the place where code resides.

5.3.2 Running from the Hard Disk

1. Mount the disk as `/diskette0` with the A/IX command `mount /diskette0`.

2. Copy the contents of the distribution disk to the hard disk, preserving the directory structure.

3. Edit the file `cl-patr.lisp` as per the instructions in the file.

4. Continue from Step 2 of the instructions for running from the distribution disk, except that the directory used in the load command should be changed appropriately.

5.4 Installing the System on a Symbolics 3600

To install the system on a Symbolics 3600 running Genera 7.0 or later software, the files are downloaded from a distribution tape as follows:

1. Create `.system` and `.translation` files for the CL-PATR system and the ParEn system. Appendix 13 includes samples of these files.
2. Place the distribution tape in a tape drive.
3. Type `Restore Distribution` to the command processor.
4. Edit the file `cl-patr.lisp` as per instructions in the file.
5. Type `Load System CL-PATR :version :newest` to the command processor.
6. Type `<select>-+` to create and expose a CL-PATR frame.

Chapter 6

Overview of CL-PATR Concepts

The functions provided by the CL-PATR system fall into a number of areas. In this chapter, we provide a taxonomy of these functions independent of the way in which they are accessed by the various interfaces to the system, and discuss the intended usage of these functions. This taxonomy will be used in later chapters to organize the descriptions of particular interface aspects.

The functionality of CL-PATR separates into six categories:

1. Interface interaction.
2. Grammar file manipulation.
3. Grammar testing.
4. Grammar debugging.
5. Grammar modification.
6. System configuration.

We will discuss each of these areas in turn.

6.1 Interface Interaction

A limited amount of the functionality of a CL-PATR interface is concerned with interaction with the interface itself, for example configuring the display or exiting the interface.

6.2 Grammar File Manipulation

As discussed in Chapter 2, grammatical information is codified in files containing grammar rules, lexical entries, and so forth. CL-PATR can access grammar information stored in one of two ways, either as the raw S-PATR grammar files described in Chapter 2, or as compiled versions of such files. Conventionally, a file containing the uncompiled, raw S-PATR form of a grammar is named with a .patr extension, e.g., sample.patr. (Such a file may use Input statements to load other files of grammatical information without the .patr extension, but the main file should have a .patr extension.) Similarly, the compiled version of a grammar uses .ptro (for 'PaTR Object') as its extension. Thus, the compiled version of the S-PATR grammar stored in sample.patr and any files it inputs would be named sample.ptro. In general, users of CL-PATR create only the .patr files; the compiled .ptro versions are created by CL-PATR automatically.

The accessing of grammatical information from these two sources goes by different names: .patr files are *read*, whereas .ptro files are *installed.*

The end result of reading a .patr file or installing its .ptro equivalent is identical; the grammatical information is available to the system. However, the installation of a .ptro file is much faster than the reading of its .patr equivalent.

While reading a .patr file, the system generates a compiled version in a file with the same name but with the .ptro extension. Thus a grammar need only be read once; as long as no changes have been made to the grammar, its automatically created .ptro can thereafter be installed.

Another class of files associated with grammars include data upon which the CL-PATR system is intended to operate, in particular, sentences to be parsed or logical forms to be generated from. These files conventionally have the extension .data. The format of such files is merely a sequence of double-quoted strings and LISP s-expressions corresponding to sentences to be parsed and logical forms to be generated from, respectively.

CL-PATR functions specifically geared toward grammar file manipulation include reading and installing .patr and .ptro files, respectively, listing the grammar files known to the system, and displaying information about the various grammar components such as rules and lexical entries.

A user might list the grammar files, then read or install one of the available files, and finally use the display facilities to browse through the grammar.

6.3 Grammar Testing

Once the grammatical information has been accessed, either by reading or installing a grammar, the grammar can be tested by presenting sentences to be parsed or logical forms to be generated from and perusing the results of these processes. CL-PATR performs such natural-language-processing tasks using a system based on chart parsing techniques. During parsing or generation, a *chart* stores all the partial and complete phrases built during the course of processing along with the grammatical information associated with the phrase encoded as a feature structure. Each such pairing of a phrase and associated information is a chart *edge*, which can be thought of as spanning the phrase between two string positions, or *vertices*. Edges associated with partial constituents are *active edges*; those associated with complete constituents are *passive edges*. Parsing or generation proceeds by combining an active edge with a passive edge immediately to its right.

CL-PATR includes two different parsing algorithms, one based on Earley's algorithm, and one that models a nondeterministic version of a shift-reduce parser. Details of the operations of these two algorithms and the chart-based generation algorithm are given in more detail in Chapter 12.

The ability to store a corpus of sentences and logical forms in a .data file is useful in grammar testing as well.

6.4 Grammar Debugging

After the parsing or generation process is completed, it is often useful to examine the contents of the chart to aid in the process of finding errors or inadequacies in the grammar. The various CL-PATR interfaces allow for display of this information, including information about chart vertices. In general, the edges in the chart are indexed by one of their vertices, passive edges by their leftmost vertex, and active edges by their rightmost vertex. Thus, when displaying information about a vertex, the edges listed will be those active edges ending at the vertex, and those passive edges starting at the vertex. Such a scheme presents exactly those active and passive edges that might be combined to form edges spanning longer strings.

In fact, the system allows for manual combination of two edges (that is, the *extension* of an active edge with a passive edge to its right) as one of its functions. The outcome of performing such parsing or generation steps by hand can often aid in the location of errors in the grammar.

6.5 Grammar Modification

Once an error in a grammar has been identified, a revision to the grammar must be made and the revised version put in effect. One method for doing this is to edit the original .patr file, and then read the file again. However, reading the entire file just to get the effect of a small change is quite wasteful. Instead, CL-PATR allows for the *incremental compilation* of changes to the grammar. Suppose a single grammar rule is to be changed. The edited version of the rule can be incrementally compiled; it will then be in force in later grammar testing, replacing the previous incarnation of the rule, without having to reread the entire grammar. A sequence of grammar rules can also be incrementally compiled.

Any component of a grammar—profile statements, macro definitions, grammar rules, lexical entries, etc.—can be incrementally compiled. The effect of the incremental compilation depends on the type of structure. Compiling a grammar rule replaces a rule of the same name, or adds the rule if no rule of the same name previously exists. When compiling a lexical entry for which a previous definition or definitions exist, the user is prompted as to whether the entry should replace all previous such definitions or augment them.

> **Compiling Multiple Lexical Entries.** When compiling a series of lexical entries with the same spelling, the system inquires about replacement of previous definitions each time. The user will probably want to replace previous definitions on the first inquiry, thereafter adding definitions without replacement.
>
> **Macro Changes Do Not Propagate.** Incrementally compiling a macro changes the macro definition, but previously read, installed, or compiled uses of the macro will not be affected by the change. Of course, grammar components using the changed macro can themselves be compiled incrementally, thus replacing the definitions using the old version of the macro with the new one.

6.6 System Configuration

CL-PATR provides for the modification of various parameters governing its behavior. For instance, the choice of which of the two parsing algorithms is used can be made.

6.7 Accessing the Functions

These various functions are accessed in different ways depending on which user interface is in force. The portable interface which runs on all machines is based on the invocation of *commands* to perform the CL-PATR functions. The user might give a command to list the grammar files or to read one of the listed files, to parse a sentence or to display the chart. The graphical interface for the Symbolics 3600 allows the functions to be accessed by clicking with a mouse on items in a menu or on pieces of text on the screen. For instance, the user might click on a menu item to list the grammar files or click on a file name to read that file, and so forth. Certain computers have enhanced versions of the portable interface that add simple mouse access to the command-oriented style of interaction. Despite the differing access methods, however, the same functionality is available in all interfaces.

The particulars of the various interfaces are discussed in the following chapters.

Chapter 7

The Command Interface

The portable command interface is a modified LISP read-eval-print (REP) loop augmented in two ways:

1. Objects are labeled with numeric labels, which allows the user to refer to them later and to redisplay them in an expanded format for *browsing.*

2. Certain *special commands* are known to the system, which use a simple non-LISP syntax and which can operate on the labeled objects.

> **Extensions to the Command Interface.** Standard extensions to this interface are available on Macintosh computers and UNIX systems running Gnu EMACS (e.g., Suns) that allow for single keystroke equivalents to the special commands. See Chapter 11 for details.

Chapter 8 includes an annotated session with the command interface.

7.1 Basic Structure of the Interface

The interface issues the command prompt ' > '. Any LISP s-expression can be typed in response to the prompt for evaluation, or a special command (described below) can be invoked. For instance, after reading a grammar and parsing a sentence, the `Chart` command can be used to provide a printed representation of the vertices in the chart.

```
  > chart
68>   <0>    <--- Crockett --->
69>   <1>     <--- wants --->
70>   <2>       <--- to --->
71>   <3>   <--- ventilate --->
72>   <4>   <--- criminals --->
73>   <5>
```

Notice that after each command is read and its response is given, a separating line of dots is printed to demarcate the interaction from the next such. (The printing of these separators is controlled by the `Separators` parameter described in Section 7.7.)

> **Interface as a LISP Interpreter.** Arbitrary s-expressions can be typed to the interface. Lists are evaluated, as are numbers, strings, and bound symbols. Unbound symbols typed to the top-level prompt are ignored.

```
    > 3
   1> 3
........................................................................
    > foo ; This unbound symbol is ignored.  The user is prompted again.
    > '(a b c) ; This list is evaluated...
   2> (A B C)
........................................................................
    > (print 'test) ; as is this one.
TEST
   3> TEST
........................................................................
```

7.1.1 Labeling of objects

Labeled lines are output with the format seen above. The marker string '> ' separates a numeric label from a printed representation of the labeled object. In the example, one of the lines is labeled with the number '`71`'; and the labeled object is the vertex following the third word in the sentence 'Crockett wants to ventilate criminals', which was presumably parsed previously.

7.1.2 Browsing with the `Display` Command

Labeled objects can be referred to in certain commands by their labels. Of particular interest is the `Display` command which takes a label as its argument and merely redisplays the object so labeled. The redisplaying typically presents more information than is given on the original labeled line, or breaks the information up so that components are labeled for later reference. In this way, the `Display` command allows browsing through data structures of interest.

For instance, further information about the chart vertices listed above by the `Chart` command can be displayed.

```
    > Display 71 ;   <3>    <--- ventilate --->
  74> Vertex: <3>
  75>   Next vertex: <4>
  76>   Following terminal: ventilate
  77>   Incoming actives...
  78>   Outgoing passives...
........................................................................
    > Display 77 ;   Incoming actives...
    79>   <3>--  VP --> . VP AdvP  / /  --<3>
    80>   <3>--  VP --> . VP X  / /  --<3>
    81>   <3>--  VP --> . V  / /  --<3>
    82>   <3>--  VP --> . VP AdvP  / /  --<3>
    83>   <3>--  VP --> . VP X  / /  --<3>
    84>   <3>--  VP --> . V  / /  --<3>
    85>   <2>--  VP --> VP . VP  / to /  --<3>
.....................................................................................
     > Display 78 ;   Outgoing passives...
    86>   <3>--  VP --> VP NP[plural] .  / ventilate criminals /  --<5>
    87>   <3>--  VP --> VP NP[plural] .  / ventilate criminals /  --<5>
    88>   <3>--  VP --> V .  / ventilate /  --<4>
    89>   <3>--  VP --> V .  / ventilate /  --<4>
    90>   <3>--  V --> ventilate .  / ventilate /  --<4>
    91>   <3>--  V --> ventilate .  / ventilate /  --<4>
.....................................................................................
```

Here vertex number 3 is redisplayed in more detail, and its list of incoming and outgoing edge lists can then themselves be displayed.

7.1.3 Special Commands

As we have seen, the basic interaction with the interface is through typing of special commands to the prompt. Already, we have seen examples of two special commands, the `Chart` and `Display` commands.

Special commands are typically single words. Certain commands (like `Display`) take arguments that are typed following the command on the same line.

> **All Commands on One Line.** Unlike , does not use parentheses to delimit the start and end of a special command. For this reason, all input to the top-level interface is assumed to be given on a single line. This includes s-expressions.

> **Command Arguments Are Not Evaluated.** The arguments to special commands are not evaluated. For instance, if the variable `x` has the value 3, a '`Display x`' command will not display the object labeled 3, but instead will lead to an error as the argument provided the `Display` command was the symbol `x` rather than an integer.

> **Known Bug: Argument Checking.** Some commands do insufficient checking of their arguments.

The next sections describe how the special commands can be used to access all the functions of the CL-PATR system.

7.2 Interface Interaction Commands

The following commands deal with generic capabilities of the interface.

Browsing displayed information: The Display command is described in Section 7.1.2 above and is used for browsing by redisplaying labeled objects usually in an expanded format. It takes a single integer label as its argument.

Listing available commands: The Help command takes no arguments and displays information about the available special commands.

```
> Help
   DISPLAY     <return>, <enter>  Display information about a labeled object
   HELP        <help>  Supply help information about existing special commands
   RESTART     Restart the history mechanism
   FILES       List grammar and data files
   READ        <c-m-r>  Read a .patr grammar FILE
   INSTALL     <c-m-i>  Install the compiled '.ptro' version of FILE (a
               filename).
   UPDATE      Incrementally update the grammar according to a STRING containing
               grammar rules.
   GENERATE    <c-m-g>  Generate sentences with logical form LF (an
               s-expression).
   PARSE       <c-m-p>  Parse a SENTENCE using the current grammar.
   EXTEND      Attempt to extend an ACTIVE EDGE with a PASSIVE EDGE.
   CHART       Display chart vertices.
   TREE        <c-m-t>  Display parse tree representation for EDGE (an edge).
   RULES       List rules in the grammar.
   RULE        Print the rule associated with a given SPELLING (a string).
   WORDS       List words in the grammar.
   WORD        Print the lexical entry associated with a given SPELLING (a
               string).
   STEMS       List stems in the grammar.
   STEM        Print the stem definition associated with a given SPELLING (a
               string).
   MACROS      List macros in the grammar.
   MACRO       Print macro definition associated with a given SPELLING (a
               string).
   SHOW        Show the current value of a configuration PARAMETER
   SET         Assign to a configuration PARAMETER a given VALUE
```

Information displayed by the Help command includes: the name of the command through which it can be invoked; a list of keystroke equivalents for the command (as described in Sections 11.1.1 and 11.2.1); a short description of the command's function, including an idea of its arguments (capitalized and often described separately).

Clearing the labeling mechanism: The labeling process can be cleared and restarted with the Restart command. After execution, labeling of objects begins with the label '1' again. Previously labeled objects are discarded and can no longer be referred to by their labels.

Exiting the system: Certain ports of the system include an `Exit` command to exit the interface and return control to the LISP system in which CL-PATR is executing.

7.3 Grammar File Manipulation Commands

Listing available files: The `Files` command prints a list of all the files with `.patr` or `.data` extensions that reside on grammar directories. The grammar directories are set when the CL-PATR system is installed for the first time by setting the value of the variable `*cl-patr-grammar-file-pathnames*`. See Section 5.

```
 > Files
 8> #.(pathname "internal:cl-patr:grammars:sample.patr")
 9> #.(pathname "internal:cl-patr:grammars:sep87.patr")
10> #.(pathname "internal:cl-patr:grammars:toycgram.patr")
11> #.(pathname "internal:cl-patr:grammars:toycgram2.patr")
12> #.(pathname "internal:cl-patr:grammars:toypsggram.patr")
13> #.(pathname "internal:cl-patr:grammars:sample.data")
14> #.(pathname "internal:cl-patr:grammars:sep87.data")
15> #.(pathname "internal:cl-patr:grammars:toycgram.data")
16> #.(pathname "internal:cl-patr:grammars:toypsggram.data")
```

The labeled filenames can be used as the arguments to the `Read` and `Install` commands. Note that although the file type in the listed pathnames is `.patr`, the `Install` command will find the `.ptro` version if it exists.

Reading grammar files: The `Read` command prompts for a file name, and reads the specified file in S-PATR format. (See Chapter 2.) The state of the system is modified so that the grammar read is in force for parsing and generation. At the same time, the grammar is converted into an intermediate LISP representation. This representation is printed in a file with the same name as the input file but with the extension `.ptro` and is intended for later use by the `Install` command.

The input file can be specified by giving a string surrounded by double quotes with the pathname of the file. If no extension is given, the extension `.patr` will be added. Alternatively, a labeled item can be used to specify a file if the object is a pathname, as, for instance, provided by the `Files` command.

As the grammar file (along with any auxiliary files specified with `Input` statements) is loaded, an appropriate message can be printed for each statement read and processed. (See the discussion of the `Reading` parameter in Section 7.7.) In case of a parsing error, the reader will attempt to resynchronize itself by skipping tokens in the file until the next period is found marking the end of a statement. It then continues reading the file, processing further statements.

Reading data files: A filename with the extension `.data` is read differently by the `Read` command. Rather than interpreting the contents of the file as a grammar, the contents are read as LISP objects and listed as labeled objects. In particular, double-quoted strings and lists in the file are available for later reference by their labels as the arguments to the `Parse` and `Generate` commands, respectively.

```
> Read 8 ; #.(pathname "internal:cl-patr:grammars:sample.patr")
```

```
Loading CL-PATR Sample Grammar

Starting file internal:cl-patr:grammars:sample.gram
Finishing file internal:cl-patr:grammars:sample.gram
Starting file internal:cl-patr:grammars:sample.lex
Warning: implicitly declaring 3sg to be a feature.
Warning: implicitly declaring aux to be a feature.
Finishing file internal:cl-patr:grammars:sample.lex
..........................................................................
     > Read 13 ;#.(pathname "internal:cl-patr:grammars:sample.data")
   17> Crockett fumes
   18> Crockett ventilated Tubbs
   19> Crockett wants to ventilate criminals
   20> Crockett seems to want to ventilate criminals
   21> Crockett was ventilated
   22> criminals were killed
   23> Castillo said Sonny shot Rico yesterday
   24> Castillo said Sonny was shot
   25> Sonny loves Kait passionately
   26> (FUME CROCKETT)
   27> (VENTILATE CROCKETT TUBBS)
   28> (WANT CROCKETT (VENTILATE CROCKETT CRIMINALS))
   29> (SEEM (WANT CROCKETT (VENTILATE CROCKETT CRIMINALS)))
   30> (KILL ?X CROCKETT)
   31> (KILL ?X CRIMINALS)
   32> (YESTERDAY (SAY CASTILLO (SHOOT SONNY RICO)))
   33> (SAY CASTILLO (SHOOT ?X RICO))
   34> (PASSIONATELY (LOVE SONNY KAIT))
..........................................................................
```

Installing grammar files: The Install command prompts for a file name as the Read command does, but adds the extension .ptro by default, and installs that file's grammatical information, which should be in in intermediate representation. Installation consists of loading the intermediate representation of the grammar so that the state of the system is the same as if the .patr file had been read. However, Install is typically much faster than Read.

The following commands are used for browsing through the currently loaded grammar.

Listing all rules, words, macros, stems: The Rules, Words, Macros, and Stems commands list the rules, words, macros, or stems, respectively, of the grammar as labeled objects, thereby enabling them to be browsed.

Listing particular rules, words, macros, stems: The Rule, Word, Macro, and Stem commands prompt for a string or symbol identifying the grammatical element and lists the matching rules, words, macros, or stems, respectively as labeled objects, thereby enabling them to be browsed. Rules are identified by their identifier string, all others are identified by spelling. If a string is entered, it is used directly for lookup. Symbols, however, are forced to lowercase before lookup.

7.4 Grammar Testing Commands

The following commands are used to parse or generate sentences.

Generating bottom-up from a logical form: The `Generate` command takes as its argument an s-expression or label whose object is an s-expression, and interprets the expression as a logical form to be generated from. The logical form is converted to DG form and used as the input to the bottom-up chart-based generator.

The logical form conversion works as follows. If the s-expression is an atom starting with a question mark, its conversion is a variable (null DG). All such atoms in the s-expression with the same spelling are converted to the token-identical variable DG. If the s-expression is any other atom, the DG conversion is the atomic DG with that name. If the s-expression is a list, the elements of the list are each converted recursively and made the values of the features *pred*, *arg1*, *arg2*, etc., respectively. Finally, the DG thus generated is taken to be the value of the start DG's semantics path as specified in the `Semantics path` statement.

```
    > Generate 26 ; (FUME CROCKETT)
      Real time elapsed: 8.77
       Run time elapsed: 8.78
                Actives: 9
               Passives: 15
                  Total: 24
           Agenda items: 46
  97> <0>--  S --> NP[singular3] VP .  / Crockett fumed /  --<0>
.................................................................................
```

Parsing sentences: The `Parse` command takes as its argument a string or label whose object is a string or edge and parses the string with respect to the currently installed grammar. If an edge label is given, the string covered by that edge is used as the sentence to be parsed. The chart edges corresponding to the full parses for the sentences are listed as labeled objects for browsing. Also the chart is available with the `Chart` command.

```
  > Parse 19 ; Crockett wants to ventilate criminals
    Real time elapsed: 8.40
     Run time elapsed: 8.37
              Actives: 28
             Passives: 15
                Total: 43
         Agenda items: 74
35> <0>--  S --> NP[singular3] VP .  / Crockett ... criminals /  --<5>
```

Generating top-down: The `Top-down` command implements a simple top-down, depth-first, backtracking generator. The start DG is expanded according to one of the rules or lexical items in the grammar to form a partial tree. The leftmost unexpanded node in the partial tree is then further expanded, and so forth, until a full tree has been generated or no further expansion is possible. In the former case, the sentence generated is printed and the user can type a newline to exit the generator, a semicolon to backtrack for more solutions, or a p to push into a new labeling REP loop (typically so that the user can use the `Set` command to change some parameters during generation). Exiting this secondary REP loop with `Exit` leaves the user in the same state of being prompted for a newline, semicolon, or p.

Because depth-first generation of this type has well-known problems with left-recursion, the search is depth-limited. The depth limit can be changed with the `Max depth` parameter.

The choice of whether a word or rule should be used for expansion and in what order the various possibilities should be tried is governed again by various configuration parameters (see Section 7.7). The `Choice mode` parameter determines whether the choice is made interactively, by textual order (a la Prolog), or randomly. The `Choices` parameter determines whether each choice made by the generator is reported to the user. Indentation of the choices corresponds to recursion depth (i.e., how far from the root the expanded node is). The `Backtracking` parameter determines whether a report is made whenever backtracking is forced.

> **Known Bug: No Top-down Command.** The Top-down command has not yet been incorporated into the portable interface.

7.5 Grammar Debugging Commands

The following commands provide for the perusal of the results of parsing and generation and other grammar debugging functions.

Displaying the chart: The `Chart` command prints a representation of the chart where each vertex of the chart is printed as a separate labeled item.

```
  > chart
68>   <0>     <--- Crockett --->
69>   <1>      <--- wants --->
70>   <2>        <--- to --->
71>   <3>    <--- ventilate --->
72>   <4>    <--- criminals --->
73>   <5>
```

Browsing the chart The `Display` command can be used in the normal way to display further information about the chart vertices.

Displaying parse trees: The Tree command takes as its argument a label for an edge. The edge and all inferior passive edges in the chart are printed in an indented tree format with each line in the output being a labeled item that can be further browsed.

```
  > Tree 35 ; <0>--  S --> NP[singular3] VP .  / Crockett ... criminals /  --<5>
41> S
42>   NP[singular3]
43>     Crockett
44>   VP
45>     VP
46>       V
47>         wants
48>     VP
49>       VP
50>         V
51>           to
52>       VP
53>         VP
54>           V
55>             ventilate
56>         NP[plural]
57>           criminals
```

Extending edges: The Extend command takes as arguments two labels, one for an active edge and one for a passive edge which abut in the chart, that is the active edge ends at the vertex the passive edge starts at. (The edges would therefore both be available by browsing the single vertex that lies between them.) An attempt is made to extend the active edge with the passive edge by unifying the passive edge DG with the DG of the next child constituent to be found in completing the active edge. Then the DG corresponding to the active edge after the extension is displayed whether or not the attempt was successful. If the attempt was unsuccessful, the clash in the DG will be put in the display as well.

```
> extend 85 87
  Unification failed.
   [*OtherComplements*: *3*=()
    *Subject*: *2*=[cat: NP
                    head: [agreement: [3sg: no]
                           trans: *1*=[]]]
    0: [cat: VP
        head: *4*=[form: infinitival
                   trans: *5*=[pred: VENTILATE
                               arg1: *1*
                               arg2: CRIMINALS]]
        subcat: [first: *2*
                 rest: *3*]]
    1: [cat: VP
    head: *4*
    subcat: [first: *2*
             rest: [first: *6*=[cat: VP
                                head: [form: finite
                                             |__ failed
                                             |   against
                                             nonfinite
                                       trans: [pred: VENTILATE
                                               arg1: []
                                               arg2: CRIMINALS]
                                       aux: false]
                                subcat: [first: [cat: NP
                                                 head: [agreement: [3sg: no]
                                                        trans: []]]
                                         rest: ()]]
                    rest: ()]]]
2: [cat: VP
    head: [form: finite
           trans: [pred: VENTILATE
                   arg1: []
                   arg2: CRIMINALS]
           aux: false]
    subcat: [first: [cat: NP
                     head: [agreement: [3sg: no]
                            trans: []]]
             rest: ()]]]
```

Notice on the preceding example that the value for `<1 subcat rest first head form>` is marked as having two contradictory values, `finite` and `nonfinite`.

7.6 Grammar Modification Commands

Updating a portion of the grammar: The `Update` command prompts for a double-quoted string and interprets the string as a series of PATR statements, an update to the currently loaded grammar. The statements are parsed and the grammar tables are modified or augmented to reflect the changed or additional rules. A grammar rule replaces a previous definition with the same identifier. Macros and stem definitions replace previous definitions with the same spelling. Word definitions either replace or augment previous definitions with the same spelling; the user is prompted for whether the previous definitions should be removed or stay extant. See the notes on incremental compilation in Chapter 6.

7.7 System Configuration Commands

Various modifiable parameters control the operation of the system. This section describes the `Set` and `Show` commands for manipulating these parameters and lists the various parameters themselves. Unless otherwise specified, the possible values for a parameter are the two symbols on and off.

Displaying the current parameter settings: The Show command takes an optional series of symbols as arguments and lists the current value of the parameter with that name. With no argument, lists the current values of all the parameters.

Modifying the value of a parameter: The Set command takes an optional series of symbols as arguments and interprets the symbols as sequences of a parameter name followed by a value to assign to the parameter. In this way, several parameters can be set with one command. If no arguments are given, or the single argument ?, the command lists the current values of all the parameters.

```
     > show
           Max depth:         20
           Choice mode:       Interactive
           Choices:           Off
           Backtrack:         Off
           Agenda tracing:    Off
           Agenda additions:  Off
           Edge tracing:      Off
           Statistics:        On
           Timings:           On
           Reading:           Off
           Installation:      Off
           Print level:       5
           Algorithm:         Earley
           Separators:        On
................................................................................
     > set print level 10
           Setting print level to 10.
................................................................................
     > show print level
           Print level:       10
................................................................................
```

7.7.1 Configuration Parameters

The following parameters control the current configuration of the system.

Max depth: Changes the depth limit used by the top-down generator invoked by the Top-down command. The value is an integer.

Choice mode: Changes the method of making choices concerning search order in top-down generation. The possible settings are ordered, interactive, or random. With the ordered setting, the generator always attempts to expand a node with a lexical entry before any grammar rules are tried; the choice of which lexical entry or grammar rule to use is determined by the textual order of definition in the grammar itself like Prolog search order. The interactive setting places all decisions at the discretion of the user, leading to a very tedious generation process. The random setting makes choices randomly, although backtracking maintains the choice point structure.

Choices: Determines whether trace lines are displayed for all choices made during top-down generation by the `Top-down` command. The trace lines are indented to reflect the recursion depth of the choice, i.e., the distance from the root to the node being expanded.

Backtracking: Determines whether trace lines are displayed for all invocations of backtracking made during top-down generation by the `Top-down` command. Indentation is similar to that in trace lines for choices.

Agenda tracing: Determines whether a trace line is printed whenever an agenda item is removed from the agenda and processed by placing in the chart.

Agenda additions: Determines whether a trace line is printed whenever an agenda item is added to the agenda.

Edge additions: Determines whether a trace line is printed whenever an edge is added to the chart.

Statistics: Determines whether statistics are kept as to the number of edges of various sorts (active, passive, hypothesis, empty) built during the course of parsing or generation. (This applies only to the chart-based operations, not `Top-down`.)

Timings: Determines whether statistics are kept as to the run time and elapsed time required for parsing or generation. (This applies only to the chart-based operations, not `Top-down`.)

Reading: Determines whether a trace of every token read during a `Read` command is echoed to the screen. This might be useful in testing to see at what point a reading error occurs in the file. If used for this purpose, the user should keep in mind that the tokenizer reads ahead by one token so that the metaparser can use a one token lookahead.

Installation: Determines whether a trace line is printed after each statement is installed using the `Read` or `Install` commands.

Print level: Determines how deep DGs can be printed before being replaced with a labeled item as a stand-in for the rest of the DG.

Algorithm: Determines which parsing algorithm to use. Currently, two parsing algorithms are available with this mechanism: one which runs the chart system as an Earley parser and one which simulates a nondeterministic shift-reduce parser so that parses are generated in right association, minimal attachment order. The possible values, therefore, are `earley` and `sr`.

Separators: Determines whether a separating line of dots is printed to demarcate each iteration of the top-level loop.

> **Known Bug: Missing Parameters.** The verbose edge printing flag and the grammar file pathname list should be added to the parameter list.

Chapter 8

Sample Session with the Command Interface

This chapter presents a session with CL-PATR running under the command interface.

```
                      CL-PATR (tm) Grammar Development Environment

             Unpublished rights reserved under the copyright laws of the United
             States.

             This data and information is proprietary to, and a valuable trade
             secret of, SRI International.  It is given in confidence by SRI
             International.  Its use, duplication, or disclosure is subject to
             the restrictions set forth in the License Agreement under which it
             has been distributed.


                    Unpublished Copyright (c) 1987, SRI International
                  CL-PATR and PATR are Trademarks of SRI International

Initializing metagrammar tables from #P"internal:cl-patr:spatr-tables.fasl".
;Loading "internal:cl-patr:spatr-tables.fasl"...


     > 3
    1> 3
................................................................................
     > '(a b c)
    2> (A B C)
................................................................................
     > (print 'test)
TEST
    3> TEST
................................................................................
```

```
     > Display 1
    4> 3
.................................................................................
     > Display 2
    5>   Element: A
    6>   Element: B
    7>   Element: C
.................................................................................
     > Help
        DISPLAY     <return>, <enter>  Display information about a labeled object
        HELP        <help>  Supply help information about existing special commands
        RESTART     Restart the history mechanism
        FILES       List grammar and data files
        READ        <c-m-r>  Read a .patr grammar FILE
        INSTALL     <c-m-i>  Install the compiled '.ptro' version of FILE (a filename).
        UPDATE      Incrementally update the grammar according to a STRING containing
                    grammar rules.
        GENERATE    <c-m-g>  Generate sentences with logical form LF (an s-expression).
        PARSE       <c-m-p>  Parse a SENTENCE using the current grammar.
        EXTEND      Attempt to extend an ACTIVE EDGE with a PASSIVE EDGE.
        CHART       Display chart vertices.
        TREE        <c-m-t>  Display parse tree representation for EDGE (an edge).
        RULES       Lists rules in the grammar.
        RULE        Print the rule associated with a given SPELLING (a string).
        WORDS       List words in the grammar.
        WORD        Print the lexical entry associated with a given SPELLING (a string).
        STEMS       List stems in the grammar.
        STEM        Print the stem definition associated with a given SPELLING (a string).
        MACROS      List macros in the grammar.
        MACRO       Print macro definition associated with a given SPELLING (a string).
        SHOW        Show the current value of a configuration PARAMETER
        SET         Assign to a configuration PARAMETER a given VALUE
.................................................................................
     > Files
    8> #.(pathname "internal:cl-patr:grammars:sample.patr")
    9> #.(pathname "internal:cl-patr:grammars:sep87.patr")
   10> #.(pathname "internal:cl-patr:grammars:toycgram.patr")
   11> #.(pathname "internal:cl-patr:grammars:toycgram2.patr")
   12> #.(pathname "internal:cl-patr:grammars:toypsggram.patr")
   13> #.(pathname "internal:cl-patr:grammars:sample.data")
   14> #.(pathname "internal:cl-patr:grammars:sep87.data")
   15> #.(pathname "internal:cl-patr:grammars:toycgram.data")
   16> #.(pathname "internal:cl-patr:grammars:toypsggram.data")
.................................................................................
     > Read 8 ; #.(pathname "internal:cl-patr:grammars:sample.patr")


Loading CL-PATR Sample Grammar

Starting file internal:cl-patr:grammars:sample.gram
Finishing file internal:cl-patr:grammars:sample.gram
Starting file internal:cl-patr:grammars:sample.lex
Warning: implicitly declaring 3sg to be a feature.
Warning: implicitly declaring aux to be a feature.
Finishing file internal:cl-patr:grammars:sample.lex
```

```
...............................................................................
    > Read 13 ;#.(pathname "internal:cl-patr:grammars:sample.data")
  17> Crockett fumes
  18> Crockett ventilated Tubbs
  19> Crockett wants to ventilate criminals
  20> Crockett seems to want to ventilate criminals
  21> Crockett was ventilated
  22> criminals were killed
  23> Castillo said Sonny shot Rico yesterday
  24> Castillo said Sonny was shot
  25> Sonny loves Kait passionately
  26> (FUME CROCKETT)
  27> (VENTILATE CROCKETT TUBBS)
  28> (WANT CROCKETT (VENTILATE CROCKETT CRIMINALS))
  29> (SEEM (WANT CROCKETT (VENTILATE CROCKETT CRIMINALS)))
  30> (KILL ?X CROCKETT)
  31> (KILL ?X CRIMINALS)
  32> (YESTERDAY (SAY CASTILLO (SHOOT SONNY RICO)))
  33> (SAY CASTILLO (SHOOT ?X RICO))
  34> (PASSIONATELY (LOVE SONNY KAIT))
...............................................................................
    > Parse 19 ; Crockett wants to ventilate criminals
      Real time elapsed: 8.40
       Run time elapsed: 8.37
                Actives: 28
               Passives: 15
                  Total: 43
           Agenda items: 74
  35> <0>--  S --> NP[singular3] VP .  / Crockett wants to ventilate criminals /  --<5>
...............................................................................
    > Display 35 ; <0>--  S --> NP[singular3] VP .  / Crockett wants to ventilate
                                                      criminals /  --<5>
  36> Edge: <0>--  S --> NP[singular3] VP .  / Crockett wants to ventilate criminals /  --<5>
  37>   Rule:sentence formation
        Sources of the edge:
  38>     Extended: <0>--  S --> NP[singular3] . VP  / Crockett /  --<1>
  39>         with: <1>--  VP --> VP VP .  / wants to ventilate criminals /  --<5>
  40>   Directed graph: ...
...............................................................................
    > Display 40 ;  Directed graph: ...
           [0: [cat: S
                head: *3*=[form: finite
                           trans: [pred: WANT
                                   arg1: *1*=CROCKETT
                                   arg2: [pred: VENTILATE
                                          arg1: *1*
                                          arg2: CRIMINALS]]]]
            1: *2*=[cat: NP
                    head: [agreement: [person: 3
                                       number: singular
                                       gender: masculine
                                       3sg: yes]
                           trans: *1*]
                    word: *1*]
            2: [cat: VP
```

```
                head: *3*
                subcat: [first: *2*
                         rest: ()]]]
.................................................................................
    > Tree 35 ; <0>--  S --> NP[singular3] VP .  / Crockett wants to ventilate
                                                     criminals /  --<5>
  41> S
  42>   NP[singular3]
  43>     Crockett
  44>   VP
  45>     VP
  46>       V
  47>         wants
  48>     VP
  49>       VP
  50>         V
  51>           to
  52>       VP
  53>         VP
  54>           V
  55>             ventilate
  56>         NP[plural]
  57>           criminals
.................................................................................
    > Display 48 ;     VP
  58> Edge: <2>--  VP --> VP VP .  / to ventilate criminals /  --<5>
  59>   Rule:complements
        Sources of the edge:
  60>     Extended: <2>--  VP --> VP . VP  / to /  --<3>
  61>         with: <3>--  VP --> VP NP[plural] .  / ventilate criminals /  --<5>
  62>   Directed graph: ...
.................................................................................
    > Display 62 ;    Directed graph: ...
           [*OtherComplements*: *4*=()
            *Subject*: *2*=[cat: NP
                            head: [trans: *1*=[]]]
            0: [cat: VP
                head: *6*=[form: infinitival
                           trans: *3*=[pred: VENTILATE
                                       arg1: *1*
                                       arg2: CRIMINALS]]
                subcat: [first: *2*
                         rest: *4*]]
            1: [cat: VP
                head: *6*
                subcat: [first: *2*
                         rest: [first: *5*=[cat: VP
                                            head: [form:
  63>                                                       ...
                                                    trans:
  64>                                                        ...
                                                    aux:
  65>                                                      ...]
                                            modified: no
                                            subcat: [first:
```

```
  66>                                                           ...
                                                        rest:
  67>                                                         ...]]
                                 rest: *4*]]]
             2: *5*]
.................................................................................
    > chart
  68>   <0>     <--- Crockett --->
  69>   <1>      <--- wants --->
  70>   <2>        <--- to --->
  71>   <3>    <--- ventilate --->
  72>   <4>    <--- criminals --->
  73>   <5>
.................................................................................
    > Display 71 ;   <3>    <--- ventilate --->
  74> Vertex: <3>
  75>   Next vertex: <4>
  76>   Following terminal: ventilate
  77>   Incoming actives...
  78>   Outgoing passives...
.................................................................................
    > Display 77 ;   Incoming actives...
  79>   <3>--  VP --> . VP AdvP  / /  --<3>
  80>   <3>--  VP --> . VP X  / /  --<3>
  81>   <3>--  VP --> . V  / /  --<3>
  82>   <3>--  VP --> . VP AdvP  / /  --<3>
  83>   <3>--  VP --> . VP X  / /  --<3>
  84>   <3>--  VP --> . V  / /  --<3>
  85>   <2>--  VP --> VP . VP  / to /  --<3>
.................................................................................
    > Display 78 ;   Outgoing passives...
  86>   <3>--  VP --> VP NP[plural] .  / ventilate criminals /  --<5>
  87>   <3>--  VP --> VP NP[plural] .  / ventilate criminals /  --<5>
  88>   <3>--  VP --> V .  / ventilate /  --<4>
  89>   <3>--  VP --> V .  / ventilate /  --<4>
  90>   <3>--  V --> ventilate .  / ventilate /  --<4>
  91>   <3>--  V --> ventilate .  / ventilate /  --<4>
.................................................................................
    > extend 85 87
      Unification failed.
           [*OtherComplements*: *4*=()
            *Subject*: *2*=[cat: NP
                            head: [agreement: [3sg: no]
                                   trans: *1*=[]]]
            0: [cat: VP
                head: *6*=[form: infinitival
                           trans: *3*=[pred: VENTILATE
                                       arg1: *1*
                                       arg2: CRIMINALS]]
                subcat: [first: *2*
                         rest: *4*]]
            1: [cat: VP
                head: *6*
                subcat: [first: *2*
                         rest: [first: *5*=[cat: VP
```

```
                                            head: [form:
  92>                                                    ...
                                                   trans:
  93>                                                     ...
                                                   aux:
  94>                                                   ...]
                                            subcat: [first:
  95>                                                       ...
                                                     rest:
  96>                                                      ...]]
                                rest: *4*]]]
            2: *5*]
................................................................................
    > show
          Max depth:         20
          Choice mode:       Interactive
          Choices:           Off
          Backtrack:         Off
          Agenda tracing:    Off
          Agenda additions:  Off
          Edge tracing:      Off
          Statistics:        On
          Timings:           On
          Reading:           Off
          Installation:      Off
          Print level:       5
          Algorithm:         Earley
          Separators:        On
................................................................................
    > set print level 10
          Setting print level to 10.
................................................................................
    > show print level
          Print level:       10
................................................................................
    > extend 85 87
      Unification failed.
           [*OtherComplements*: *3*=()
            *Subject*: *2*=[cat: NP
                            head: [agreement: [3sg: no]
                                   trans: *1*=[]]]
            0: [cat: VP
                head: *4*=[form: infinitival
                           trans: *5*=[pred: VENTILATE
                                       arg1: *1*
                                       arg2: CRIMINALS]]
                subcat: [first: *2*
                         rest: *3*]]
            1: [cat: VP
                head: *4*
                subcat: [first: *2*
                         rest: [first: *6*=[cat: VP
                                            head: [form: finite
                                                         |__ failed
                                                         |   against
```

```
                                                          nonfinite
                                                    trans: [pred: VENTILATE
                                                           arg1: []
                                                           arg2: CRIMINALS]
                                                    aux: false]
                                             subcat: [first: [cat: NP
                                                              head: [agreement: [3sg: no]
                                                                     trans: []]]
                                                      rest: ()]]
                                 rest: ()]]]
             2: [cat: VP
                 head: [form: finite
                        trans: [pred: VENTILATE
                                arg1: []
                                arg2: CRIMINALS]
                        aux: false]
                 subcat: [first: [cat: NP
                                  head: [agreement: [3sg: no]
                                         trans: []]]
                          rest: ()]]]
................................................................................
     > Generate 26 ; (FUME CROCKETT)
       Real time elapsed: 8.77
        Run time elapsed: 8.78
                 Actives: 9
                Passives: 15
                   Total: 24
            Agenda items: 46
   97> <0>--  S --> NP[singular3] VP .  / Crockett fumed /  --<0>
................................................................................
     > Tree 97 ; <0>--  S --> NP[singular3] VP .  / Crockett fumed /  --<0>
   98> S
   99>   NP[singular3]
  100>     Crockett
  101>   VP
  102>     V
  103>       fumed
................................................................................
     > words
  104> Word:Crockett
  105> Word:Sonny
  106> Word:Tubbs
  107> Word:Ricardo
  108> Word:Rico
  109> Word:Castillo
  110> Word:Gina
  111> Word:Kait
  112> Word:detectives
  113> Word:criminals
  114> Word:says
  115> Word:say
  116> Word:say
  117> Word:said
  118> Word:fumes
  119> Word:fume
```

```
 120> Word:fume
 121> Word:fumed
 122> Word:ventilates
 123> Word:ventilate
 124> Word:ventilate
 125> Word:ventilated
 126> Word:ventilated
 127> Word:kills
 128> Word:kill
 129> Word:kill
 130> Word:killed
 131> Word:killed
 132> Word:shoots
 133> Word:shoot
 134> Word:shoot
 135> Word:shot
 136> Word:shot
 137> Word:loves
 138> Word:love
 139> Word:love
 140> Word:loved
 141> Word:loved
 142> Word:is
 143> Word:was
 144> Word:were
 145> Word:to
 146> Word:seems
 147> Word:seem
 148> Word:seem
 149> Word:seemed
 150> Word:wants
 151> Word:want
 152> Word:want
 153> Word:wanted
 154> Word:passionately
 155> Word:quickly
 156> Word:happily
 157> Word:yesterday
..............................................................................
    > Display 104 ; Word:Crockett
 158> Word:Crockett
        Constraints:
                   <* word> = CROCKETT
                   Masc
                   ProperNoun
 159>   Directed graph: ...
..............................................................................
    > Display 159 ;   Directed graph: ...
           [cat: NP
            head: [agreement: [person: 3
                               number: singular
                               gender: masculine
                               3sg: yes]
                   trans: *1*=CROCKETT]
            word: *1*]
```

```
..........................................................................
    > word wanted
  160> Word:wanted
..........................................................................
    > Display 160 ; Word:wanted
  161> Word:wanted
        Constraints:
                  <* word> = WANTED
                  Finite
                  *want*
  162>   Directed graph: ...
..........................................................................
    >
```

Chapter 9

The Symbolics 3600 Graphical Interface

CL-PATR makes available a graphical interface to the kinds of functions previously described for the Symbolics 3600 running Genera 7.2 or later software. This section describes this graphical interface. Chapter 10 includes an annotated session with the graphical interface.

9.1 Basic Structure of the Interface

All interaction with CL-PATR is through the CL-PATR frame, a collection of menus, and displays on the screen. The CL-PATR frame can be selected by typing `<select>`—+ (the `<select>` key followed by the plus key). This will create a CL-PATR frame and expose it on the screen if no such frame existed before, or will expose an existing frame.

The frame consists of twelve panes, only eight of which are initially visible. These include, from top to bottom and left to right, the title pane, the command menu, up to eight display panes, an interactor, and a display menu. On startup, only four of the display panes are visible. The upper left pane includes a legal notice.

Interaction with CL-PATR proceeds by typing commands to the interactor (at the '`CL-PATR command:`' prompt) or by clicking the mouse on menu items or mouse-sensitive items in the display panes.

All menu items are invoked by clicking left or right on the item. Most operations on mouse-sensitive items are invoked by clicking middle on the item, although shift-middle (= double-middle) is also used.

Most menu items have corresponding interactor commands that can be typed into the interactor pane instead of clicking on the item.

9.1.1 Display Panes

The frame includes eight display panes in two columns of four panes each, divided conceptually into an upper and a lower bank of four panes. Usually, only one of the two banks is visible. Some operations require the user to choose a pane in which the output of the operations is to be displayed. In such a case, all eight panes will be displayed simultaneously (allowing only the top few lines of each to be easily read. The mouse cursor will change from an arrow to a circle with a cross inside. Clicking the mouse on one of the eight panes selects that pane for output. The bank that the chosen pane is in will be displayed, and the operation will use the chosen pane for output.

The eight display panes are able to scroll both horizontally and vertically using the scroll bars in the left and bottom margins. This allows output that was clipped to be viewed. Another method for viewing clipped output is to configure the frame so that only one of the eight panes occupies the entire area for display panes, four times the usual area. Techniques for reconfiguring the display panes are described below.

9.1.2 Browsing with the Mouse

Many items printed in the display panes are mouse-sensitive; mouse-sensitive items representing rules, lexical entries, chart vertices, edges, and so forth will be displayed in these panes. Just as these items can be browsed using the `Display` command in the portable command interface, the graphical interface allows for browsing using the mouse. By clicking middle on these items, the user is asked to choose a display pane in which to display the expanded information about the chosen item. This information itself may contain mouse-sensitive items that can be further browsed.

9.2 Interface Interaction Functions

In addition to browsing information using the mouse as previously described, the configuration of the graphical interface can itself be modified in various ways. The configuration is controlled through the use of the bottom display menu.

Selecting panes 1 through 4 for single-pane display: The display panes in the upper bank can be individually selected for full screen display by clicking on the `Display 1-4` menu items.

Switching display banks: The `Change display` menu item reconfigures the display panes so that the bank other than the one being currently displayed is made visible. If a single pane is being displayed, the bank that does not include that pane is made visible.

Configuration changes can also be performed through certain mouse actions.

Selecting a pane for single-pane display: Any display pane can be chosen for single-pane display (not just display panes 1 through 4) by clicking middle in an otherwise unoccupied area of the pane.

Deselecting a pane in single-pane display: If a single pane is being displayed, clicking middle in an unoccupied portion of the screen reverts to displaying the bank of panes that includes the single pane.

Clearing a pane: Double-clicking or shift-clicking middle on a pane clears the contents of the pane.

Increasing the interactor pane size: Clicking control-middle anywhere in the frame increases the size of the interactor pane at the bottom of the screen. When in the increased size configuration, clicking control-middle reduces the interactor pane back to its normal size.

9.3 Grammar Manipulation Functions

Listing available files: The `Files` menu item in the top command menu places a list of file pathnames in a chosen pane. (See Section 9.1.1.) The last file to be operated on (i.e., read or installed) is printed in boldface. The pathnames are mouse-sensitive. Clicking on them can cause the files to be read or installed.

This operation can also be invoked by typing the interactor command `Files`.

Reading and installing grammar files: Clicking middle on a pathname causes the file to be read as a grammar file. Clicking shift-middle (= double-middle) on a pathname causes the `.ptro` version of the file to be installed.

The `Read` and `Install` menu items can also be used. After clicking on these commands the user enters a pathname, either by typing in the pathname as a double-quoted string or by clicking left on a pathname being displayed. To complete the command, type a return.

In either case, the selected file will be read or installed, output being sent to a chosen pane.

This operation can also be invoked by typing the interactor commands `Read file` and `Install file`.

These operations correspond to the `Read` and `Install` commands in the portable interface.

Reading data files: A filename with the type `.data` is read differently by the `Read` command and by clicking middle on it. Rather than interpreting the contents of the file as a grammar, the contents are read as LISP objects and listed in a chosen pane. These objects are mouse-sensitive, so that the file can contain sentences to be parsed (as double-quoted strings) or s-expressions to be interpreted as logical forms to generate from, or pathnames of other files to be loaded.

Initializing grammars:

> **Known Bug.**The Initialize menu item should be removed. It is obsolete.

Listing rules, words, macros, and stems: The components of the currently loaded grammar, the rules, words, macros, and stems, can be listed in a chosen pane by invoking the corresponding menu items: `Rules`, `Words`, `Macros`, and `Stems`.

These operations can also be invoked by typing the interactor commands `List rules`, `List words`, `List macros`, and `List stems`. The commands correspond to the `Rules`, `Words`, `Macros`, and `Stems` commands in the portable interface.

The interactor commands `Show rule`, `Show word`, `Show macro`, and `Show stem` prompt for a string and display the specified grammatical item in a chosen pane. The commands correspond to the `Rule`, `Word`, `Macro`, and `Stem` commands in the portable interface. There are no menu equivalents of these commands.

Browsing the grammar: The listed rules, words, macros, and stems are mouse-sensitive. Clicking middle on an item causes its definition to be displayed in a chosen pane.

9.4 Grammar Testing Functions

Parsing sentences: Clicking middle on a string (as displayed, for instance, in a listing of a `.data` file) causes the string to be parsed. Output, including edges corresponding to complete parses and a representation of the chart, is displayed in a chosen pane.

This operation can also be invoked by the command menu item `Parse` or the interactor command `Parse`. The sentence must be entered as a double-quoted string or by clicking left on a displayed string.

Generating bottom up from a logical form: Clicking middle on an s-expression (as displayed, for instance, in a listing of a `.data` file) causes the s-expression to be generated from as a logical form. Output, including edges corresponding to successfully generated strings and a representation of the chart, is displayed in a chosen pane.

The bottom-up generator can also be invoked using the `Generate` command menu item. A logical form is prompted for in the interactor window and can be entered by typing an s-expression or by clicking left on a displayed s-expression (for instance, in the listing of a `.data` file). Results of the generation process are displayed in a chosen pane, including the edges corresponding to successfully generated sentences and a representation of the chart.

This operation can also be invoked with the interactor command `Generate bottom up`.

Generating top-down: The top-down generator can be invoked using the `Generate top down` interactor command. Output is sent to a chosen pane. The user is prompted with menus to guide the generation process, if the choice mode (as specified in the `Parameters` menu) is `interactive`.

9.5 Grammar Debugging Functions

Displaying the chart: The `Chart` command menu item displays in a chosen pane a representation of the chart for the last sentence parsed or generated. The vertices in the representation are mouse-sensitive. A representation of the chart is also printed whenever a sentence is parsed or generated in the same pane as the parsing or generation output.

Browsing the chart: Clicking middle on a vertex in a chart causes the vertex information to be displayed in a chosen pane. The information includes all of the incoming active edges and outgoing passive edges. These are also mouse-sensitive and can themselves be clicked on to display further details.

Displaying parse trees: Clicking shift-middle (= double-middle) on a displayed edge causes the parse tree for that edge to be displayed in a chosen pane.

Extending edges: The command menu item `Extend` allows the user to attempt to extend an active edge in the chart with a passive edge. Results of the attempt are output in a chosen pane. If the attempt is successful, the resulting extended edge is displayed. If the result is unsuccessful, the DG associated with the edge is displayed with an indication of where the unification failed.

9.6 Grammar Modification Functions

Updating a portion of the grammar: The ZMACS editor on the 3600 is given an additional command, bound to the key `<super>-c`, to incrementally compile changes to a loaded grammar. The currently selected region, which should include a series of S-PATR statements, is parsed and the grammar tables are modified or augmented to reflect the changed or additional statements. A grammar ruie replaces a previous definition with the same identifier. Macros and stem definitions replace previous definitions with the same spelling. Word definitions either replace or augment previous definitions with the same spelling; the user is prompted in a pop-up menu for whether the previous definitions should be removed. See the notes on incremental compilation in Chapter 6.

9.7 System Configuration Functions

Displaying and modifying the parameter settings: The `Parameters` menu item pops up a menu of configuration parameters that can be altered. It replaces the `Set` and `Show` commands in the portable command interface.

9.7.1 Configuration Parameters

The parameters menu allows the setting of CL-PATR system parameters. These parameters are, for the most part, identical to those available under the portable command interface by the `Set` command (see Section 7.7.1). The differences are described here.

Set maximum print level for directed graphs. Not included in the parameters menu. Scrolling of windows makes it reasonable to display entire graphs.

Orientation of parse tree printing. Determines whether parse trees are displayed horizontally with the root at the left, or vertically with the root at the top.

> **Known Bug: Missing Parameters.** The verbose edge printing flag and the grammar file pathname list should be added to the parameter list.

9.8 Normal Interaction

The normal method of interacting with the system using the graphical interface is the following. The `Files` command menu item is invoked and the file listing placed in one of the lower bank panes. The lower bank panes are usually reserved for longer term information, as panes in the upper bank are more easily selected.

A pathname in the listing is clicked on to be read or installed, as is a corresponding `.data` file. Again, output is put in the lower bank. Sentences from the data file listing can be parsed, with output including the chart placed in the upper bank. Browsing through the chart and the edges is best done in the upper bank, because of the simplicity of entering single-pane display using the display menu items.

Grammar debugging proceeds by browsing through the chart, attempting to extend edges using the `Extend` command menu item, and so forth.

Chapter 10

Sample Session with the Symbolics 3600 Graphical Interface

This chapter presents snapshots of the graphical interface during a CL-PATR session.

1. The initial appearance of the CL-PATR frame includes a legal notice in the upper-left display pane. Note that in this and future snapshots, the documentation line at the bottom of the screen describes what operations can be performed by clicking the mouse buttons.

   The user clicks shift-middle to clear the pane.

2. The pane that was clicked on (upper left) is now clear. The mouse is over the `Files` command menu item, which appears highlighted with a surrounding rectangle.

   The user clicks left on `Files`.

3. The mouse cursor changes from an arrow to a circle with a cross inside, signalling that the user should select a display pane for output. Note also the documentation line. The frame is now configured to display all eight panes; at the moment, all are empty.

   The user clicks on the lower left pane to place the file listing there.

4. A listing of grammar and data files is placed in the chosen pane and the display is configured to show the lower bank of four display panes.

   The user clicks middle on the pathname for the sample grammar to read it.

5. Again, the user is asked to choose a pane for output. The lower right pane is chosen.

6. The user clicks middle on the corresponding data file pathname to read the sentences and logical forms into a pane to be chosen. (As the pane choice method has been demonstrated, we will skip that portion of the interaction in the future.)

7. The sentences and logical forms are listed in the chosen pane. Note that the pathname is mouse-sensitive.

8. The user clicks on `Initialize` to initialize the grammar in preparation for parsing or generating sentences.

9. The user clicks middle on the sentence "Crockett wants to ventilate criminals" to parse it.

10. After choosing the upper left pane, the parse information is listed.

    The user clicks shift-middle on the chart edge icon corresponding to the single parse of the sentence.

11. A parse tree for the edge is displayed. The nodes are mouse-sensitive and can be clicked on to display information about the corresponding edge.

    The user clicks middle on one of the VP nodes.

12. Information about the edge is printed in a chosen pane. The information does not all fit in the pane. The scroll bars record how much of the information is visible.

    The user clicks middle on a blank area of the pane to change the configuration to show only this pane.

13. Now the display fits.

    The user clicks on the icon for the rule used in constructing the edge.

14. The user chooses a pane in the lower bank in which to display the rule.

15. The rule is displayed.

    The user clicks on `Change display` to switch the display configuration to show the upper bank.

16. The upper bank is displayed.

    The user clicks on `Display 4` to display the lower right pane of the upper bank in single-pane mode. (Note that this would work even if the lower bank were being displayed at the time.)

17. The appropriate pane is displayed.

    The user clicks on the vertex component of the edge icon. (Notice that the edge and certain of its components are separately mouse-sensitive.)

18. Information about the vertex is displayed in the chosen pane. Some of the edges have scrolled off the bottom.

    The user clicks on `Extend` to try to combine some of these edges manually.

19. The user is prompted for an active edge in the interactor window at the bottom of the screen. An active edge is chosen by clicking left.

20. The user is prompted for a passive edge in the interactor window. The user scrolls the display pane to reveal more of the edges.

21. The user selects a passive edge by clicking left, and types the `end` key to complete the command.

22. Information about the attempt to combine these two edges is displayed in a chosen pane. In this case, the attempt failed.

    The user clicks middle on a blank area of the pane to change to single-pane mode.

23. The cause of the failure is listed in the DG for the edge. The active edge required a nonfinite verb phrase complement, but the passive edge was finite.

    The user clicks middle to return to four-pane mode.

24. The user clicks middle on a logical form from which to generate.

25. Information about the generation is displayed in the chosen pane. Only one sentence is generated since the generator was asked only to generate the first sentence. There is only one vertex in the chart for reasons explained in Chapter 12. The user has clicked shift-middle on the icon for the generated sentence and displayed its parse tree in the lower right as well.

    The user clicks on `Parameters` to change some system parameters.

26. The parameters menu is displayed. The user clicks on `Vertical` to change the parse tree display from horizontal trees to vertical trees.

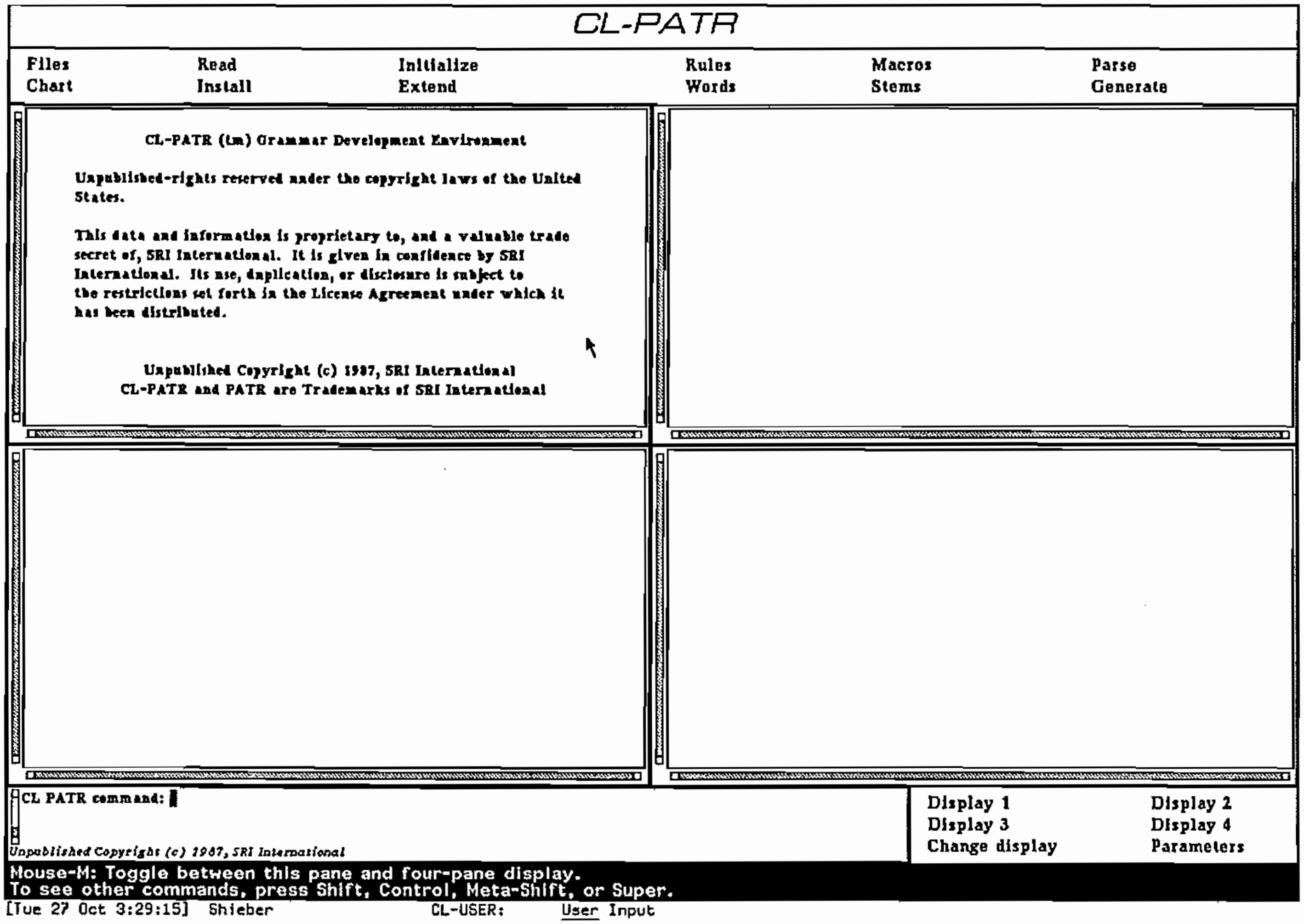
27. The user clicks on `Done` to enable the changes.

28. The user again shift-clicks middle on the edge icon to redisplay the parse tree, now in the lower left, and in vertical format.

    The user clicks on `Words` to list the words in the grammar.

29. A list of words is displayed in a chosen pane. The user selects one of the words by clicking middle.

30. Information about the selected word is displayed in a chosen pane.



CL-PATR (tm) Grammar Development Environment

Unpublished-rights reserved under the copyright laws of the United States.

This data and information is proprietary to, and a valuable trade secret of, SRI International. It is given in confidence by SRI International. Its use, duplication, or disclosure is subject to the restrictions set forth in the License Agreement under which it has been distributed.

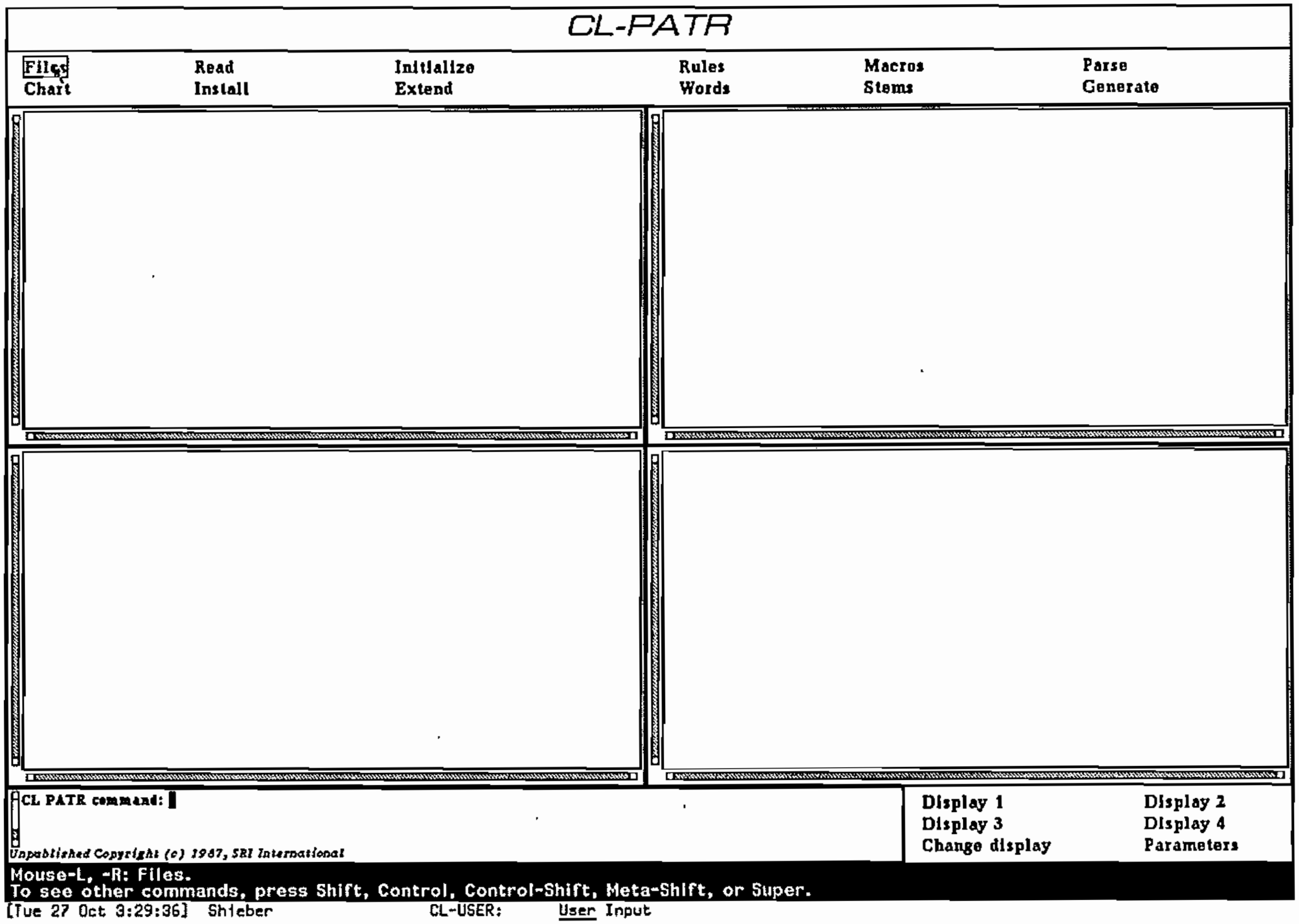
Unpublished Copyright (c) 1987, SRI International
CL-PATR and PATR are Trademarks of SRI International

```
CL PATR command:
```

Unpublished Copyright (c) 1987, SRI International

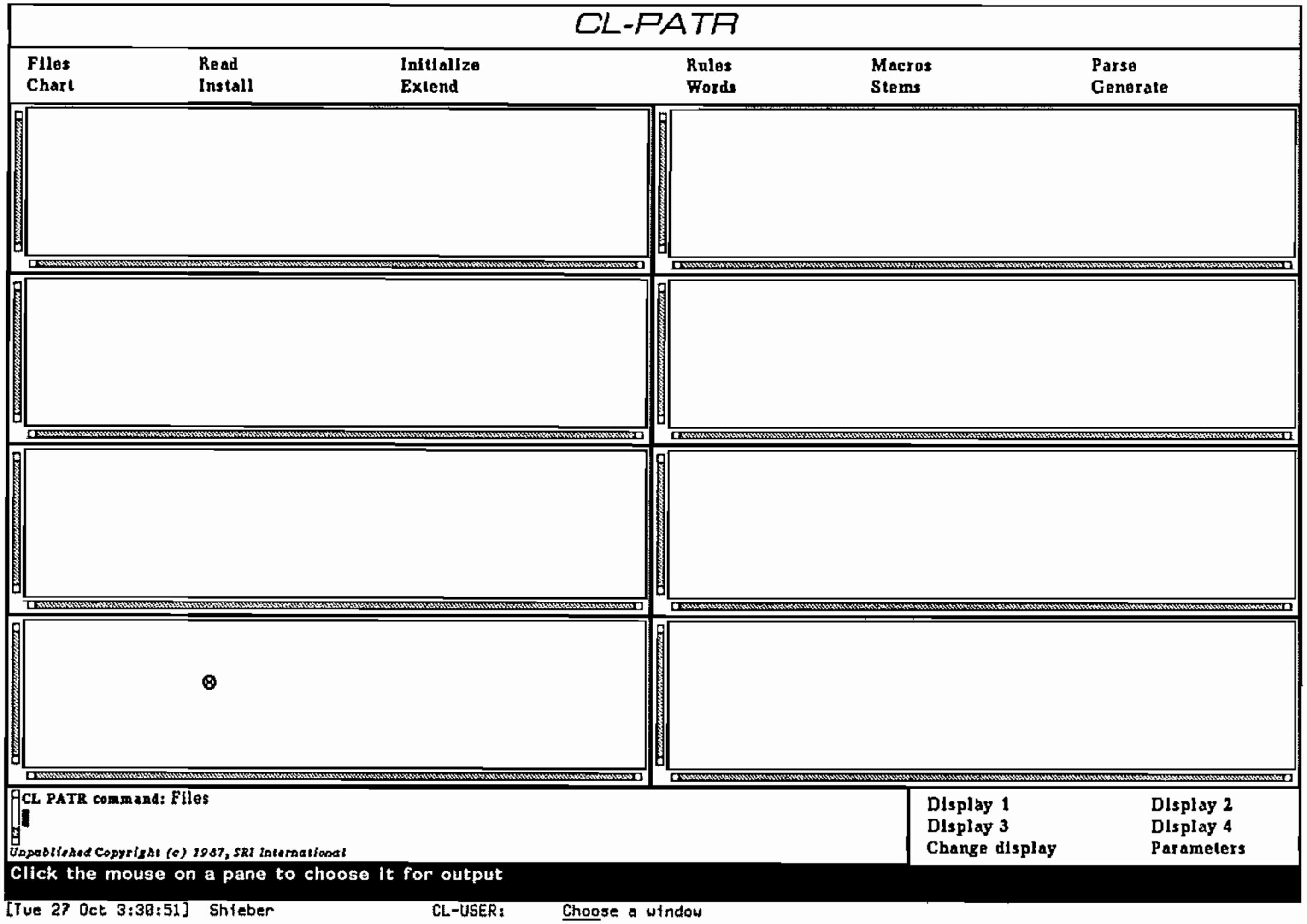
```
Mouse-M: Toggle between this pane and four-pane display.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Tue 27 Oct 3:29:15]  Shieber            CL-USER:      User Input
```

Figure 1



```
CL PATR command: █
Unpublished Copyright (c) 1987, SRI International
Mouse-L, -R: Files.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, or Super.
[Tue 27 Oct 3:29:36]  Shieber            CL-USER:     User Input
```

Figure 2



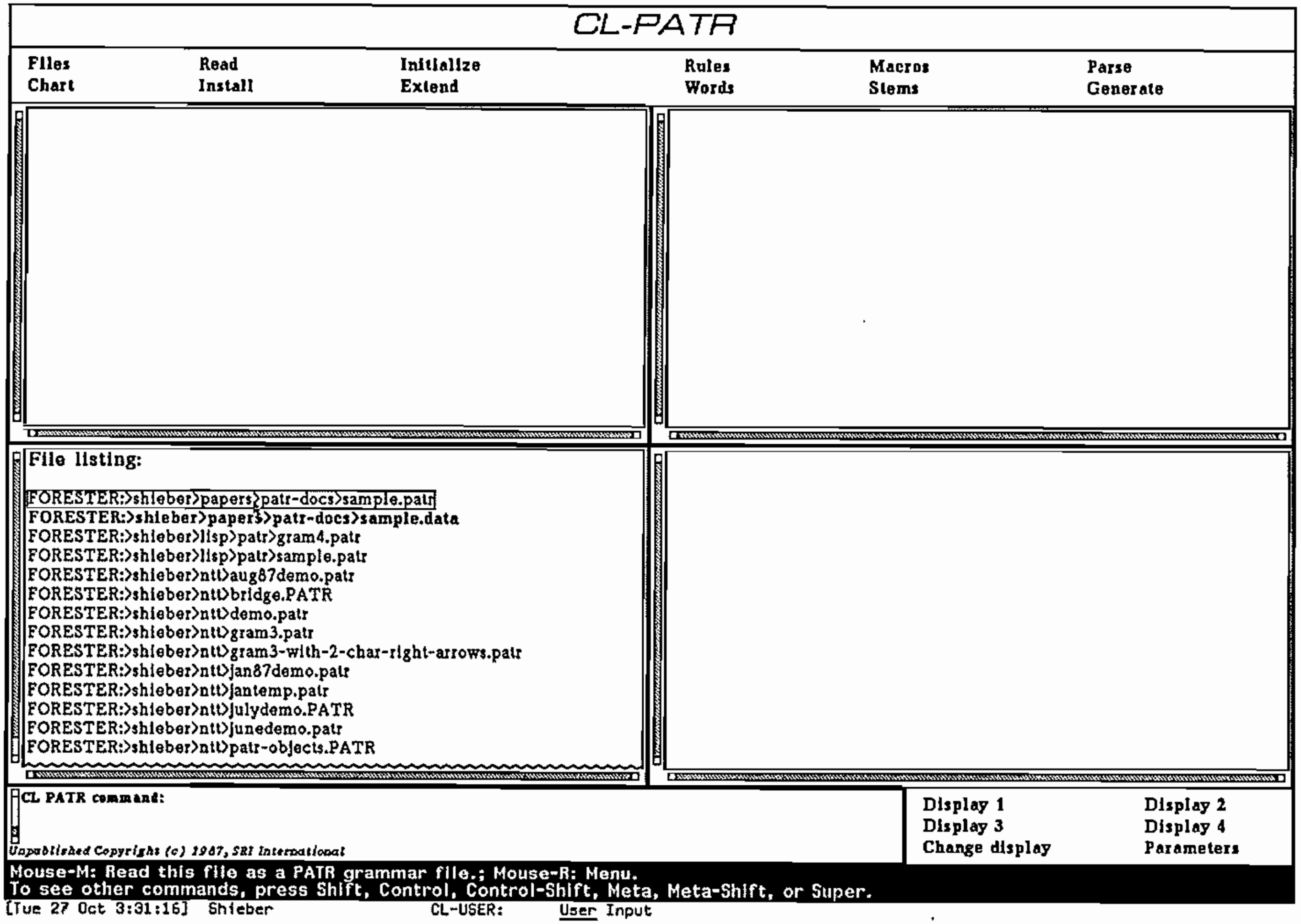
CL PATR command: Files

Unpublished Copyright (c) 1987, SRI International

Click the mouse on a pane to choose it for output

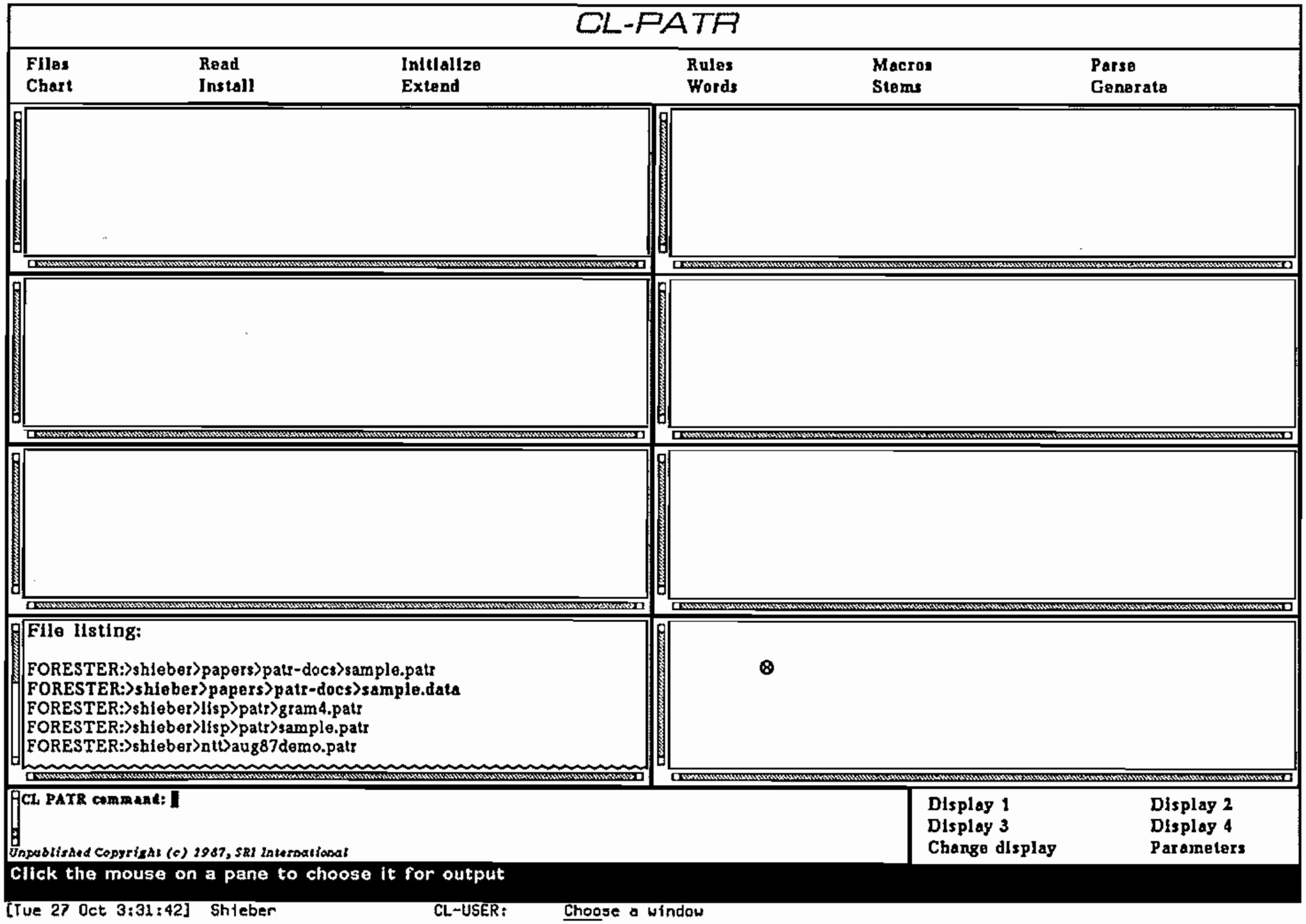
[Tue 27 Oct 3:30:51] Shieber CL-USER: Choose a window

Figure 3



```
File listing:

FORESTER:>shieber>papers>patr-docs>sample.patr
FORESTER:>shieber>papers>patr-docs>sample.data
FORESTER:>shieber>lisp>patr>gram4.patr
FORESTER:>shieber>lisp>patr>sample.patr
FORESTER:>shieber>ntt>aug87demo.patr
FORESTER:>shieber>ntt>bridge.PATR
FORESTER:>shieber>ntt>demo.patr
FORESTER:>shieber>ntt>gram3.patr
FORESTER:>shieber>ntt>gram3-with-2-char-right-arrows.patr
FORESTER:>shieber>ntt>jan87demo.patr
FORESTER:>shieber>ntt>jantemp.patr
FORESTER:>shieber>ntt>julydemo.PATR
FORESTER:>shieber>ntt>junedemo.patr
FORESTER:>shieber>ntt>patr-objects.PATR

CL PATR command:

Unpublished Copyright (c) 1987, SRI International
Mouse-M: Read this file as a PATR grammar file.; Mouse-R: Menu.
To see other commands, press Shift, Control, Control-Shift, Meta, Meta-Shift, or Super.
[Tue 27 Oct 3:31:16]  Shieber            CL-USER:      User Input
```

Figure 4



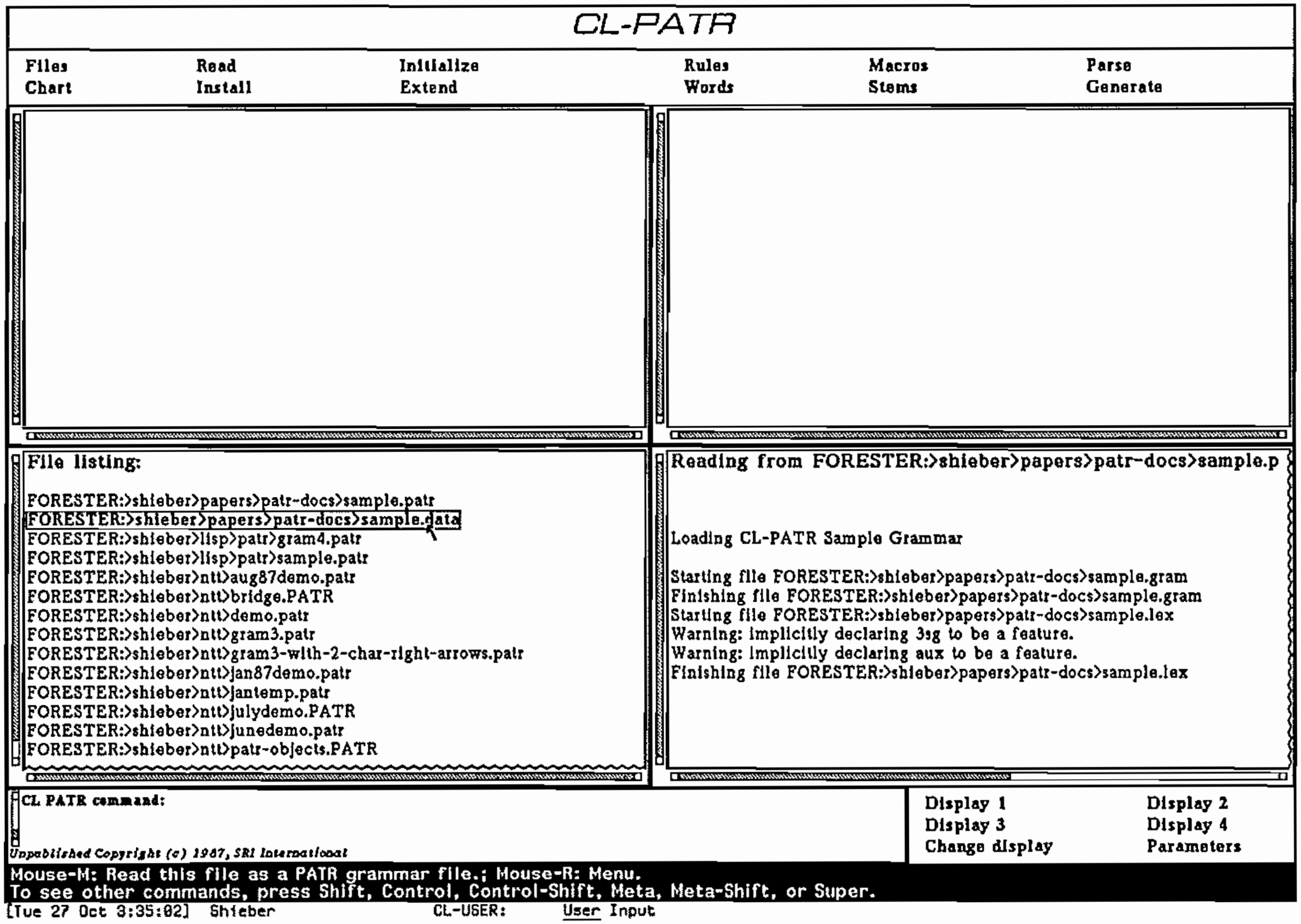
```
File listing:

FORESTER:>shieber>papers>patr-docs>sample.patr
FORESTER:>shieber>papers>patr-docs>sample.data
FORESTER:>shieber>lisp>patr>gram4.patr
FORESTER:>shieber>lisp>patr>sample.patr
FORESTER:>shieber>ntt>aug87demo.patr
```

```
CL PATR command: ▌
```

Unpublished Copyright (c) 1987, SRI International

```
Click the mouse on a pane to choose it for output
[Tue 27 Oct 3:31:42]  Shieber          CL-USER:     Choose a window
```

Figure 5



```
File listing:

FORESTER:>shieber>papers>patr-docs>sample.patr
FORESTER:>shieber>papers>patr-docs>sample.data
FORESTER:>shieber>lisp>patr>gram4.patr
FORESTER:>shieber>lisp>patr>sample.patr
FORESTER:>shieber>ntt>aug87demo.patr
FORESTER:>shieber>ntt>bridge.PATR
FORESTER:>shieber>ntt>demo.patr
FORESTER:>shieber>ntt>gram3.patr
FORESTER:>shieber>ntt>gram3-with-2-char-right-arrows.patr
FORESTER:>shieber>ntt>jan87demo.patr
FORESTER:>shieber>ntt>jantemp.patr
FORESTER:>shieber>ntt>julydemo.PATR
FORESTER:>shieber>ntt>junedemo.patr
FORESTER:>shieber>ntt>patr-objects.PATR
```

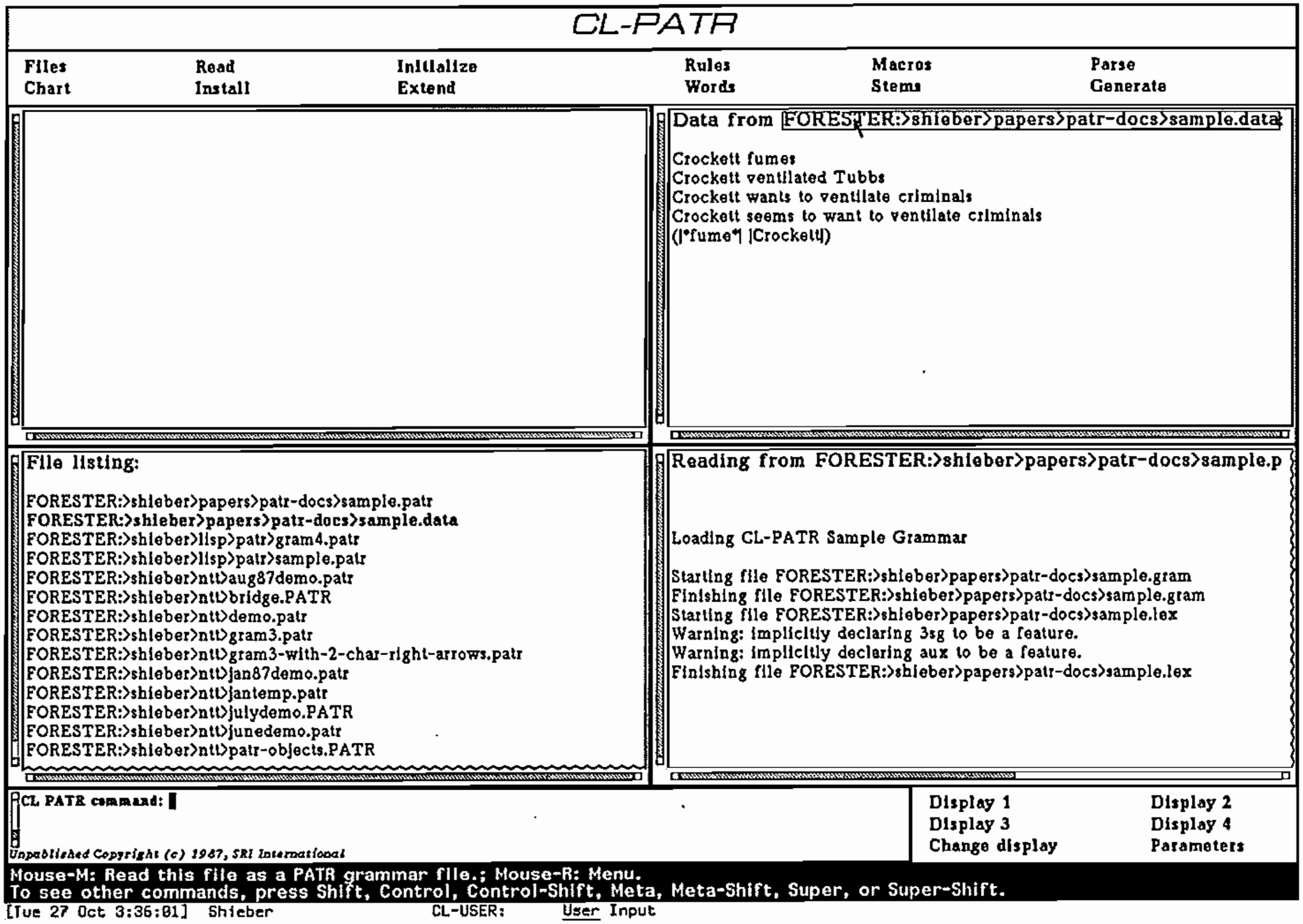
```
Reading from FORESTER:>shieber>papers>patr-docs>sample.p


Loading CL-PATR Sample Grammar

Starting file FORESTER:>shieber>papers>patr-docs>sample.gram
Finishing file FORESTER:>shieber>papers>patr-docs>sample.gram
Starting file FORESTER:>shieber>papers>patr-docs>sample.lex
Warning: implicitly declaring 3sg to be a feature.
Warning: implicitly declaring aux to be a feature.
Finishing file FORESTER:>shieber>papers>patr-docs>sample.lex
```

```
CL PATR command:
```

Unpublished Copyright (c) 1987, SRI International

```
Mouse-M: Read this file as a PATR grammar file.; Mouse-R: Menu.
To see other commands, press Shift, Control, Control-Shift, Meta, Meta-Shift, or Super.
[Tue 27 Oct 3:35:02]  Shieber            CL-USER:      User Input
```

Figure 6



```
Data from FORESTER:>shieber>papers>patr-docs>sample.data

Crockett fumes
Crockett ventilated Tubbs
Crockett wants to ventilate criminals
Crockett seems to want to ventilate criminals
(|*fume*| |Crockett|)
```

```
File listing:

FORESTER:>shieber>papers>patr-docs>sample.patr
FORESTER:>shieber>papers>patr-docs>sample.data
FORESTER:>shieber>lisp>patr>gram4.patr
FORESTER:>shieber>lisp>patr>sample.patr
FORESTER:>shieber>ntt>aug87demo.patr
FORESTER:>shieber>ntt>bridge.PATR
FORESTER:>shieber>ntt>demo.patr
FORESTER:>shieber>ntt>gram3.patr
FORESTER:>shieber>ntt>gram3-with-2-char-right-arrows.patr
FORESTER:>shieber>ntt>jan87demo.patr
FORESTER:>shieber>ntt>jantemp.patr
FORESTER:>shieber>ntt>julydemo.PATR
FORESTER:>shieber>ntt>junedemo.patr
FORESTER:>shieber>ntt>patr-objects.PATR
```

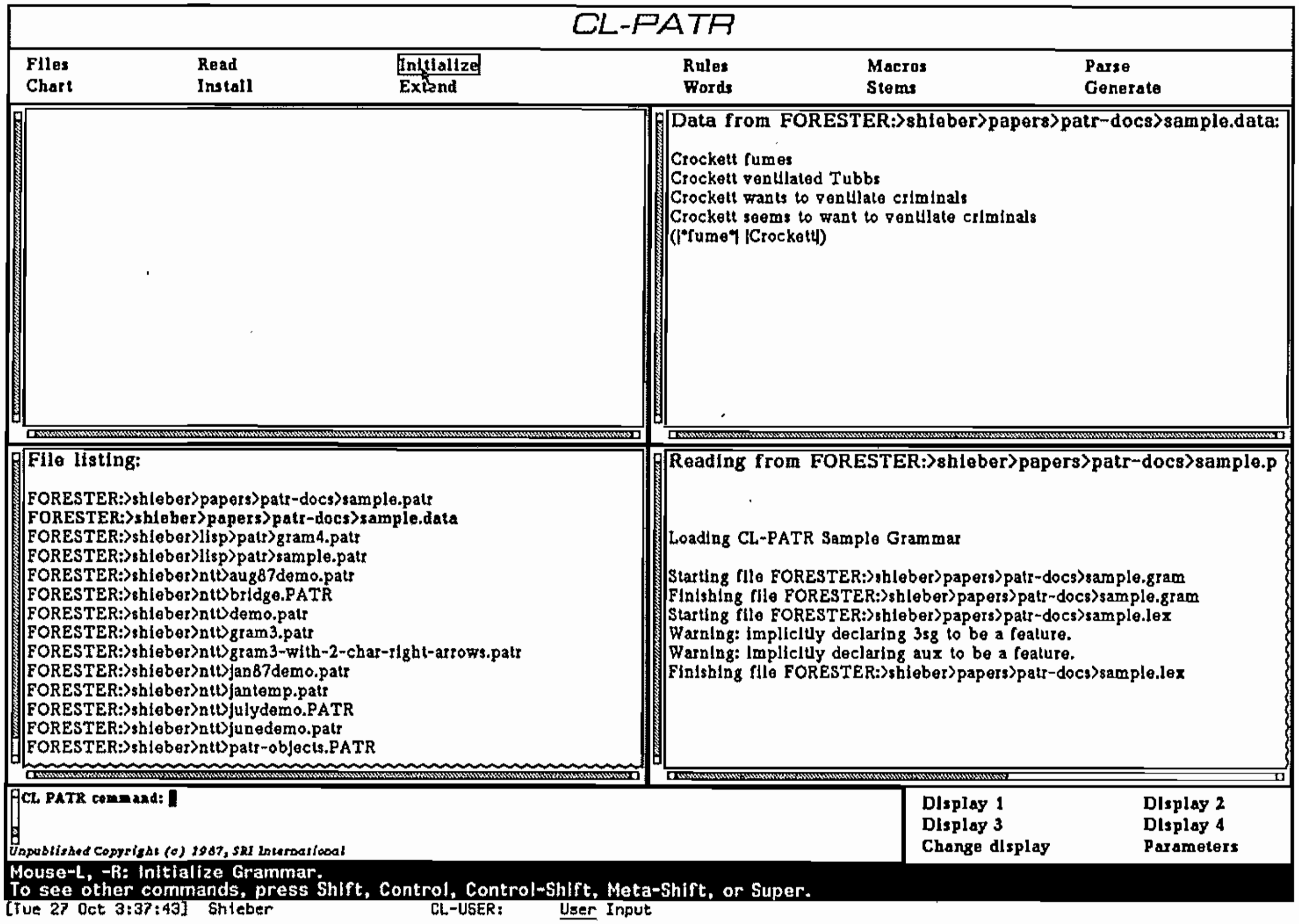
```
Reading from FORESTER:>shieber>papers>patr-docs>sample.p


Loading CL-PATR Sample Grammar

Starting file FORESTER:>shieber>papers>patr-docs>sample.gram
Finishing file FORESTER:>shieber>papers>patr-docs>sample.gram
Starting file FORESTER:>shieber>papers>patr-docs>sample.lex
Warning: implicitly declaring 3sg to be a feature.
Warning: implicitly declaring aux to be a feature.
Finishing file FORESTER:>shieber>papers>patr-docs>sample.lex
```

```
CL PATR command:
```

Unpublished Copyright (c) 1987, SRI International

```
Mouse-M: Read this file as a PATR grammar file.; Mouse-R: Menu.
To see other commands, press Shift, Control, Control-Shift, Meta, Meta-Shift, Super, or Super-Shift.
[Tue 27 Oct 3:36:01]  Shieber          CL-USER:     User Input
```

Figure 7



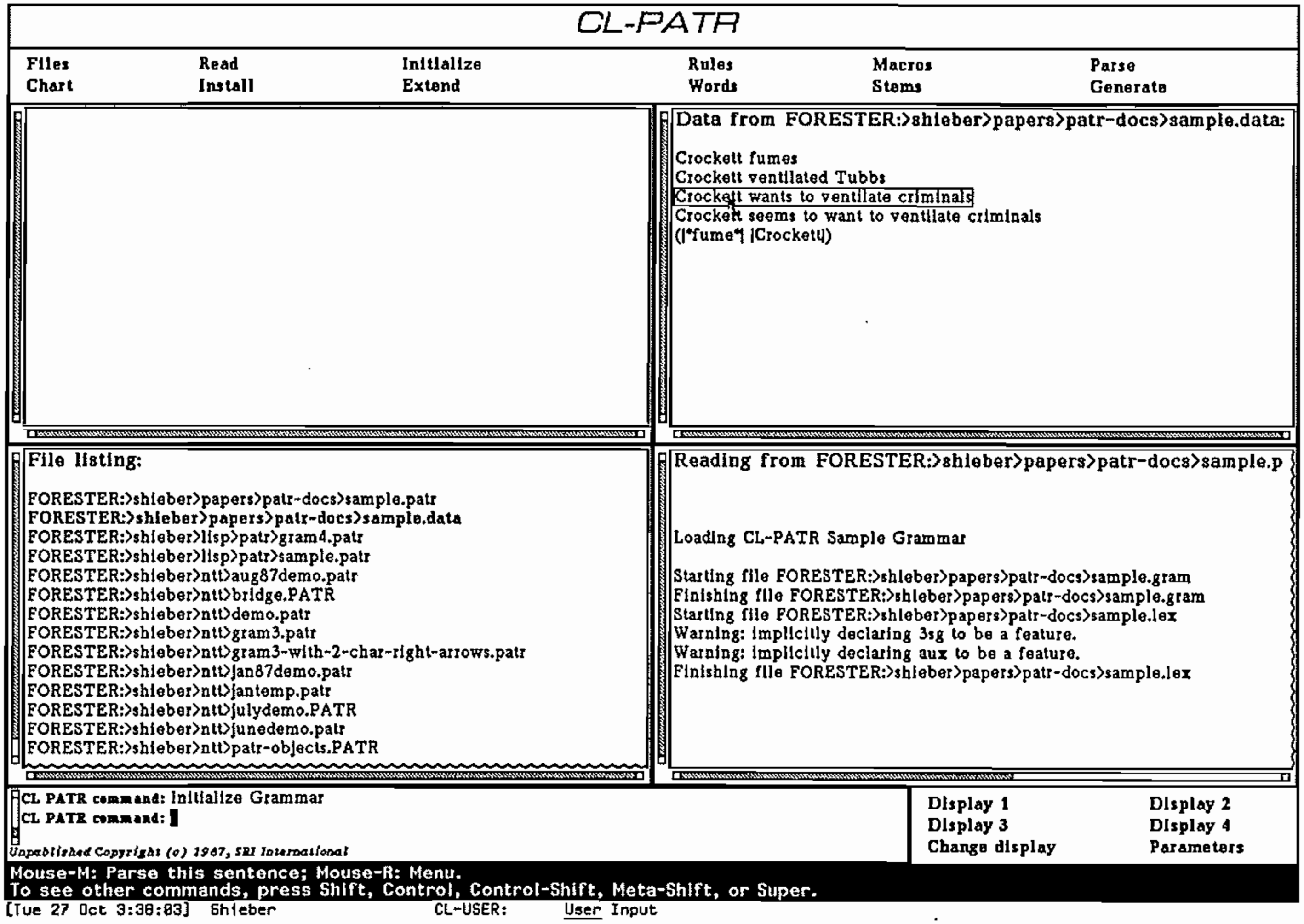
```
Data from FORESTER:>shieber>papers>patr-docs>sample.data:

Crockett fumes
Crockett ventilated Tubbs
Crockett wants to ventilate criminals
Crockett seems to want to ventilate criminals
(|*fume*| |Crockett|)
```

```
File listing:

FORESTER:>shieber>papers>patr-docs>sample.patr
FORESTER:>shieber>papers>patr-docs>sample.data
FORESTER:>shieber>lisp>patr>gram4.patr
FORESTER:>shieber>lisp>patr>sample.patr
FORESTER:>shieber>ntt>aug87demo.patr
FORESTER:>shieber>ntt>bridge.PATR
FORESTER:>shieber>ntt>demo.patr
FORESTER:>shieber>ntt>gram3.patr
FORESTER:>shieber>ntt>gram3-with-2-char-right-arrows.patr
FORESTER:>shieber>ntt>jan87demo.patr
FORESTER:>shieber>ntt>jantemp.patr
FORESTER:>shieber>ntt>julydemo.PATR
FORESTER:>shieber>ntt>junedemo.patr
FORESTER:>shieber>ntt>patr-objects.PATR
```

```
Reading from FORESTER:>shieber>papers>patr-docs>sample.p


Loading CL-PATR Sample Grammar

Starting file FORESTER:>shieber>papers>patr-docs>sample.gram
Finishing file FORESTER:>shieber>papers>patr-docs>sample.gram
Starting file FORESTER:>shieber>papers>patr-docs>sample.lex
Warning: implicitly declaring 3sg to be a feature.
Warning: implicitly declaring aux to be a feature.
Finishing file FORESTER:>shieber>papers>patr-docs>sample.lex
```

```
CL PATR command:
Unpublished Copyright (c) 1987, SRI International
Mouse-L, -R: Initialize Grammar.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, or Super.
[Tue 27 Oct 3:37:43]  Shieber          CL-USER:     User Input
```

Figure 8



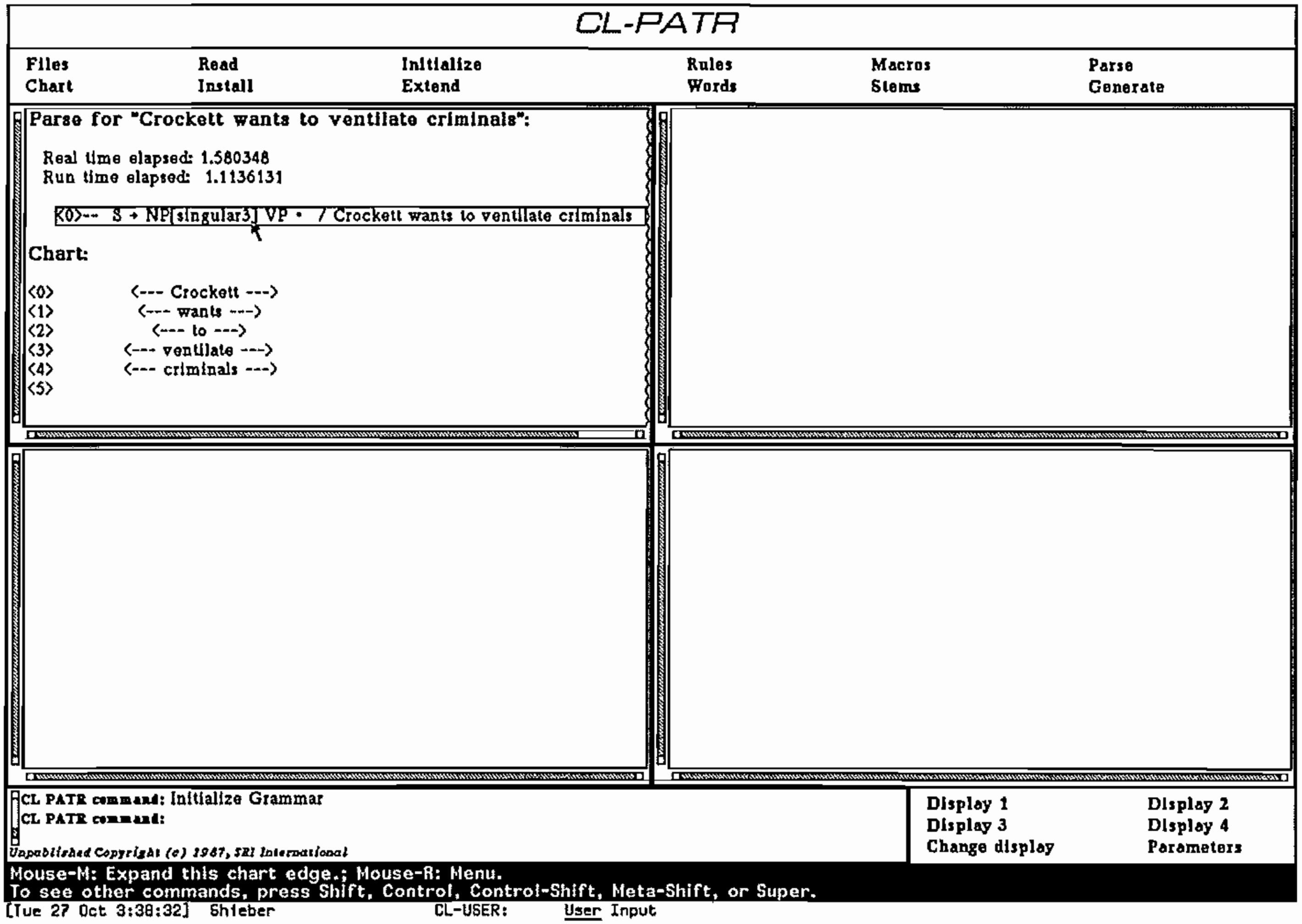
```
Data from FORESTER:>shieber>papers>patr-docs>sample.data:

Crockett fumes
Crockett ventilated Tubbs
Crockett wants to ventilate criminals
Crockett seems to want to ventilate criminals
(|*fume*| |Crockett|)
```

```
File listing:

FORESTER:>shieber>papers>patr-docs>sample.patr
FORESTER:>shieber>papers>patr-docs>sample.data
FORESTER:>shieber>lisp>patr>gram4.patr
FORESTER:>shieber>lisp>patr>sample.patr
FORESTER:>shieber>ntt>aug87demo.patr
FORESTER:>shieber>ntt>bridge.PATR
FORESTER:>shieber>ntt>demo.patr
FORESTER:>shieber>ntt>gram3.patr
FORESTER:>shieber>ntt>gram3-with-2-char-right-arrows.patr
FORESTER:>shieber>ntt>jan87demo.patr
FORESTER:>shieber>ntt>jantemp.patr
FORESTER:>shieber>ntt>julydemo.PATR
FORESTER:>shieber>ntt>junedemo.patr
FORESTER:>shieber>ntt>patr-objects.PATR
```

```
Reading from FORESTER:>shieber>papers>patr-docs>sample.p


Loading CL-PATR Sample Grammar

Starting file FORESTER:>shieber>papers>patr-docs>sample.gram
Finishing file FORESTER:>shieber>papers>patr-docs>sample.gram
Starting file FORESTER:>shieber>papers>patr-docs>sample.lex
Warning: implicitly declaring 3sg to be a feature.
Warning: implicitly declaring aux to be a feature.
Finishing file FORESTER:>shieber>papers>patr-docs>sample.lex
```

```
CL PATR command: Initialize Grammar
CL PATR command: █
Unpublished Copyright (c) 1987, SRI International
Mouse-M: Parse this sentence; Mouse-R: Menu.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, or Super.
[Tue 27 Oct 3:38:03]  Shieber            CL-USER:     User Input
```

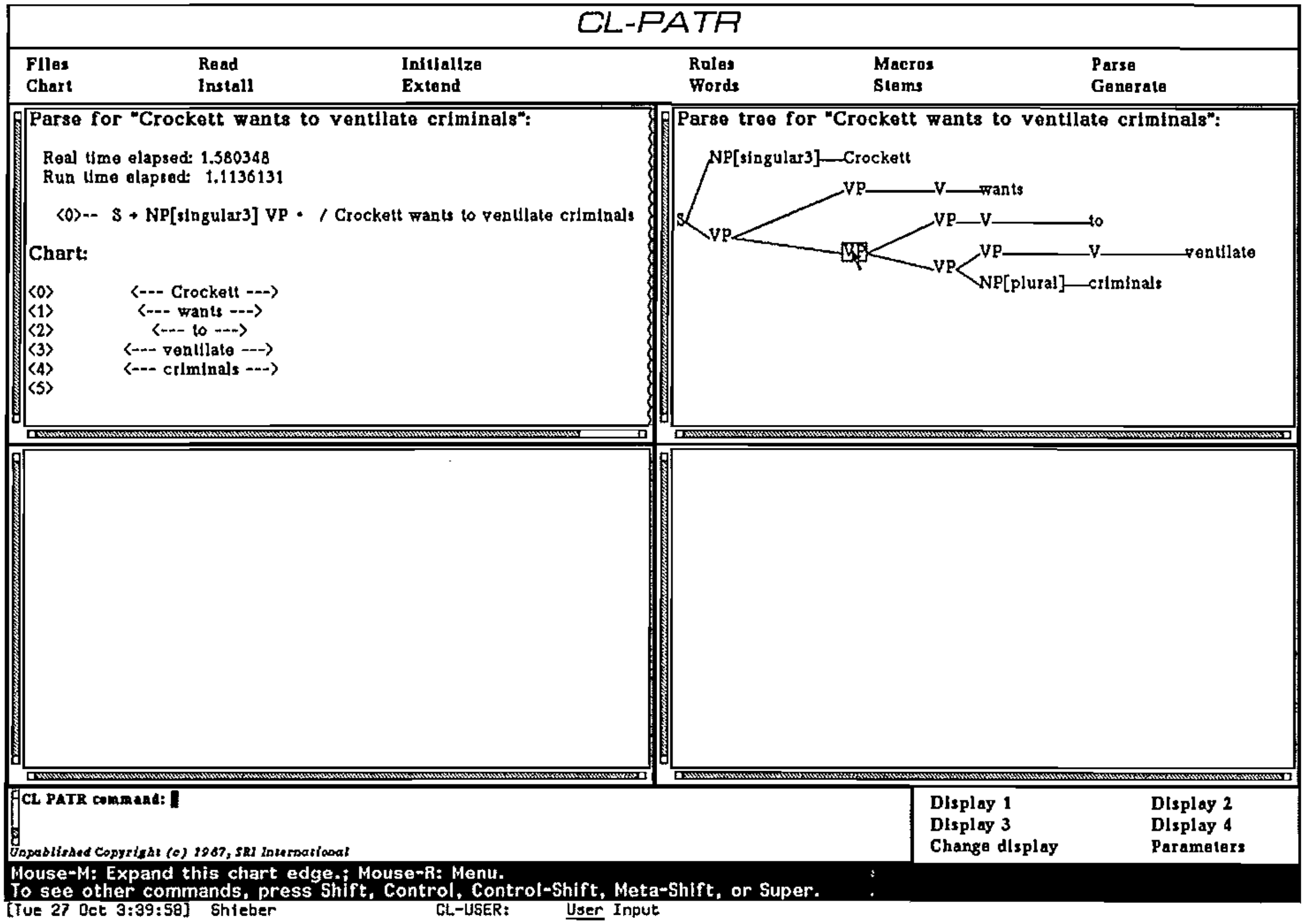
Figure 9



```
Parse for "Crockett wants to ventilate criminals":

  Real time elapsed: 1.580348
  Run time elapsed:  1.1136131

  <0>--  S → NP[singular3] VP •  / Crockett wants to ventilate criminals

Chart:

<0>        <--- Crockett --->
<1>         <--- wants --->
<2>           <--- to --->
<3>       <--- ventilate --->
<4>       <--- criminals --->
<5>
```

```
CL PATR command: Initialize Grammar
CL PATR command:
Unpublished Copyright (c) 1987, SRI International
Mouse-M: Expand this chart edge.; Mouse-R: Menu.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, or Super.
[Tue 27 Oct 3:38:32]  Shieber              CL-USER:       User Input
```

Figure 10

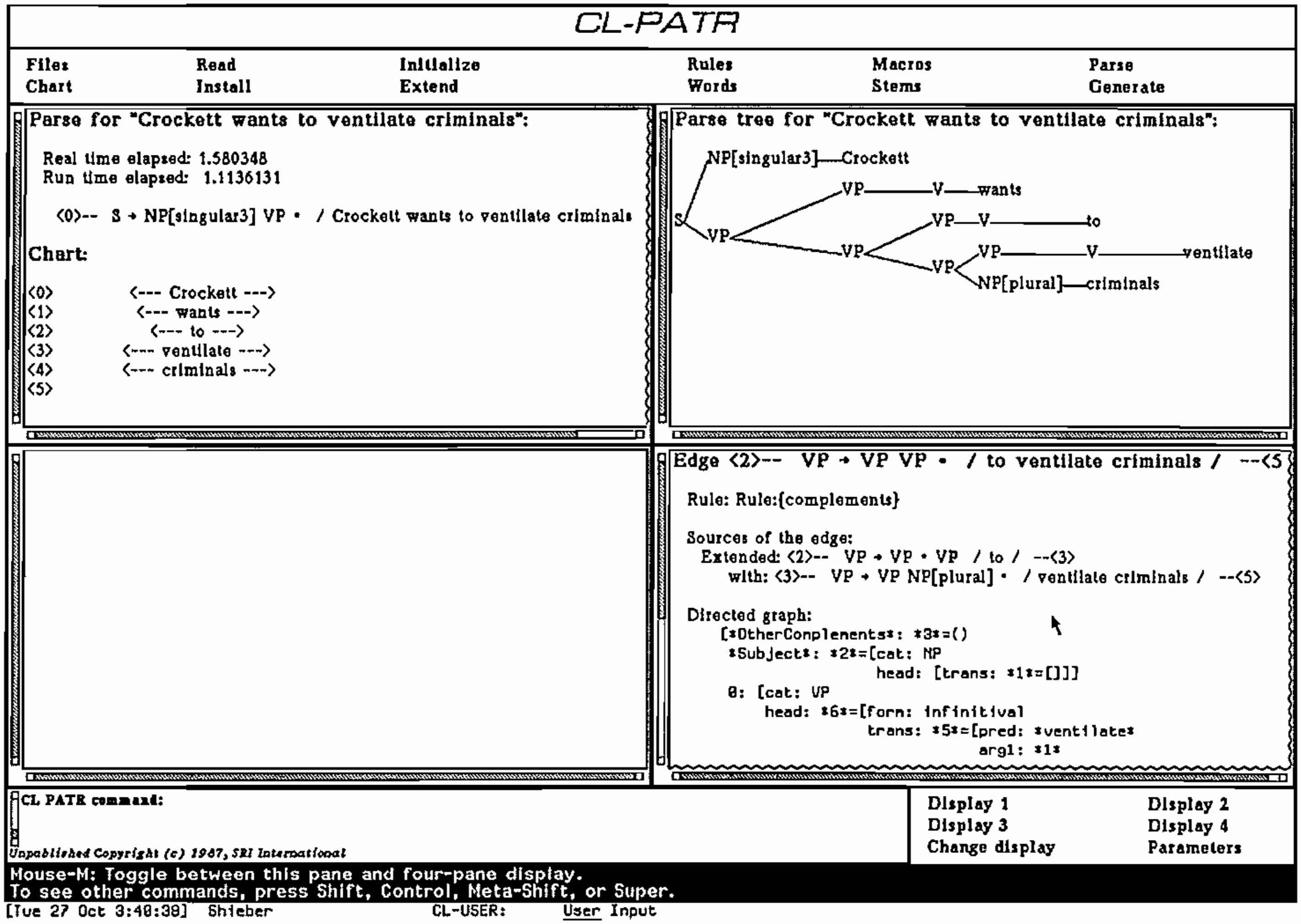


```
Parse for "Crockett wants to ventilate criminals":

  Real time elapsed: 1.580348
  Run time elapsed:  1.1136131

   <0>--  S → NP[singular3] VP •  / Crockett wants to ventilate criminals

Chart:

<0>         <--- Crockett --->
<1>          <--- wants --->
<2>            <--- to --->
<3>         <--- ventilate --->
<4>         <--- criminals --->
<5>
```

```
Parse tree for "Crockett wants to ventilate criminals":
```

```
CL PATR command: ▌
Unpublished Copyright (c) 1987, SRI International
Mouse-M: Expand this chart edge.; Mouse-R: Menu.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, or Super.
[Tue 27 Oct 3:39:58]  Shieber              CL-USER:      User Input
```

Figure 11



```
Parse for "Crockett wants to ventilate criminals":

  Real time elapsed: 1.580348
  Run time elapsed:  1.1136131

    <0>--  S → NP[singular3] VP •  / Crockett wants to ventilate criminals

Chart:

<0>          <--- Crockett --->
<1>           <--- wants --->
<2>             <--- to --->
<3>         <--- ventilate --->
<4>         <--- criminals --->
<5>
```

Parse tree for "Crockett wants to ventilate criminals":

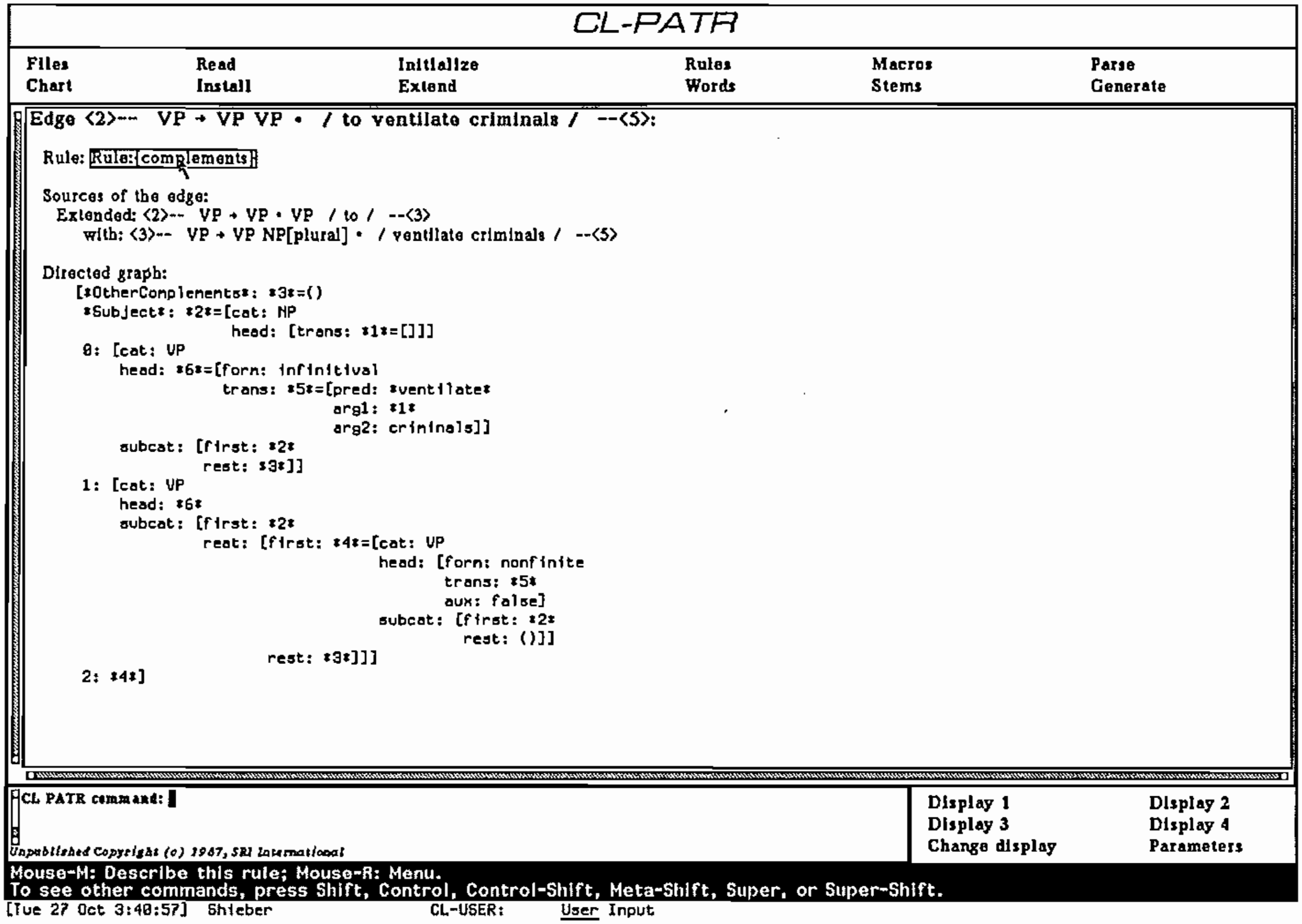
```
Edge <2>--  VP → VP VP •  / to ventilate criminals /  --<5

  Rule: Rule:{complements}

  Sources of the edge:
    Extended: <2>--  VP → VP • VP  / to /  --<3>
        with: <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>

  Directed graph:
      [*OtherConplenents*: *3*=()
       *Subject*: *2*=[cat: NP
                       head: [trans: *1*=[]]]
        0: [cat: VP
            head: *6*=[forn: infinitival
                       trans: *5*=[pred: *ventilate*
                                   arg1: *1*
```

```
CL PATR command:
```

Unpublished Copyright (c) 1987, SRI International

```
Mouse-M: Toggle between this pane and four-pane display.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Tue 27 Oct 3:40:30]  Shieber            CL-USER:     User Input
```

Figure 12



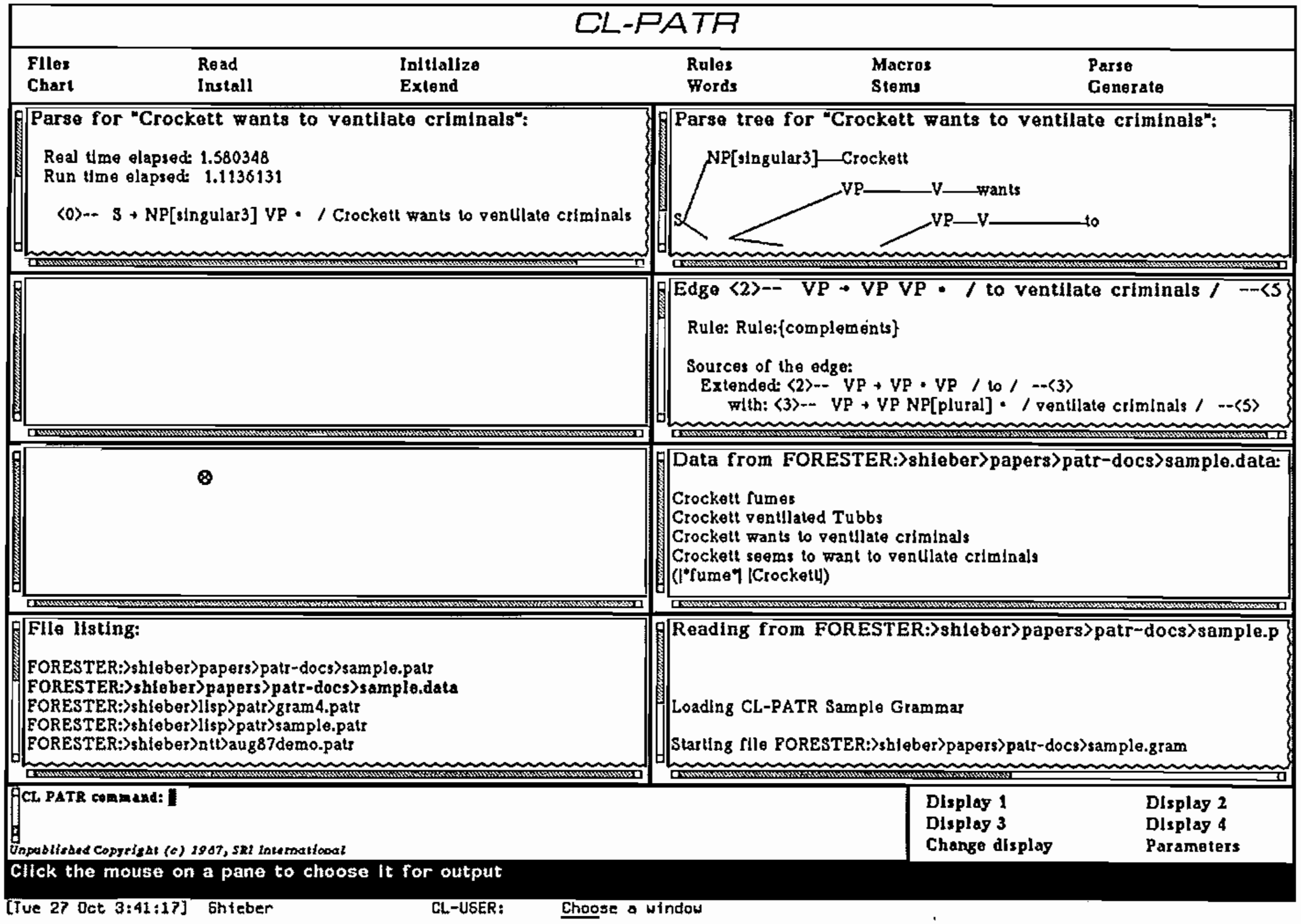
```
Edge <2>--  VP → VP VP •  / to ventilate criminals /  --<5>:

 Rule: Rule:{complements}

 Sources of the edge:
  Extended: <2>--  VP → VP • VP  / to /  --<3>
     with: <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>

 Directed graph:
    [*OtherComplements*: *3*=()
     *Subject*: *2*=[cat: NP
                     head: [trans: *1*=[]]]
     0: [cat: VP
         head: *6*=[form: infinitival
                    trans: *5*=[pred: *ventilate*
                                arg1: *1*
                                arg2: criminals]]
         subcat: [first: *2*
                  rest: *3*]]
     1: [cat: VP
         head: *6*
         subcat: [first: *2*
                  rest: [first: *4*=[cat: VP
                                     head: [form: nonfinite
                                            trans: *5*
                                            aux: false]
                                     subcat: [first: *2*
                                              rest: ()]]
                         rest: *3*]]]
     2: *4*]
```

```
CL PATR command: █
```

Unpublished Copyright (c) 1987, SRI International

```
Mouse-M: Describe this rule; Mouse-R: Menu.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, Super, or Super-Shift.
[Tue 27 Oct 3:40:57]  Shieber                  CL-USER:     User Input
```

Figure 13



```
Parse for "Crockett wants to ventilate criminals":

  Real time elapsed: 1.580348
  Run time elapsed:  1.1136131

   <0>-- S → NP[singular3] VP •  / Crockett wants to ventilate criminals
```

```
Parse tree for "Crockett wants to ventilate criminals":
```

```
Edge <2>--  VP → VP VP •  / to ventilate criminals /  --<5

  Rule: Rule:{complements}

  Sources of the edge:
    Extended: <2>-- VP → VP • VP  / to /  --<3>
       with: <3>-- VP → VP NP[plural] •  / ventilate criminals /  --<5>
```

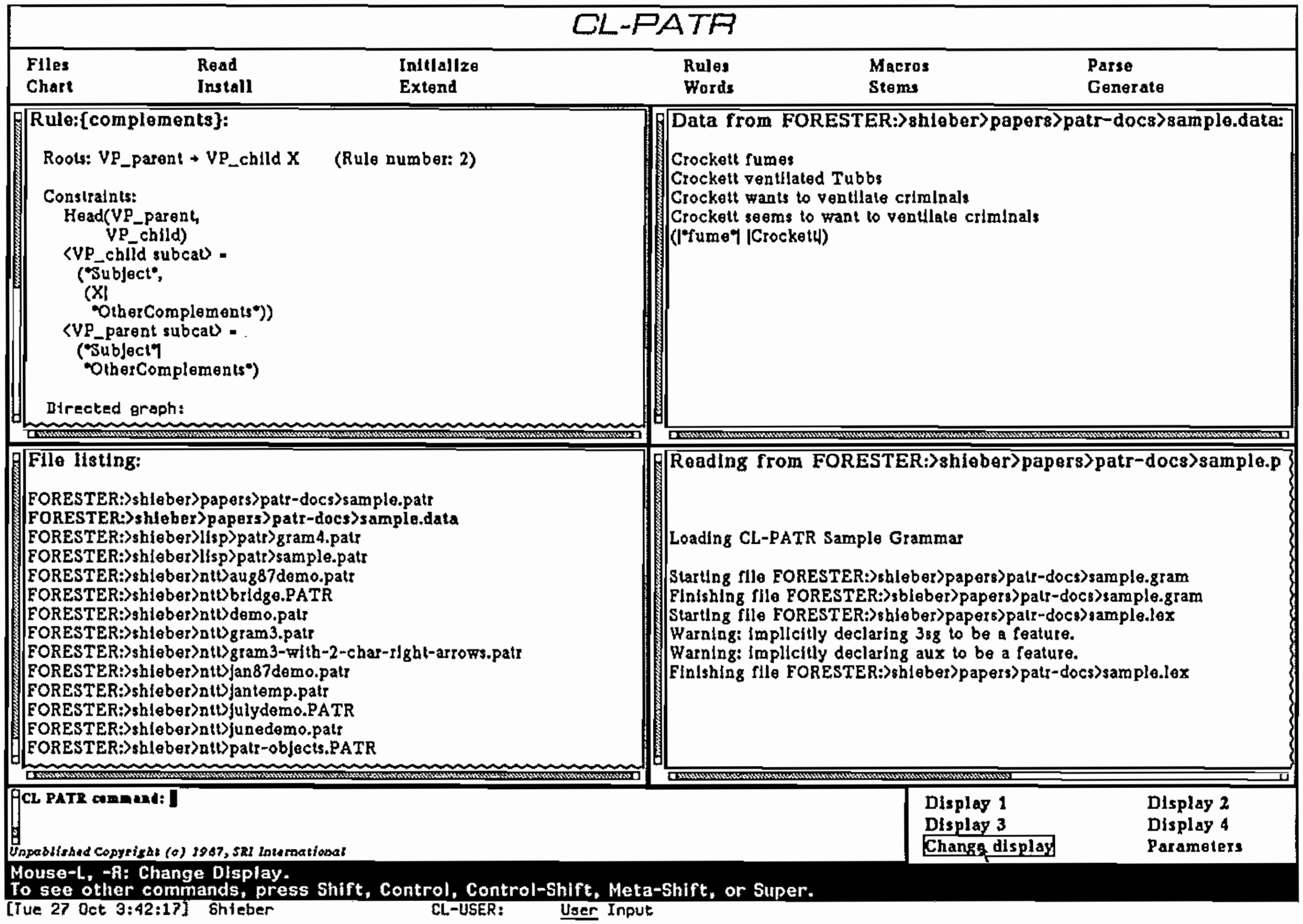
```
Data from FORESTER:>shieber>papers>patr-docs>sample.data:

Crockett fumes
Crockett ventilated Tubbs
Crockett wants to ventilate criminals
Crockett seems to want to ventilate criminals
(|*fume*| |Crockett|)
```

```
File listing:

FORESTER:>shieber>papers>patr-docs>sample.patr
FORESTER:>shieber>papers>patr-docs>sample.data
FORESTER:>shieber>lisp>patr>gram4.patr
FORESTER:>shieber>lisp>patr>sample.patr
FORESTER:>shieber>ntt>aug87demo.patr
```

```
Reading from FORESTER:>shieber>papers>patr-docs>sample.p


Loading CL-PATR Sample Grammar

Starting file FORESTER:>shieber>papers>patr-docs>sample.gram
```

```
CL PATR command:
Unpublished Copyright (c) 1987, SRI International
Click the mouse on a pane to choose it for output
[Tue 27 Oct 3:41:17]  Shieber          CL-USER:      Choose a window
```

Figure 14



```
Rule:{complements}:

  Roots: VP_parent → VP_child X     (Rule number: 2)

  Constraints:
     Head(VP_parent,
           VP_child)
    <VP_child subcat> =
      (*Subject*,
       (X|
        *OtherComplements*))
    <VP_parent subcat> =
      (*Subject*|
       *OtherComplements*)

  Directed graph:
```

```
Data from FORESTER:>shieber>papers>patr-docs>sample.data:

Crockett fumes
Crockett ventilated Tubbs
Crockett wants to ventilate criminals
Crockett seems to want to ventilate criminals
(|*fume*| |Crockett|)
```

```
File listing:

FORESTER:>shieber>papers>patr-docs>sample.patr
FORESTER:>shieber>papers>patr-docs>sample.data
FORESTER:>shieber>lisp>patr>gram4.patr
FORESTER:>shieber>lisp>patr>sample.patr
FORESTER:>shieber>ntl>aug87demo.patr
FORESTER:>shieber>ntl>bridge.PATR
FORESTER:>shieber>ntl>demo.patr
FORESTER:>shieber>ntl>gram3.patr
FORESTER:>shieber>ntl>gram3-with-2-char-right-arrows.patr
FORESTER:>shieber>ntl>jan87demo.patr
FORESTER:>shieber>ntl>jantemp.patr
FORESTER:>shieber>ntl>julydemo.PATR
FORESTER:>shieber>ntl>junedemo.patr
FORESTER:>shieber>ntl>patr-objects.PATR
```

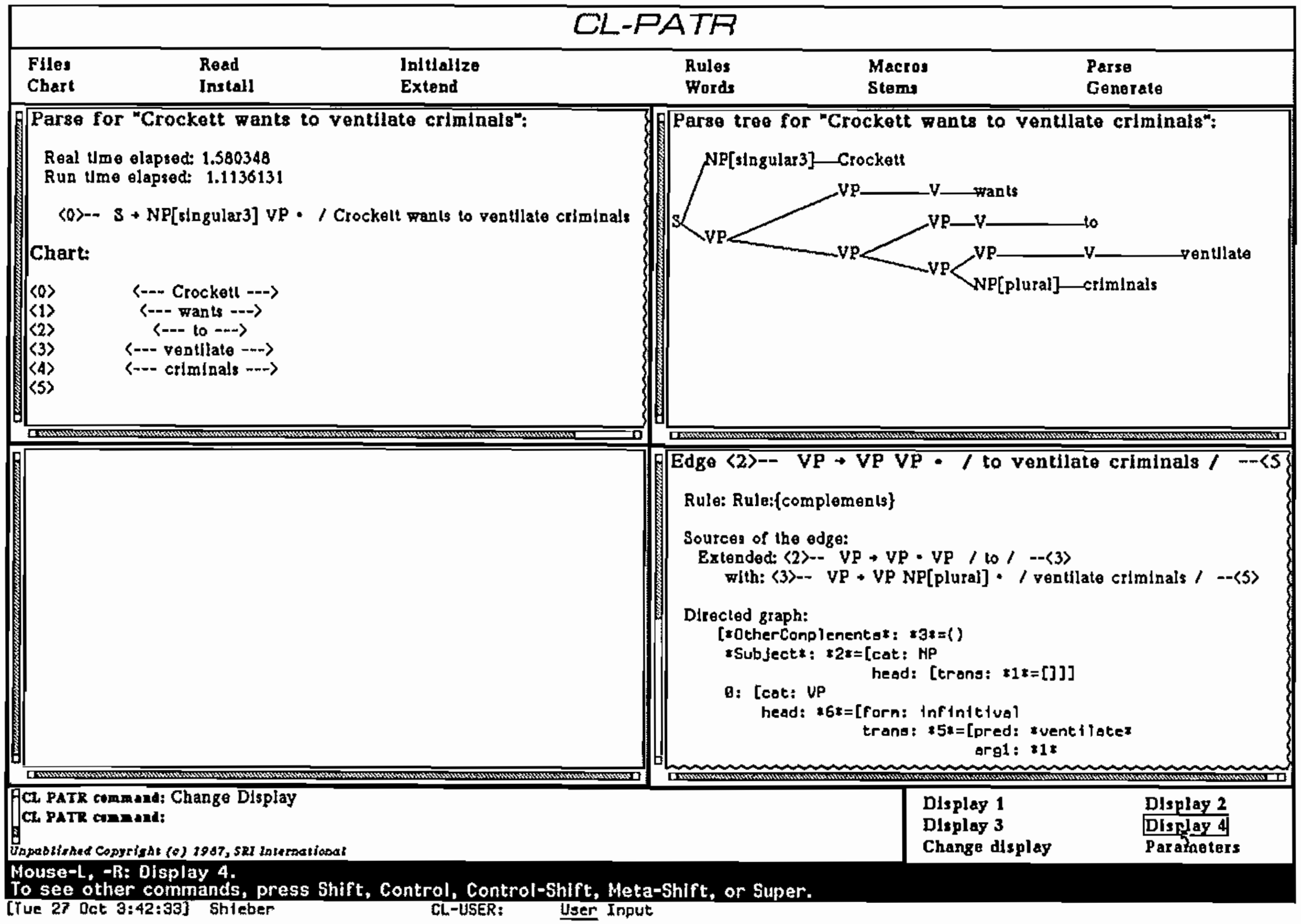
```
Reading from FORESTER:>shieber>papers>patr-docs>sample.p


Loading CL-PATR Sample Grammar

Starting file FORESTER:>shieber>papers>patr-docs>sample.gram
Finishing file FORESTER:>shieber>papers>patr-docs>sample.gram
Starting file FORESTER:>shieber>papers>patr-docs>sample.lex
Warning: implicitly declaring 3sg to be a feature.
Warning: implicitly declaring aux to be a feature.
Finishing file FORESTER:>shieber>papers>patr-docs>sample.lex
```

```
CL PATR command: █
```

Unpublished Copyright (c) 1987, SRI International

```
Mouse-L, -R: Change Display.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, or Super.
[Tue 27 Oct 3:42:17]  Shieber            CL-USER:      User Input
```

Figure 15



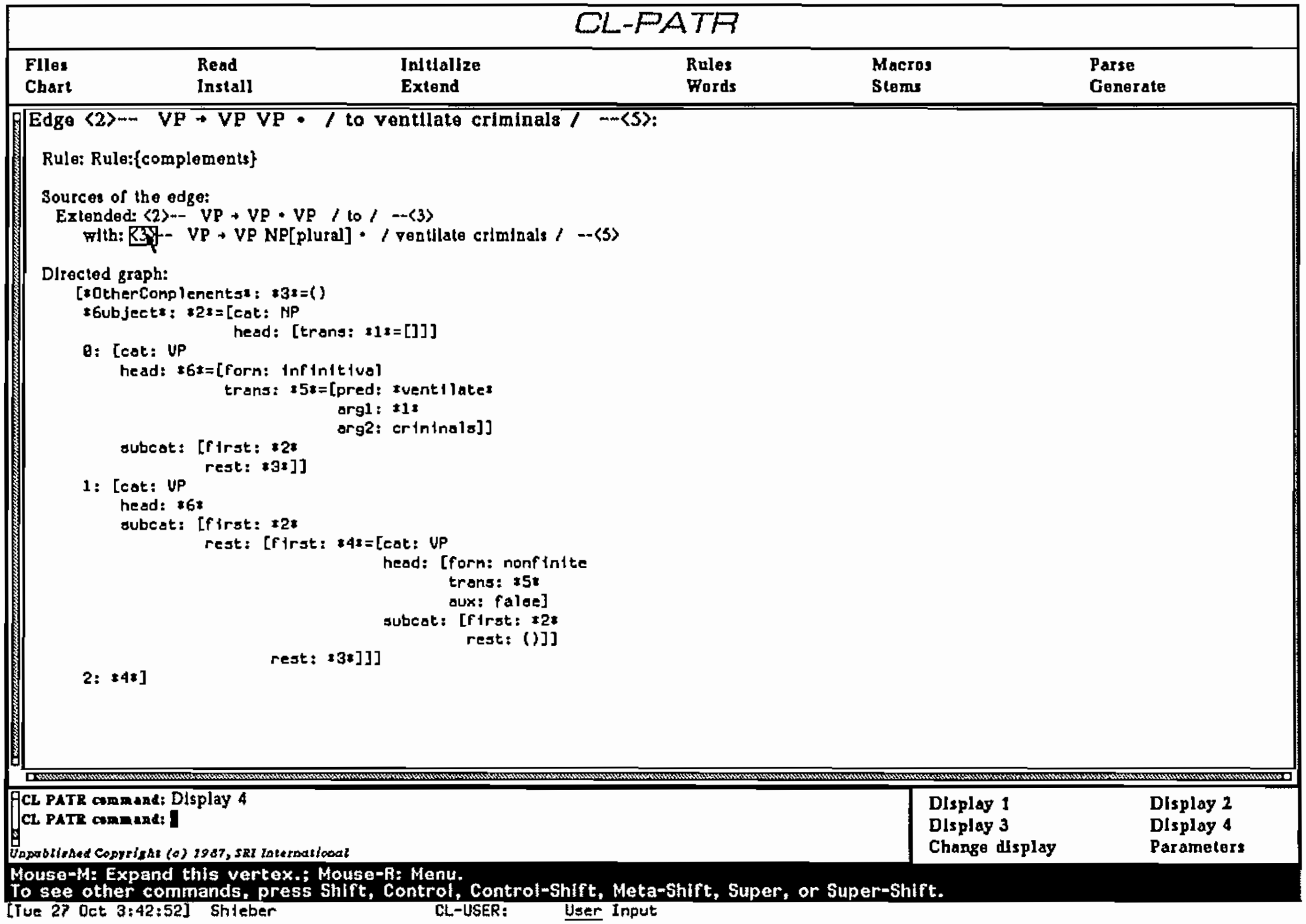
```
Parse for "Crockett wants to ventilate criminals":

 Real time elapsed: 1.580348
 Run time elapsed:  1.1136131

   <0>--  S → NP[singular3] VP •  / Crockett wants to ventilate criminals

Chart:

<0>        <--- Crockett --->
<1>         <--- wants --->
<2>           <--- to --->
<3>        <--- ventilate --->
<4>        <--- criminals --->
<5>
```

Parse tree for "Crockett wants to ventilate criminals":

```
Edge <2>--  VP → VP VP •  / to ventilate criminals /  --<5

 Rule: Rule:{complements}

 Sources of the edge:
  Extended: <2>--  VP → VP • VP  / to /  --<3>
     with: <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>

 Directed graph:
    [*OtherComplements*: *3*=()
     *Subject*: *2*=[cat: NP
                     head: [trans: *1*=[]]]
     0: [cat: VP
         head: *6*=[form: infinitival
                    trans: *5*=[pred: *ventilate*
                                arg1: *1*
```

```
CL PATR command: Change Display
CL PATR command:
Unpublished Copyright (c) 1987, SRI International
Mouse-L, -R: Display 4.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, or Super.
[Tue 27 Oct 3:42:33]  Shieber          CL-USER:      User Input
```

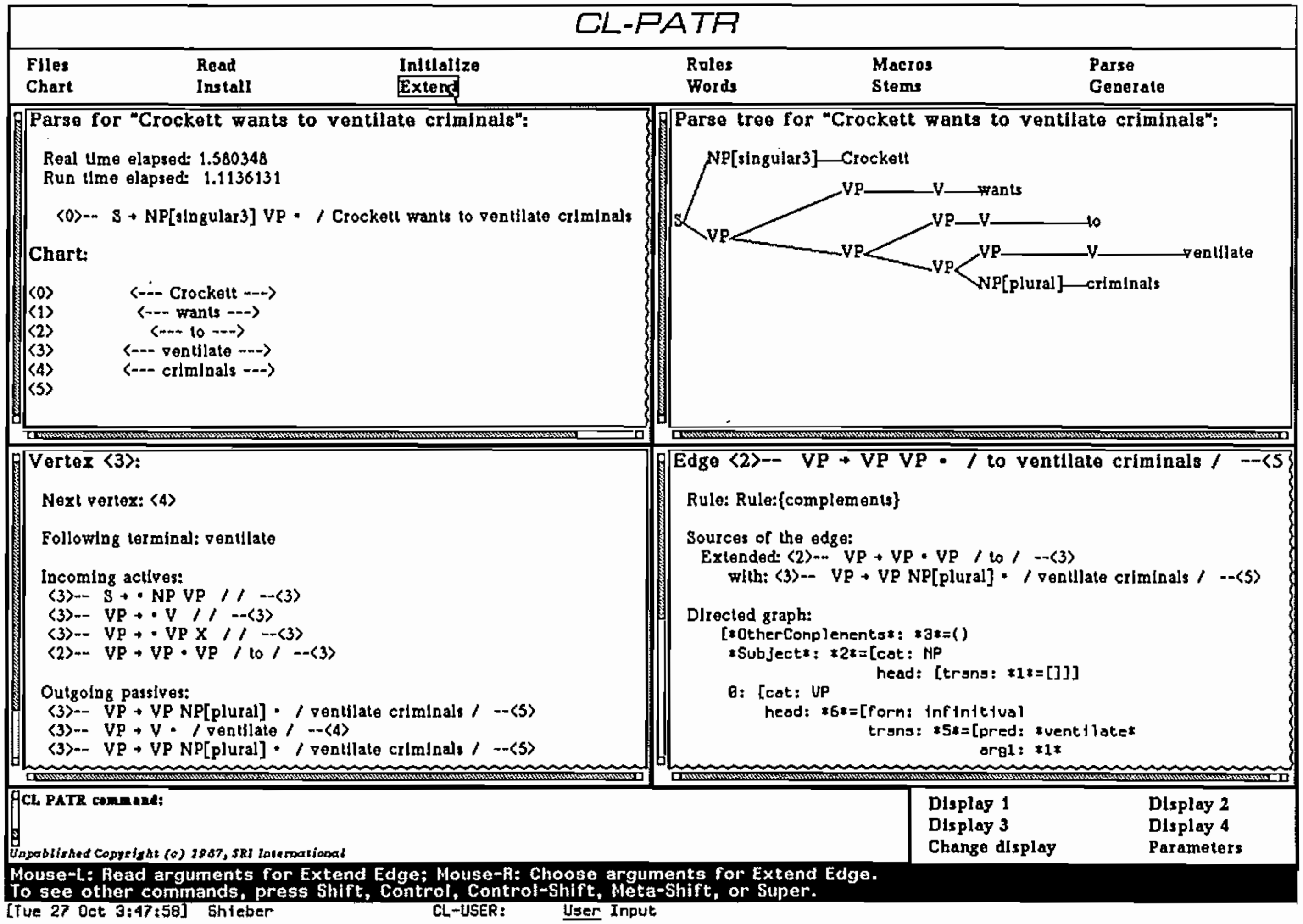
Figure 16



```
Edge <2>--  VP → VP VP •  / to ventilate criminals /  --<5>:

 Rule: Rule:{complements}

 Sources of the edge:
   Extended: <2>--  VP → VP • VP  / to /  --<3>
     with: <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>

 Directed graph:
     [*OtherComplements*: *3*=()
      *Subject*: *2*=[cat: NP
                      head: [trans: *1*=[]]]
      0: [cat: VP
          head: *6*=[form: infinitival
                     trans: *5*=[pred: *ventilate*
                                 arg1: *1*
                                 arg2: criminals]]
          subcat: [first: *2*
                   rest: *3*]]
      1: [cat: VP
          head: *6*
          subcat: [first: *2*
                   rest: [first: *4*=[cat: VP
                                      head: [form: nonfinite
                                             trans: *5*
                                             aux: false]
                                      subcat: [first: *2*
                                               rest: ()]]
                          rest: *3*]]]
      2: *4*]
```

```
CL PATR command: Display 4
CL PATR command: █
Unpublished Copyright (c) 1987, SRI International
Mouse-M: Expand this vertex.; Mouse-R: Menu.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, Super, or Super-Shift.
[Tue 27 Oct 3:42:52]  Shieber              CL-USER:      User Input
```

Figure 17



```
Parse for "Crockett wants to ventilate criminals":

  Real time elapsed: 1.580348
  Run time elapsed:  1.1136131

   <0>--  S → NP[singular3] VP •  / Crockett wants to ventilate criminals

Chart:

<0>          <--- Crockett --->
<1>           <--- wants --->
<2>             <--- to --->
<3>          <--- ventilate --->
<4>          <--- criminals --->
<5>
```

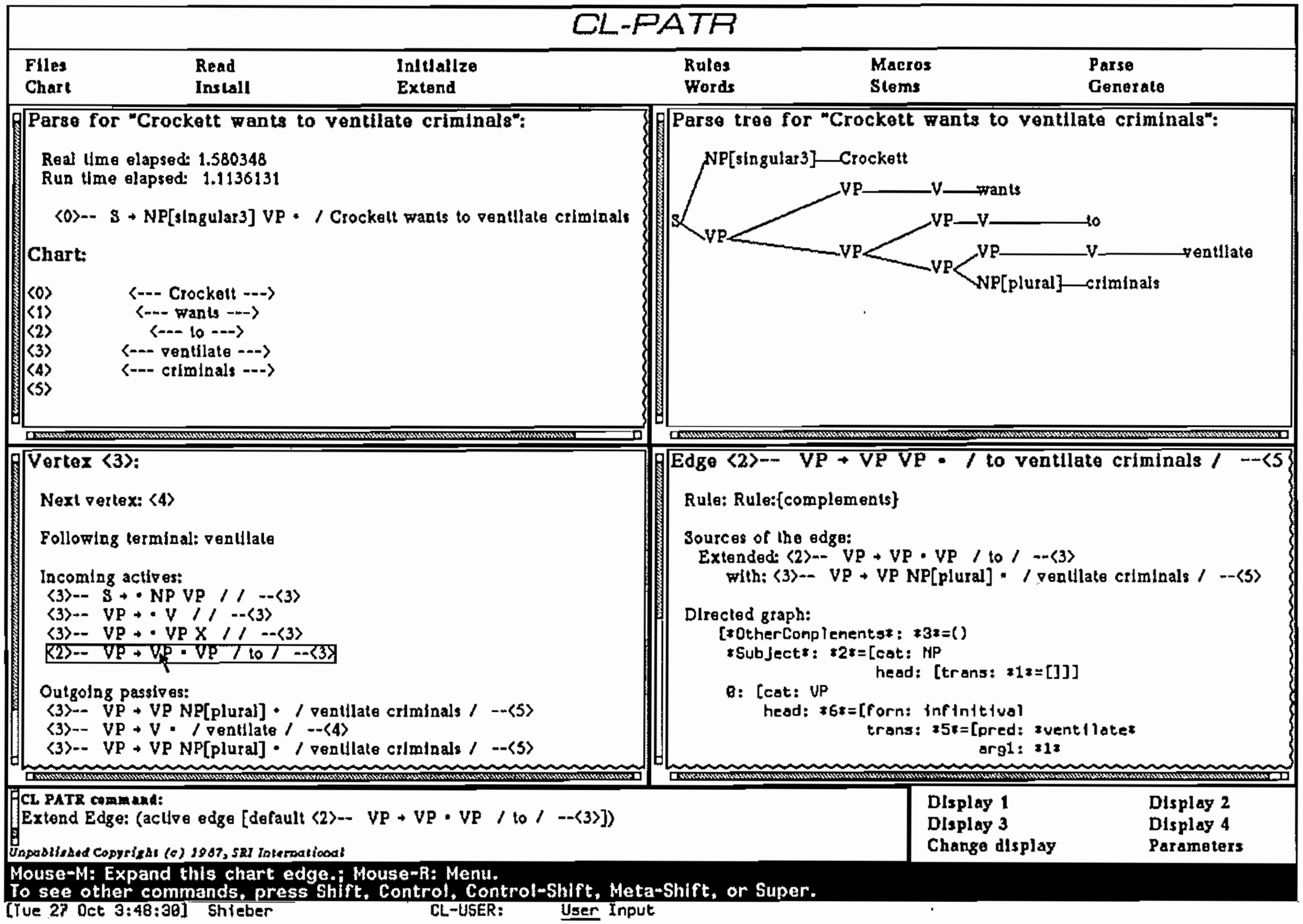
```
Parse tree for "Crockett wants to ventilate criminals":
```

```
Vertex <3>:

  Next vertex: <4>

  Following terminal: ventilate

  Incoming actives:
   <3>--  S → • NP VP  / /  --<3>
   <3>--  VP → • V  / /  --<3>
   <3>--  VP → • VP X  / /  --<3>
   <2>--  VP → VP • VP  / to /  --<3>

  Outgoing passives:
   <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>
   <3>--  VP → V •  / ventilate /  --<4>
   <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>
```

```
Edge <2>--  VP → VP VP •  / to ventilate criminals /  --<5

  Rule: Rule:{complements}

  Sources of the edge:
    Extended: <2>--  VP → VP • VP  / to /  --<3>
       with: <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>

  Directed graph:
     [*OtherComplements*: *3*=()
      *Subject*: *2*=[cat: NP
                      head: [trans: *1*=[]]]
      0: [cat: VP
          head: *6*=[form: infinitival
                     trans: *5*=[pred: *ventilate*
                                 arg1: *1*
```

```
CL PATR command:

Unpublished Copyright (c) 1987, SRI International
Mouse-L: Read arguments for Extend Edge; Mouse-R: Choose arguments for Extend Edge.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, or Super.
[Tue 27 Oct 3:47:58]  Shieber              CL-USER:      User Input
```

Figure 18



```
Parse for "Crockett wants to ventilate criminals":

  Real time elapsed: 1.580348
  Run time elapsed:  1.1136131

   <0>--  S → NP[singular3] VP •  / Crockett wants to ventilate criminals

Chart:

<0>          <--- Crockett --->
<1>           <--- wants --->
<2>             <--- to --->
<3>         <--- ventilate --->
<4>         <--- criminals --->
<5>
```

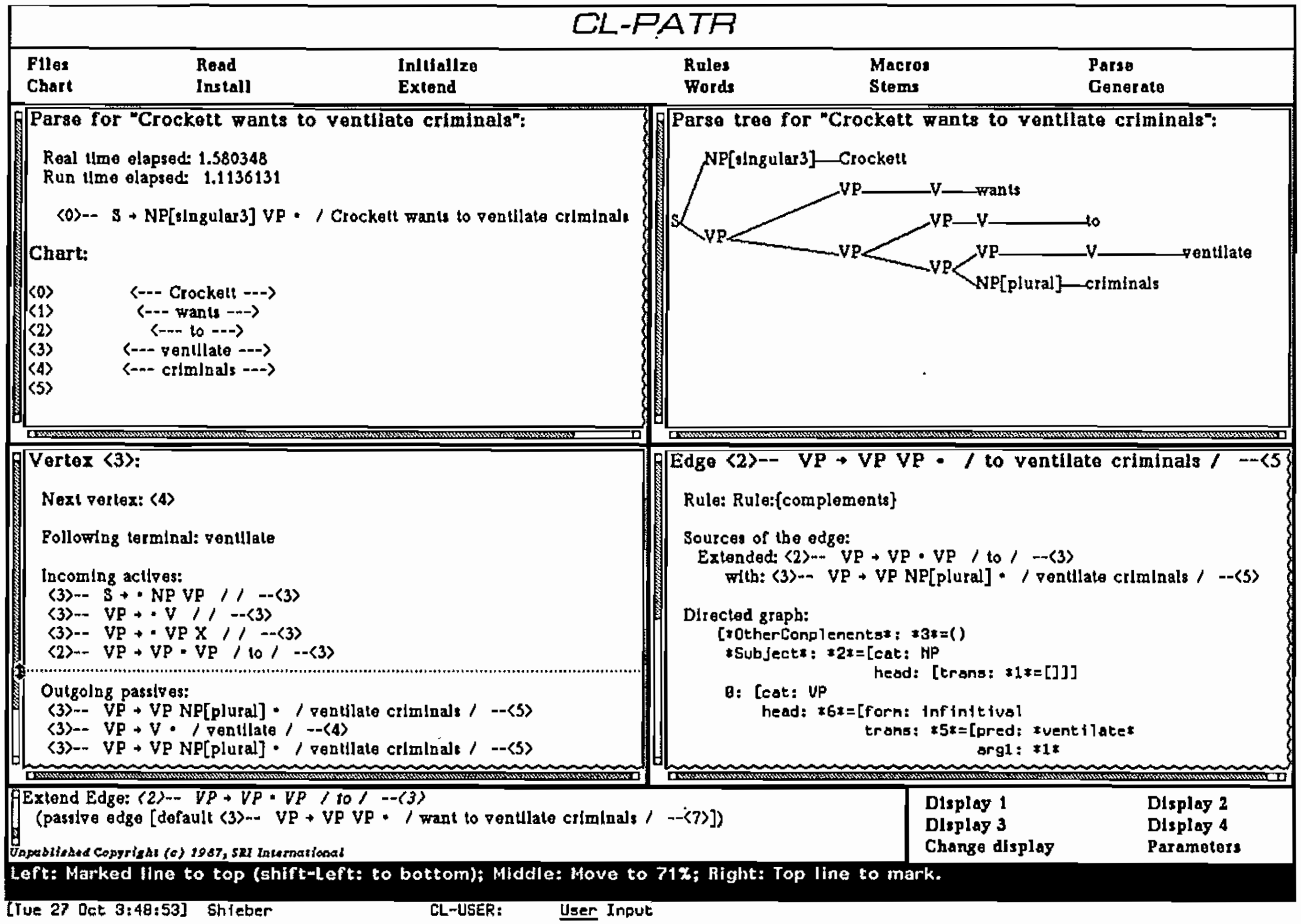
```
Parse tree for "Crockett wants to ventilate criminals":
```

```
Vertex <3>:

  Next vertex: <4>

  Following terminal: ventilate

  Incoming actives:
   <3>--  S → • NP VP  / /  --<3>
   <3>--  VP → • V  / /  --<3>
   <3>--  VP → • VP X  / /  --<3>
   <2>--  VP → VP • VP  / to /  --<3>

  Outgoing passives:
   <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>
   <3>--  VP → V •  / ventilate /  --<4>
   <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>
```

```
Edge <2>--  VP → VP VP •  / to ventilate criminals /  --<5

  Rule: Rule:{complements}

  Sources of the edge:
    Extended: <2>--  VP → VP • VP  / to /  --<3>
       with: <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>

  Directed graph:
       [*OtherComplements*: *3*=()
        *Subject*: *2*=[cat: NP
                        head: [trans: *1*=[]]]
        0: [cat: VP
            head: *6*=[form: infinitival
                       trans: *5*=[pred: *ventilate*
                                   arg1: *1*
```

```
CL PATR command:
Extend Edge: (active edge [default <2>--  VP → VP • VP  / to /  --<3>])
Unpublished Copyright (c) 1987, SRI International
Mouse-M: Expand this chart edge.; Mouse-R: Menu.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, or Super.
[Tue 27 Oct 3:48:30]  Shieber              CL-USER:      User Input
```

Figure 19



```
Parse for "Crockett wants to ventilate criminals":

  Real time elapsed: 1.580348
  Run time elapsed:  1.1136131

   <0>--  S → NP[singular3] VP •  / Crockett wants to ventilate criminals

Chart:

<0>          <--- Crockett --->
<1>           <--- wants --->
<2>             <--- to --->
<3>         <--- ventilate --->
<4>         <--- criminals --->
<5>
```

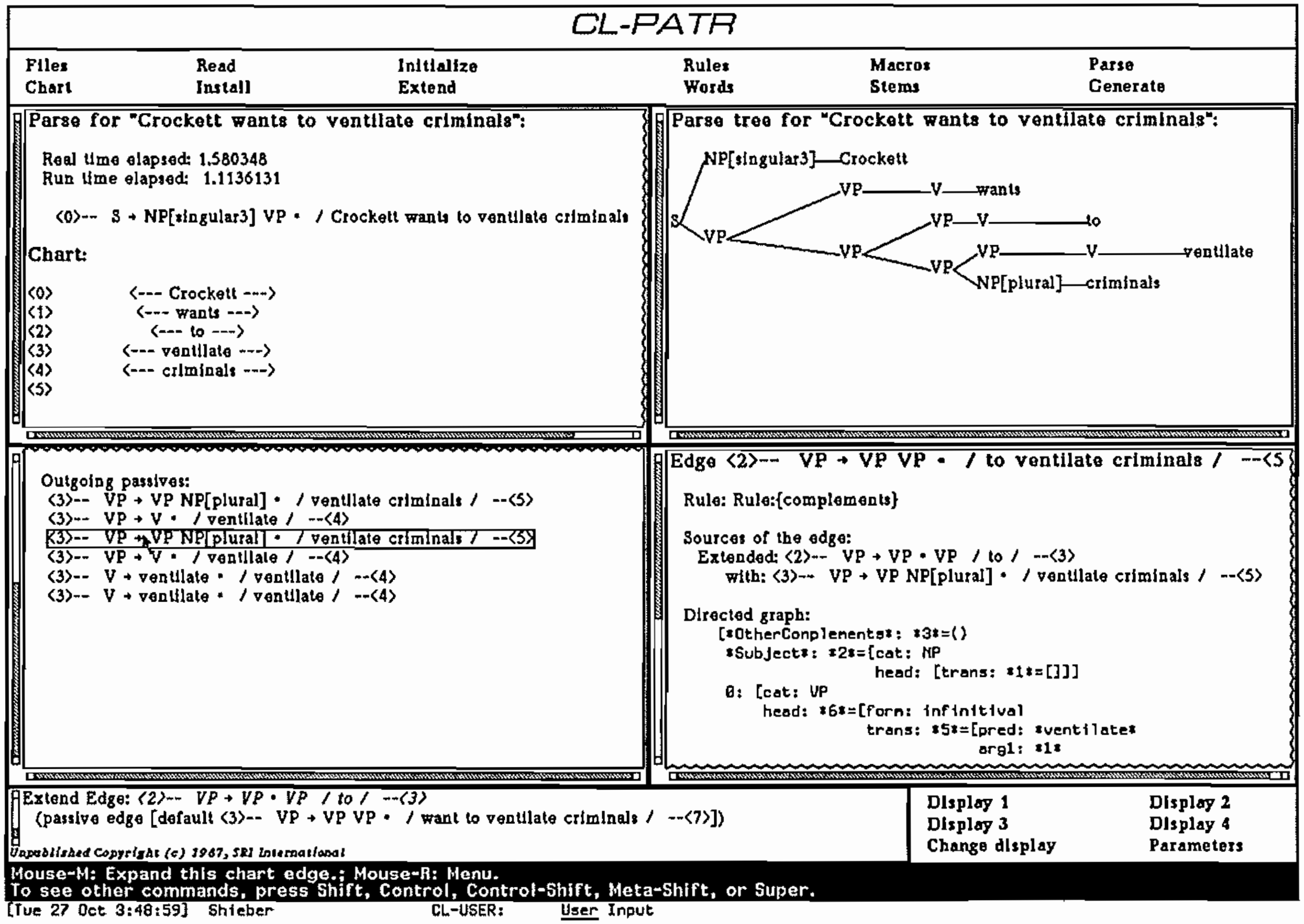
```
Parse tree for "Crockett wants to ventilate criminals":
```

```
Vertex <3>:

  Next vertex: <4>

  Following terminal: ventilate

  Incoming actives:
   <3>--  S → • NP VP  / /  --<3>
   <3>--  VP → • V  / /  --<3>
   <3>--  VP → • VP X  / /  --<3>
   <2>--  VP → VP • VP  / to /  --<3>

  Outgoing passives:
   <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>
   <3>--  VP → V •  / ventilate /  --<4>
   <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>
```

```
Edge <2>--  VP → VP VP •  / to ventilate criminals /  --<5

 Rule: Rule:{complements}

 Sources of the edge:
   Extended: <2>--  VP → VP • VP  / to /  --<3>
       with: <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>

 Directed graph:
     [*OtherComplements*: *3*=()
      *Subject*: *2*=[cat: NP
                      head: [trans: *1*=[]]]
      0: [cat: VP
          head: *6*=[form: infinitival
                     trans: *5*=[pred: *ventilate*
                                 arg1: *1*
```

```
Extend Edge: <2>--  VP → VP • VP  / to /  --<3>
  (passive edge [default <3>--  VP → VP VP •  / want to ventilate criminals /  --<7>])
Unpublished Copyright (c) 1987, SRI International
Left: Marked line to top (shift-Left: to bottom); Middle: Move to 71%; Right: Top line to mark.
[Tue 27 Oct 3:48:53]  Shieber              CL-USER:     User Input
```

Figure 20



```
Parse for "Crockett wants to ventilate criminals":

  Real time elapsed: 1.580348
  Run time elapsed:  1.1136131

   <0>--  S → NP[singular3] VP •  / Crockett wants to ventilate criminals

Chart:

<0>         <--- Crockett --->
<1>          <--- wants --->
<2>            <--- to --->
<3>         <--- ventilate --->
<4>         <--- criminals --->
<5>
```

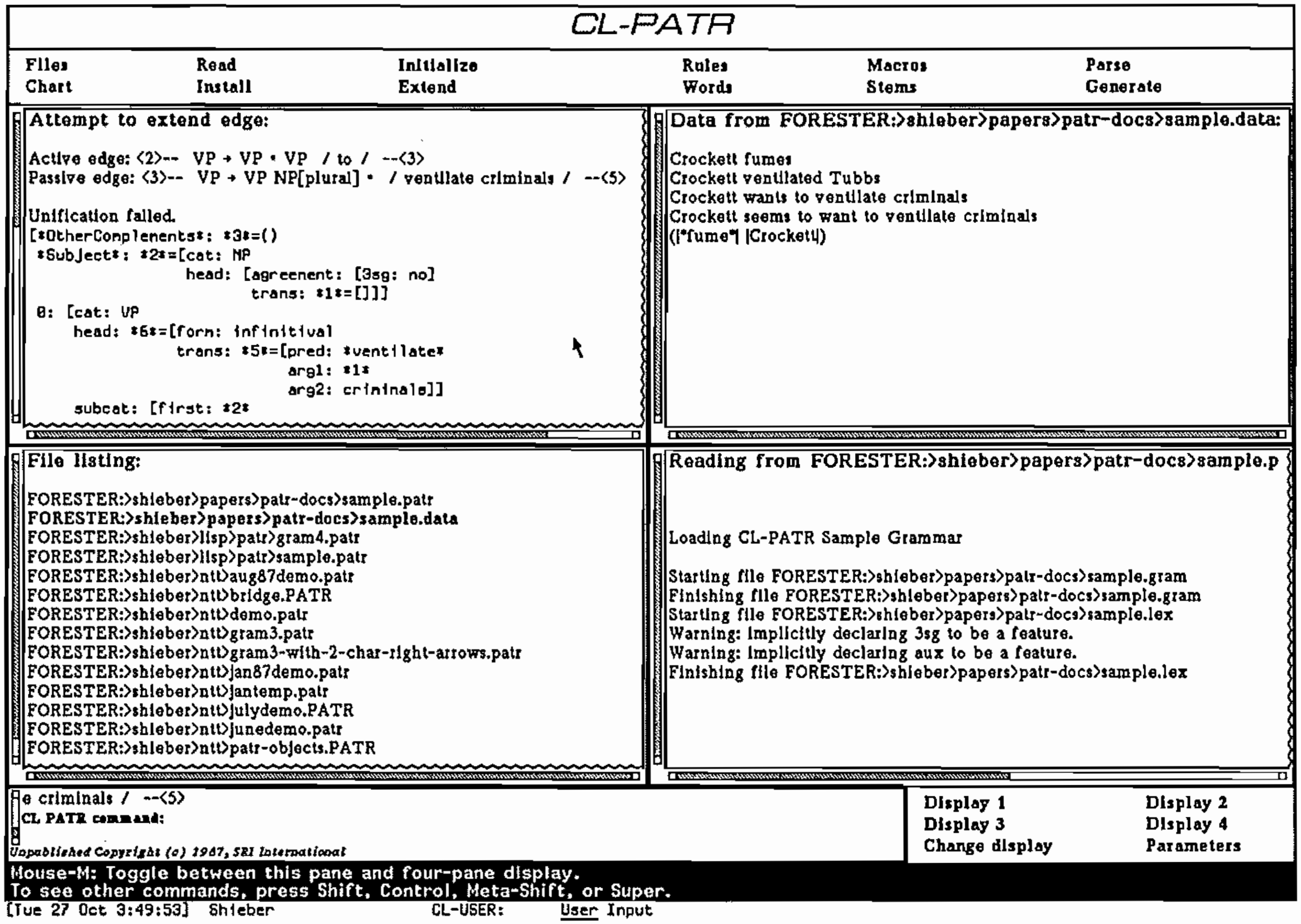
```
Parse tree for "Crockett wants to ventilate criminals":
```

```
  Outgoing passives:
   <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>
   <3>--  VP → V •  / ventilate /  --<4>
   <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>
   <3>--  VP → V •  / ventilate /  --<4>
   <3>--  V → ventilate •  / ventilate /  --<4>
   <3>--  V → ventilate •  / ventilate /  --<4>
```

```
Edge <2>--  VP → VP VP •  / to ventilate criminals /  --<5

  Rule: Rule:{complements}

  Sources of the edge:
    Extended: <2>--  VP → VP • VP  / to /  --<3>
        with: <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>

  Directed graph:
      [*OtherConplenents*: *3*=()
       *Subject*: *2*=[cat: NP
                       head: [trans: *1*=[]]]
       0: [cat: VP
           head: *6*=[forn: infinitival
                      trans: *5*=[pred: *ventilate*
                                  argl: *1*
```

```
Extend Edge: <2>--  VP → VP • VP  / to /  --<3>
  (passive edge [default <3>--  VP → VP VP •  / want to ventilate criminals /  --<7>])
Unpublished Copyright (c) 1987, SRI International
Mouse-M: Expand this chart edge.; Mouse-R: Menu.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, or Super.
[Tue 27 Oct 3:48:59]  Shieber                CL-USER:        User Input
```

Figure 21



```
Attempt to extend edge:

Active edge: <2>--  VP → VP • VP  / to /  --<3>
Passive edge: <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>

Unification failed.
[*OtherComplements*: *3*=()
 *Subject*: *2*=[cat: NP
                 head: [agreement: [3sg: no]
                        trans: *1*=[]]]
 0: [cat: VP
     head: *6*=[form: infinitival
                trans: *5*=[pred: *ventilate*
                            arg1: *1*
                            arg2: criminals]]
     subcat: [first: *2*
```

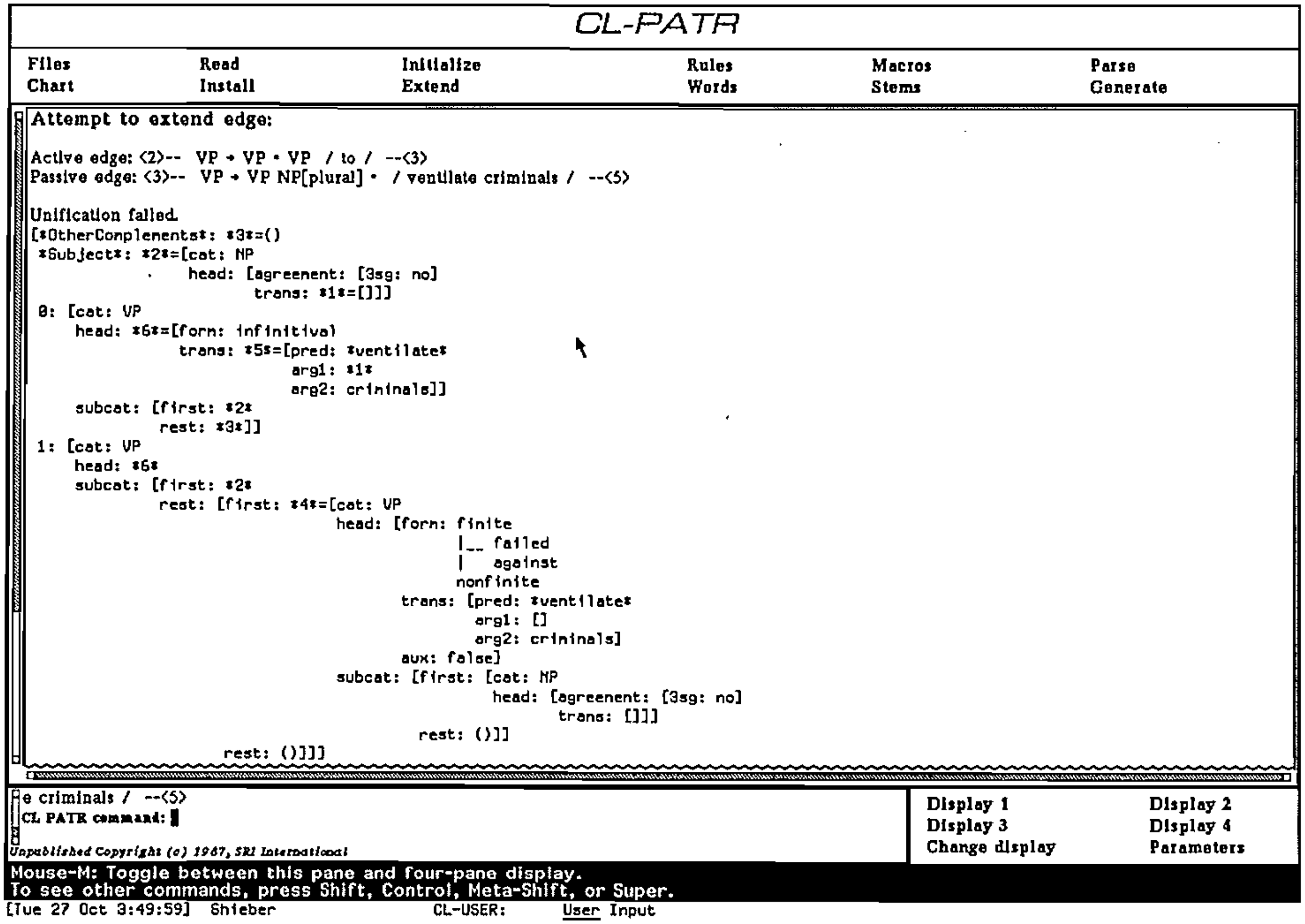
```
Data from FORESTER:>shieber>papers>patr-docs>sample.data:

Crockett fumes
Crockett ventilated Tubbs
Crockett wants to ventilate criminals
Crockett seems to want to ventilate criminals
(|*fume*| |Crockett|)
```

```
File listing:

FORESTER:>shieber>papers>patr-docs>sample.patr
FORESTER:>shieber>papers>patr-docs>sample.data
FORESTER:>shieber>lisp>patr>gram4.patr
FORESTER:>shieber>lisp>patr>sample.patr
FORESTER:>shieber>ntt>aug87demo.patr
FORESTER:>shieber>ntt>bridge.PATR
FORESTER:>shieber>ntt>demo.patr
FORESTER:>shieber>ntt>gram3.patr
FORESTER:>shieber>ntt>gram3-with-2-char-right-arrows.patr
FORESTER:>shieber>ntt>jan87demo.patr
FORESTER:>shieber>ntt>jantemp.patr
FORESTER:>shieber>ntt>julydemo.PATR
FORESTER:>shieber>ntt>junedemo.patr
FORESTER:>shieber>ntt>patr-objects.PATR
```

```
Reading from FORESTER:>shieber>papers>patr-docs>sample.p


Loading CL-PATR Sample Grammar

Starting file FORESTER:>shieber>papers>patr-docs>sample.gram
Finishing file FORESTER:>shieber>papers>patr-docs>sample.gram
Starting file FORESTER:>shieber>papers>patr-docs>sample.lex
Warning: implicitly declaring 3sg to be a feature.
Warning: implicitly declaring aux to be a feature.
Finishing file FORESTER:>shieber>papers>patr-docs>sample.lex
```

```
e criminals /  --<5>
CL PATR command:

Unpublished Copyright (c) 1987, SRI International
Mouse-M: Toggle between this pane and four-pane display.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Tue 27 Oct 3:49:53]  Shieber              CL-USER:      User Input
```

Figure 22



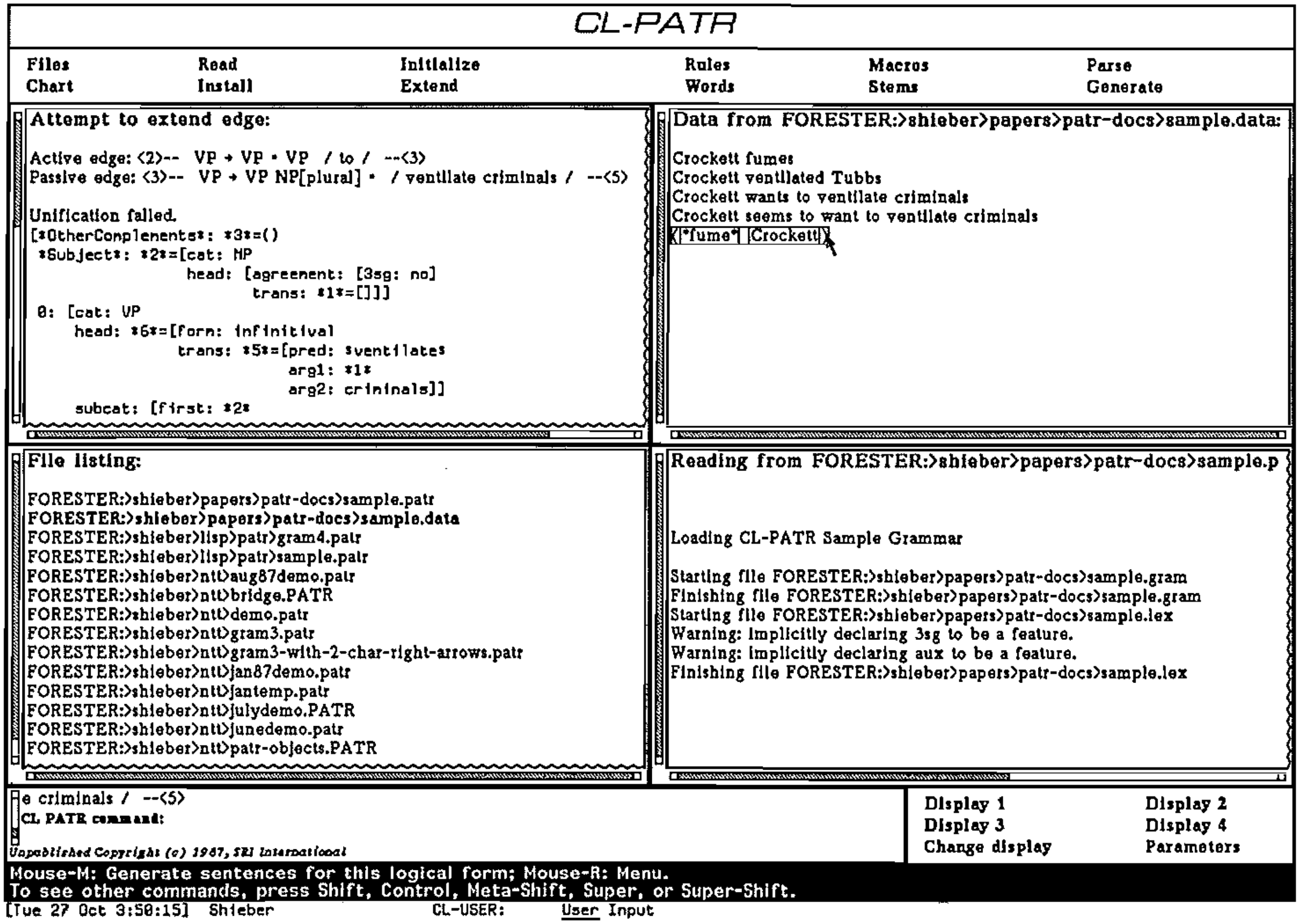
```
Attempt to extend edge:

Active edge: <2>--  VP → VP • VP  / to /  --<3>
Passive edge: <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>

Unification failed.
[*OtherComplements*: *3*=()
 *Subject*: *2*=[cat: NP
                 head: [agreement: [3sg: no]
                        trans: *1*=[]]]
 0: [cat: VP
     head: *6*=[form: infinitival
                trans: *5*=[pred: *ventilate*
                            arg1: *1*
                            arg2: criminals]]
     subcat: [first: *2*
              rest: *3*]]
 1: [cat: VP
     head: *6*
     subcat: [first: *2*
              rest: [first: *4*=[cat: VP
                                 head: [form: finite
                                              |__ failed
                                              |   against
                                              nonfinite
                                        trans: [pred: *ventilate*
                                                arg1: []
                                                arg2: criminals]
                                        aux: false]
                                 subcat: [first: [cat: NP
                                                  head: [agreement: [3sg: no]
                                                         trans: []]]
                                          rest: ()]]
                     rest: ()]]]
```

```
e criminals /  --<5>
CL PATR command:
```

Unpublished Copyright (c) 1987, SRI International

```
Mouse-M: Toggle between this pane and four-pane display.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Tue 27 Oct 3:49:59]  Shieber          CL-USER:      User Input
```

Figure 23



```
Attempt to extend edge:

Active edge: <2>--  VP → VP • VP  / to /  --<3>
Passive edge: <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>

Unification failed.
[*OtherConplenents*: *3*=()
 *Subject*: *2*=[cat: NP
                 head: [agreenent: [3sg: no]
                        trans: *1*=[]]]
 0: [cat: VP
     head: *6*=[forn: infinitival
                trans: *5*=[pred: $ventilate$
                            arg1: *1*
                            arg2: crininals]]
     subcat: [first: *2*
```

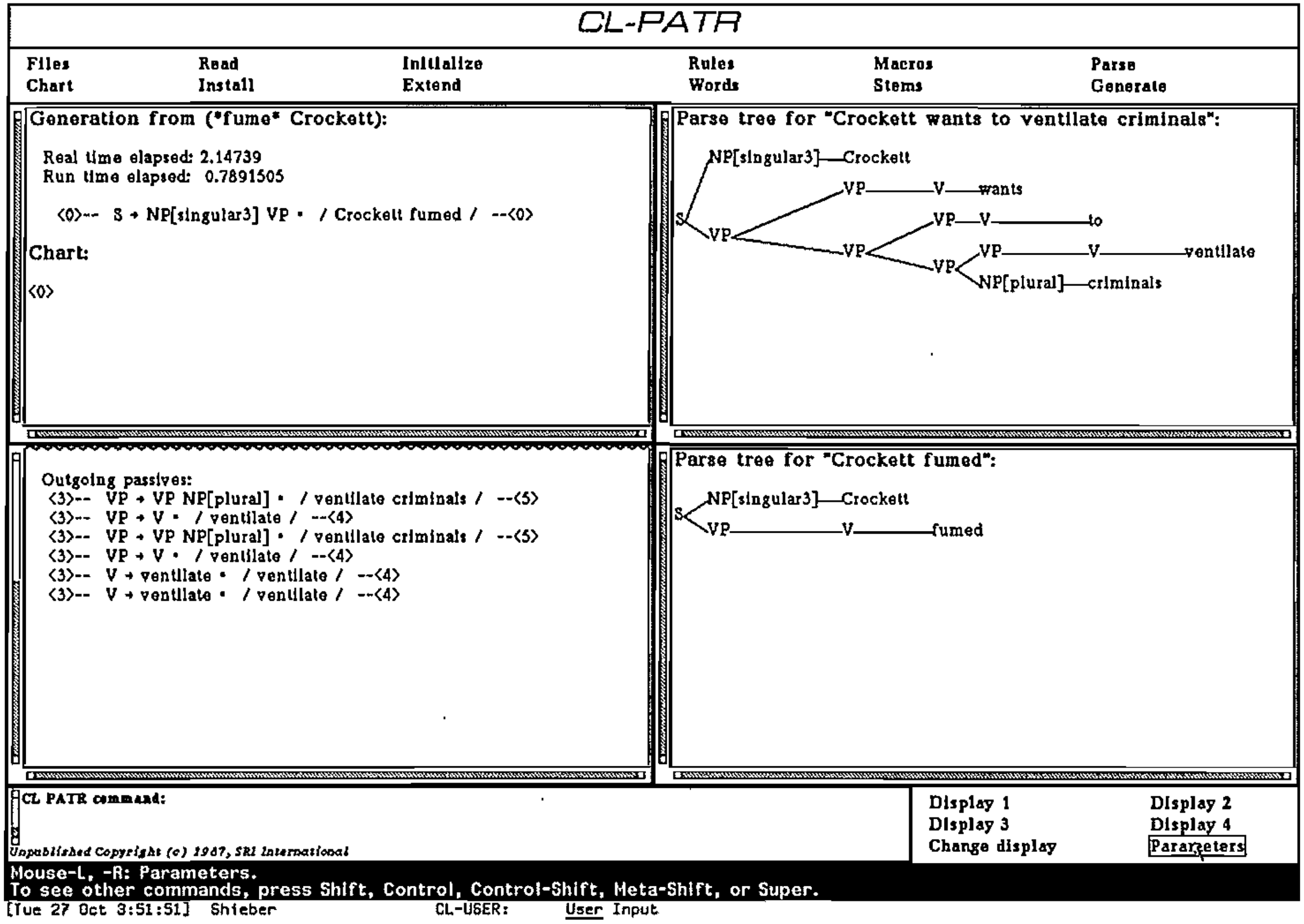
```
Data from FORESTER:>shieber>papers>patr-docs>sample.data:

Crockett fumes
Crockett ventilated Tubbs
Crockett wants to ventilate criminals
Crockett seems to want to ventilate criminals
(*fume* Crockett)
```

```
File listing:

FORESTER:>shieber>papers>patr-docs>sample.patr
FORESTER:>shieber>papers>patr-docs>sample.data
FORESTER:>shieber>lisp>patr>gram4.patr
FORESTER:>shieber>lisp>patr>sample.patr
FORESTER:>shieber>ntl>aug87demo.patr
FORESTER:>shieber>ntl>bridge.PATR
FORESTER:>shieber>ntl>demo.patr
FORESTER:>shieber>ntl>gram3.patr
FORESTER:>shieber>ntl>gram3-with-2-char-right-arrows.patr
FORESTER:>shieber>ntl>jan87demo.patr
FORESTER:>shieber>ntl>jantemp.patr
FORESTER:>shieber>ntl>julydemo.PATR
FORESTER:>shieber>ntl>junedemo.patr
FORESTER:>shieber>ntl>patr-objects.PATR
```

```
Reading from FORESTER:>shieber>papers>patr-docs>sample.p


Loading CL-PATR Sample Grammar

Starting file FORESTER:>shieber>papers>patr-docs>sample.gram
Finishing file FORESTER:>shieber>papers>patr-docs>sample.gram
Starting file FORESTER:>shieber>papers>patr-docs>sample.lex
Warning: Implicitly declaring 3sg to be a feature.
Warning: Implicitly declaring aux to be a feature.
Finishing file FORESTER:>shieber>papers>patr-docs>sample.lex
```

```
e criminals /  --<5>
CL PATR command:
Unpublished Copyright (c) 1987, SRI International
Mouse-M: Generate sentences for this logical form; Mouse-R: Menu.
To see other commands, press Shift, Control, Meta-Shift, Super, or Super-Shift.
[Tue 27 Oct 3:50:15]  Shieber            CL-USER:       User Input
```

Figure 24



```
Generation from (*fume* Crockett):

  Real time elapsed: 2.14739
  Run time elapsed:  0.7891505

    <0>--  S → NP[singular3] VP •  / Crockett fumed /  --<0>

Chart:

<0>
```

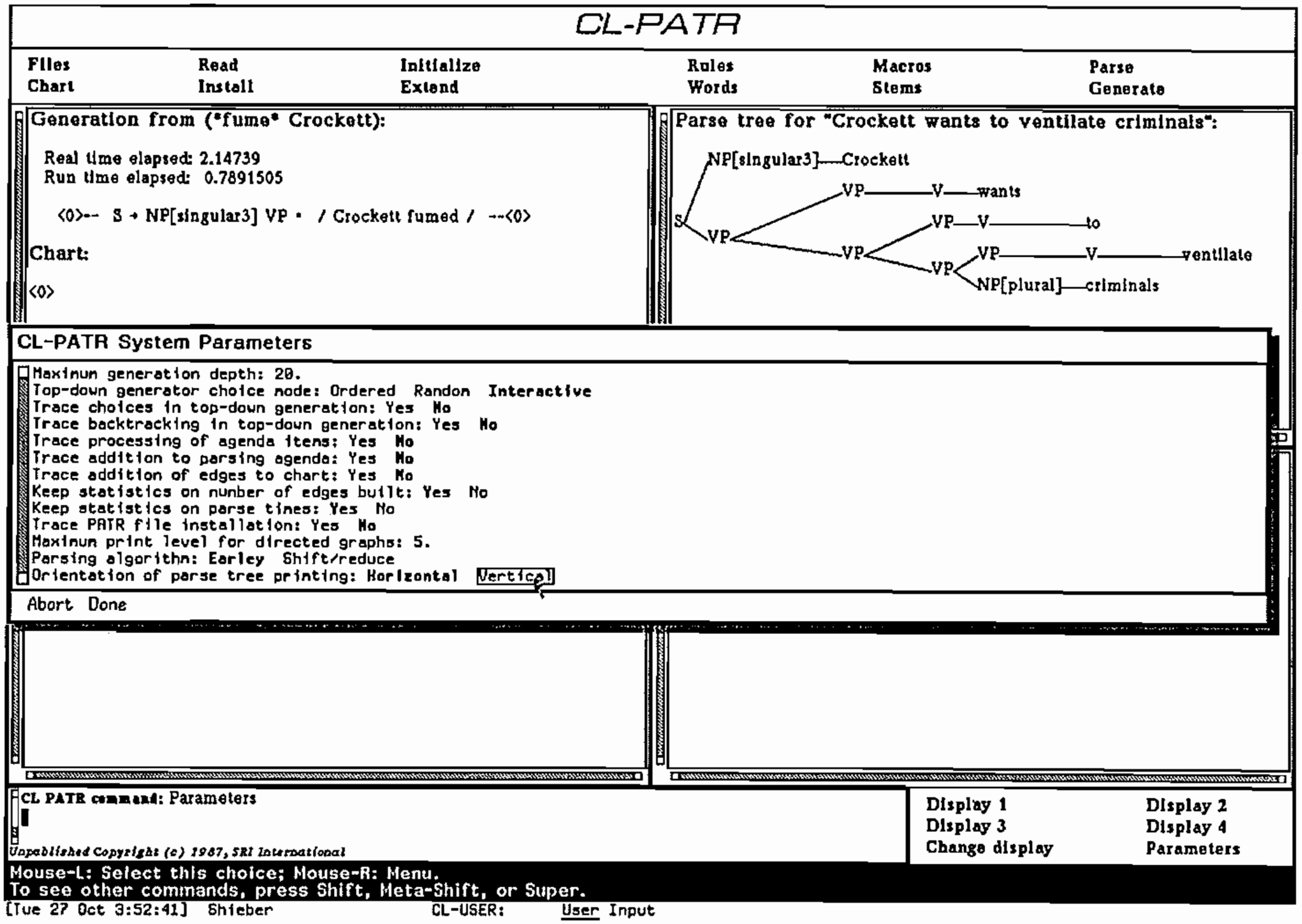
Parse tree for "Crockett wants to ventilate criminals":

```
  Outgoing passives:
   <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>
   <3>--  VP → V •  / ventilate /  --<4>
   <3>--  VP → VP NP[plural] •  / ventilate criminals /  --<5>
   <3>--  VP → V •  / ventilate /  --<4>
   <3>--  V → ventilate •  / ventilate /  --<4>
   <3>--  V → ventilate •  / ventilate /  --<4>
```

Parse tree for "Crockett fumed":

```
CL PATR command:

Unpublished Copyright (c) 1987, SRI International
Mouse-L, -R: Parameters.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, or Super.
[Tue 27 Oct 3:51:51]  Shieber              CL-USER:     User Input
```

Figure 25



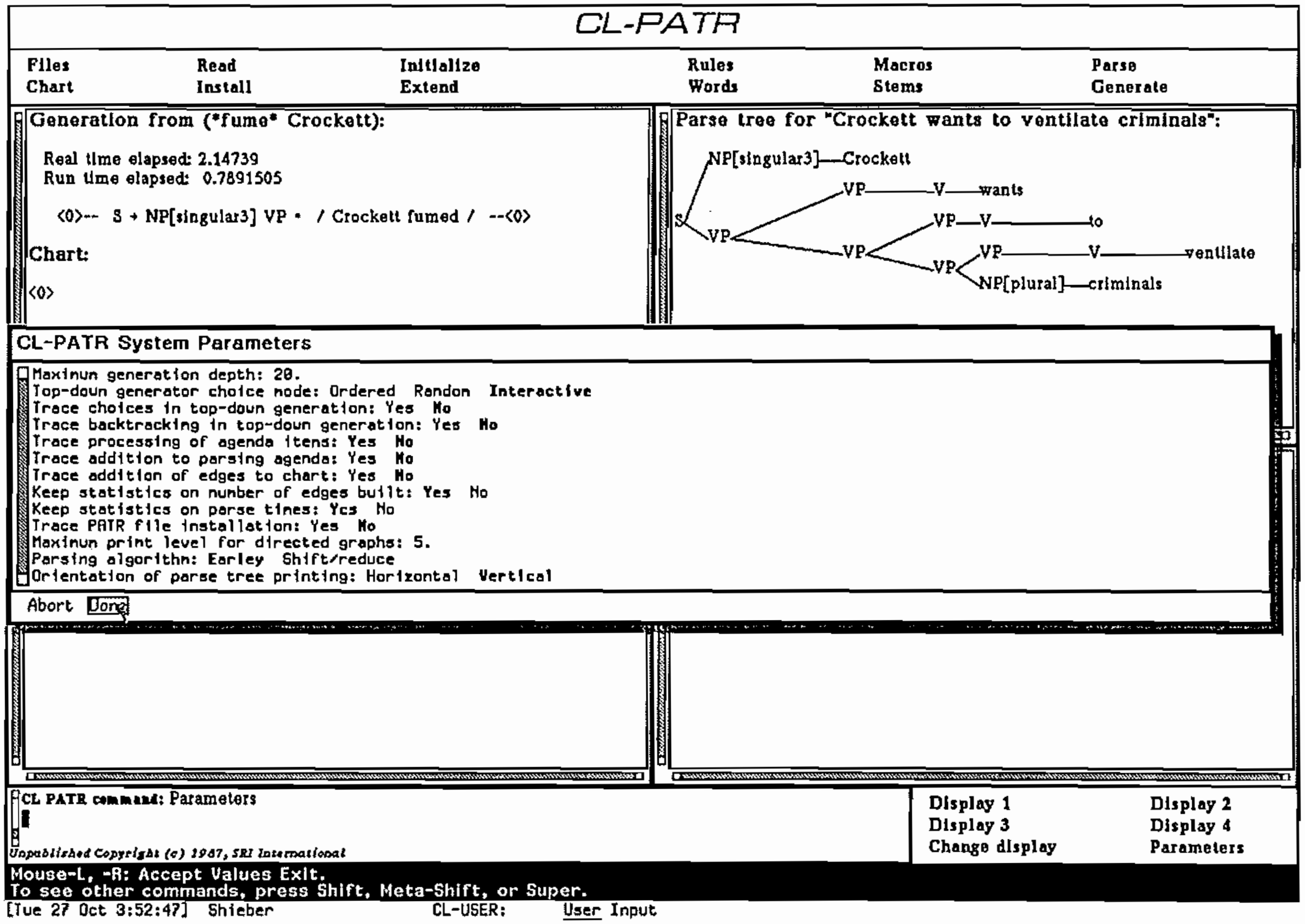
```
Generation from (*fume* Crockett):

  Real time elapsed: 2.14739
  Run time elapsed:  0.7891505

    <0>-- S → NP[singular3] VP •  / Crockett fumed /  --<0>

Chart:

<0>
```

```
Parse tree for "Crockett wants to ventilate criminals":
```

```
CL-PATR System Parameters

Maximum generation depth: 20.
Top-down generator choice mode: Ordered  Random  Interactive
Trace choices in top-down generation: Yes  No
Trace backtracking in top-down generation: Yes  No
Trace processing of agenda items: Yes  No
Trace addition to parsing agenda: Yes  No
Trace addition of edges to chart: Yes  No
Keep statistics on number of edges built: Yes  No
Keep statistics on parse times: Yes  No
Trace PATR file installation: Yes  No
Maximum print level for directed graphs: 5.
Parsing algorithm: Earley  Shift/reduce
Orientation of parse tree printing: Horizontal  Vertical
```

```
CL PATR command: Parameters

Unpublished Copyright (c) 1987, SRI International
Mouse-L: Select this choice; Mouse-R: Menu.
To see other commands, press Shift, Meta-Shift, or Super.
[Tue 27 Oct 3:52:41]  Shieber              CL-USER:      User Input
```

Figure 26



Generation from (*fume* Crockett):

```
Real time elapsed: 2.14739
Run time elapsed: 0.7891505

 <0>-- S → NP[singular3] VP •  / Crockett fumed /  --<0>
```

Chart:

<0>

Parse tree for "Crockett wants to ventilate criminals":

CL-PATR System Parameters

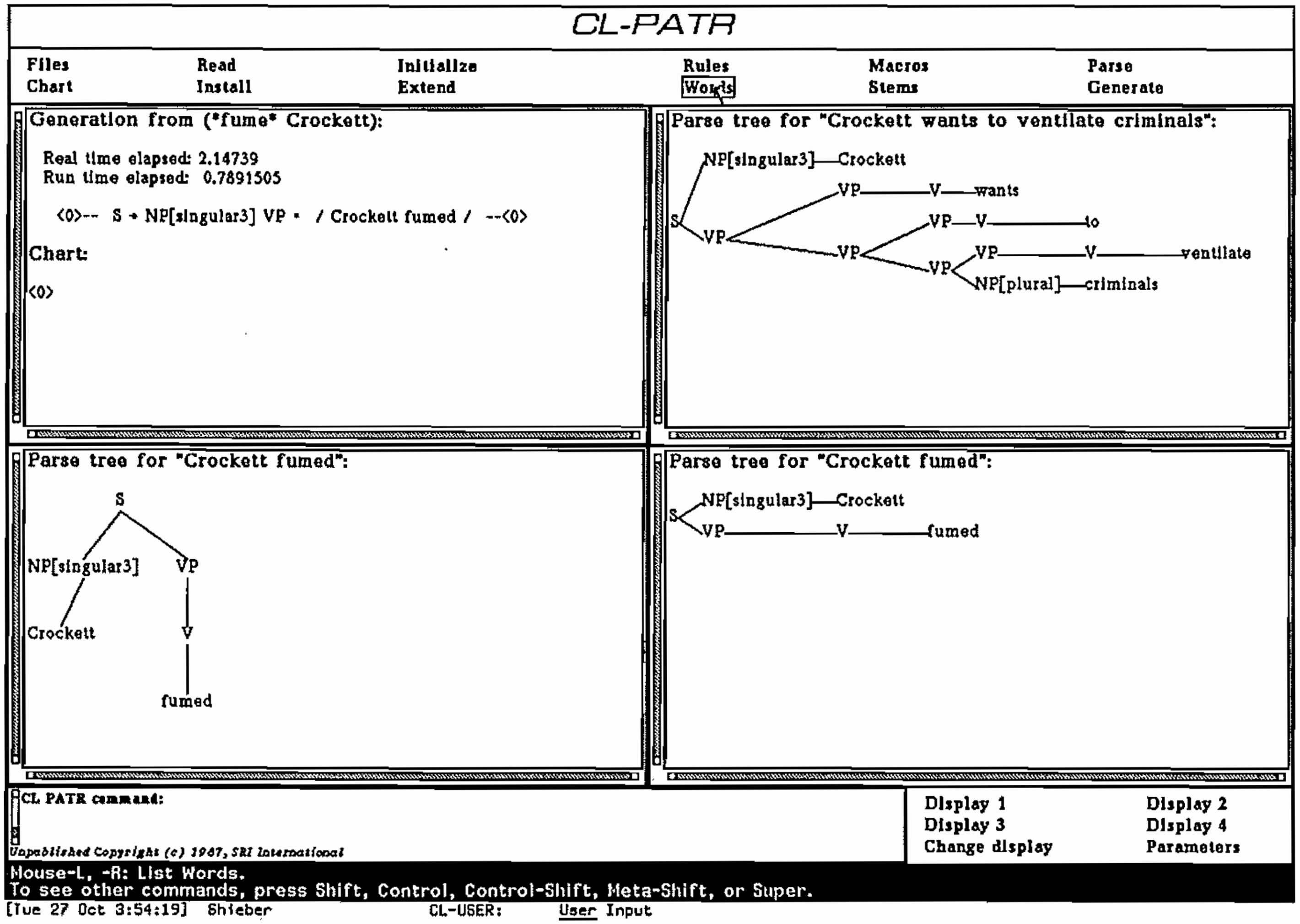
```
Maxinun generation depth: 20.
Top-doun generator choice node: Ordered  Randon  Interactive
Trace choices in top-doun generation: Yes  No
Trace backtracking in top-doun generation: Yes  No
Trace processing of agenda itens: Yes  No
Trace addition to parsing agenda: Yes  No
Trace addition of edges to chart: Yes  No
Keep statistics on nunber of edges built: Yes  No
Keep statistics on parse tines: Yes  No
Trace PATR file installation: Yes  No
Maxinun print level for directed graphs: 5.
Parsing algorithn: Earley  Shift/reduce
Orientation of parse tree printing: Horizontal  Vertical
```

CL PATR command: Parameters

Unpublished Copyright (c) 1987, SRI International

```
Mouse-L, -R: Accept Values Exit.
To see other commands, press Shift, Meta-Shift, or Super.
[Tue 27 Oct 3:52:47]  Shieber           CL-USER:      User Input
```

Figure 27



Generation from (*fume* Crockett):

Real time elapsed: 2.14739
Run time elapsed: 0.7891505

```
<0>-- S → NP[singular3] VP •  / Crockett fumed /  --<0>
```

Chart:

<0>

Parse tree for "Crockett wants to ventilate criminals":

Parse tree for "Crockett fumed":

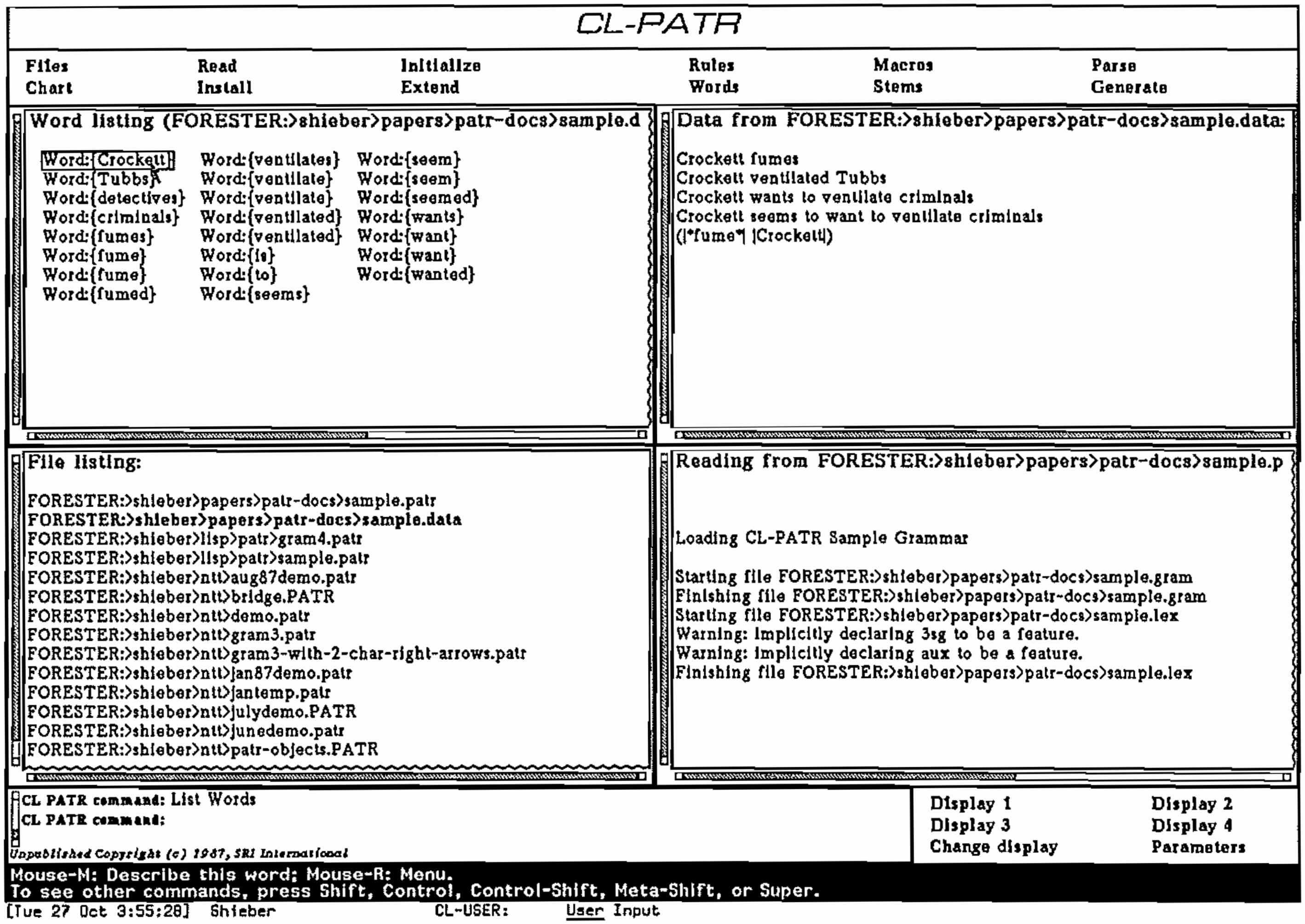
Parse tree for "Crockett fumed":

CL PATR command:

Unpublished Copyright (c) 1987, SRI International

```
Mouse-L, -R: List Words.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, or Super.
[Tue 27 Oct 3:54:19]  Shieber            CL-USER:     User Input
```

Figure 28



```
Word listing (FORESTER:>shieber>papers>patr-docs>sample.d

Word:{Crockett}   Word:{ventilates}  Word:{seem}
Word:{Tubbs}      Word:{ventilate}   Word:{seem}
Word:{detectives} Word:{ventilate}   Word:{seemed}
Word:{criminals}  Word:{ventilated}  Word:{wants}
Word:{fumes}      Word:{ventilated}  Word:{want}
Word:{fume}       Word:{is}          Word:{want}
Word:{fume}       Word:{to}          Word:{wanted}
Word:{fumed}      Word:{seems}
```

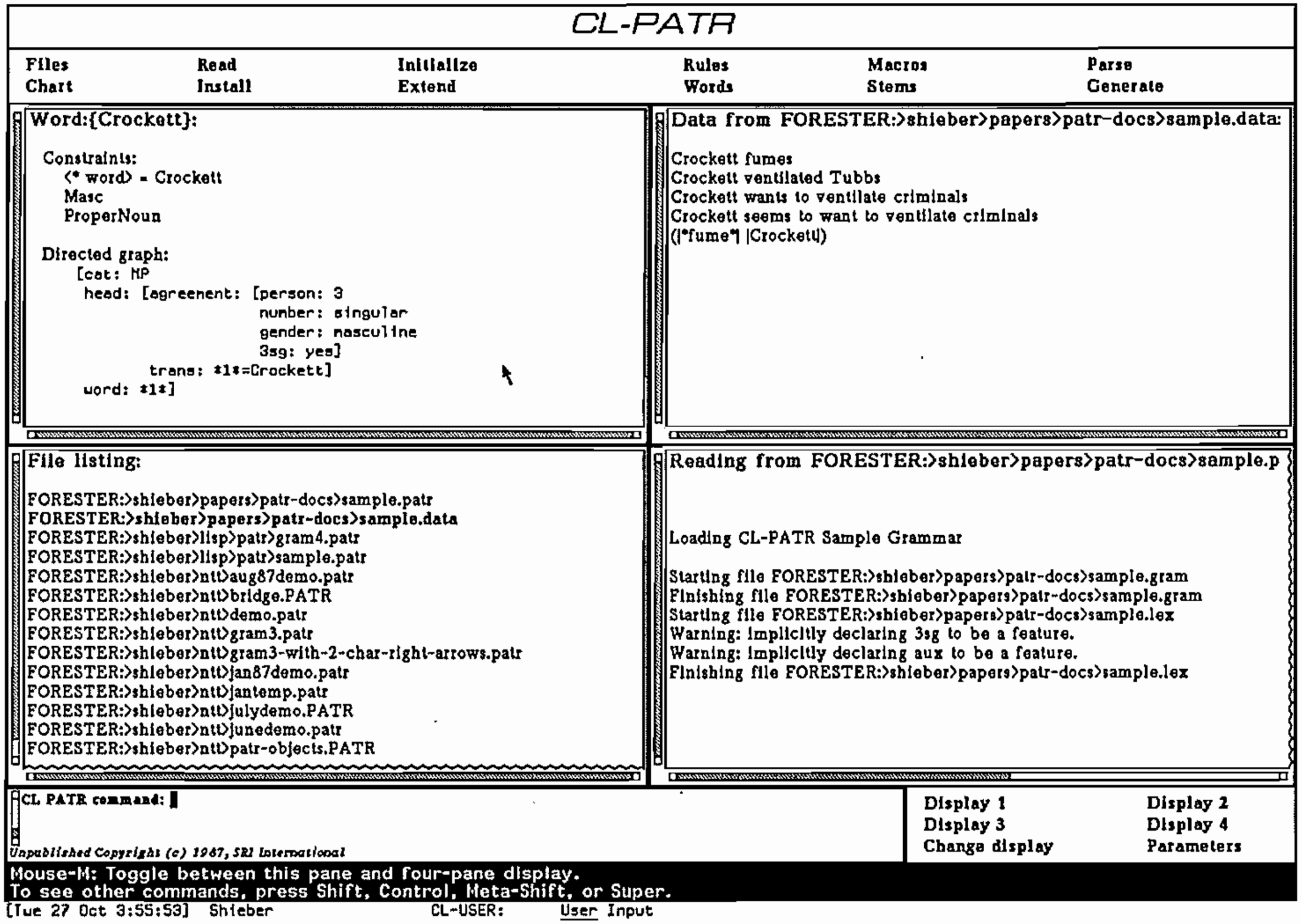
```
Data from FORESTER:>shieber>papers>patr-docs>sample.data:

Crockett fumes
Crockett ventilated Tubbs
Crockett wants to ventilate criminals
Crockett seems to want to ventilate criminals
(|*fume*| |Crockett|)
```

```
File listing:

FORESTER:>shieber>papers>patr-docs>sample.patr
FORESTER:>shieber>papers>patr-docs>sample.data
FORESTER:>shieber>lisp>patr>gram4.patr
FORESTER:>shieber>lisp>patr>sample.patr
FORESTER:>shieber>ntt>aug87demo.patr
FORESTER:>shieber>ntt>bridge.PATR
FORESTER:>shieber>ntt>demo.patr
FORESTER:>shieber>ntt>gram3.patr
FORESTER:>shieber>ntt>gram3-with-2-char-right-arrows.patr
FORESTER:>shieber>ntt>jan87demo.patr
FORESTER:>shieber>ntt>jantemp.patr
FORESTER:>shieber>ntt>julydemo.PATR
FORESTER:>shieber>ntt>junedemo.patr
FORESTER:>shieber>ntt>patr-objects.PATR
```

```
Reading from FORESTER:>shieber>papers>patr-docs>sample.p


Loading CL-PATR Sample Grammar

Starting file FORESTER:>shieber>papers>patr-docs>sample.gram
Finishing file FORESTER:>shieber>papers>patr-docs>sample.gram
Starting file FORESTER:>shieber>papers>patr-docs>sample.lex
Warning: Implicitly declaring 3sg to be a feature.
Warning: Implicitly declaring aux to be a feature.
Finishing file FORESTER:>shieber>papers>patr-docs>sample.lex
```

```
CL PATR command: List Words
CL PATR command:
Unpublished Copyright (c) 1987, SRI International
Mouse-M: Describe this word; Mouse-R: Menu.
To see other commands, press Shift, Control, Control-Shift, Meta-Shift, or Super.
[Tue 27 Oct 3:55:28]  Shieber          CL-USER:      User Input
```

Figure 29



```
Word:{Crockett}:

 Constraints:
   <* word> = Crockett
   Masc
   ProperNoun

 Directed graph:
     [cat: NP
      head: [agreement: [person: 3
                         number: singular
                         gender: masculine
                         3sg: yes]
             trans: *1*=Crockett]
      word: *1*]
```

```
Data from FORESTER:>shieber>papers>patr-docs>sample.data:

Crockett fumes
Crockett ventilated Tubbs
Crockett wants to ventilate criminals
Crockett seems to want to ventilate criminals
(|*fume*| |Crockett|)
```

```
File listing:

FORESTER:>shieber>papers>patr-docs>sample.patr
FORESTER:>shieber>papers>patr-docs>sample.data
FORESTER:>shieber>lisp>patr>gram4.patr
FORESTER:>shieber>lisp>patr>sample.patr
FORESTER:>shieber>ntt>aug87demo.patr
FORESTER:>shieber>ntt>bridge.PATR
FORESTER:>shieber>ntt>demo.patr
FORESTER:>shieber>ntt>gram3.patr
FORESTER:>shieber>ntt>gram3-with-2-char-right-arrows.patr
FORESTER:>shieber>ntt>jan87demo.patr
FORESTER:>shieber>ntt>jantemp.patr
FORESTER:>shieber>ntt>julydemo.PATR
FORESTER:>shieber>ntt>junedemo.patr
FORESTER:>shieber>ntt>patr-objects.PATR
```

```
Reading from FORESTER:>shieber>papers>patr-docs>sample.p


Loading CL-PATR Sample Grammar

Starting file FORESTER:>shieber>papers>patr-docs>sample.gram
Finishing file FORESTER:>shieber>papers>patr-docs>sample.gram
Starting file FORESTER:>shieber>papers>patr-docs>sample.lex
Warning: Implicitly declaring 3sg to be a feature.
Warning: Implicitly declaring aux to be a feature.
Finishing file FORESTER:>shieber>papers>patr-docs>sample.lex
```

```
CL PATR command: █

Unpublished Copyright (c) 1987, SRI International
Mouse-M: Toggle between this pane and four-pane display.
To see other commands, press Shift, Control, Meta-Shift, or Super.
[Tue 27 Oct 3:55:53]  Shieber          CL-USER:     User Input
```

Figure 30

Chapter 11

Other Enhanced Interfaces

11.1 Extensions for the Macintosh

The Macintosh version of CL-PATR written for Allegro COMMON LISP adds functionality to the bare command interface by adding keystroke equivalents for certain of the special commands. The Macintosh interface runs in a window titled "Portable Interface" which is created upon loading the CL-PATR system. This window works exactly like a Fred window (the EMACS-like editor windows available under Allegro CL) except that certain keys are rebound to make interaction with the interface easier. Among the capabilities of Fred windows that the interface window inherits are scrolling, searching for strings, and saving of the buffer contents in a file.

11.1.1 Macintosh Keystroke Equivalents

Typically, the rebound keys, when pressed while the insertion cursor is anywhere on a labeled line, execute a special command taking as its argument the label of that line. The command thus invoked is also printed after the prompt at the end of the buffer, so that the interaction appears as if the command had actually been typed in. Finally, the typed command includes a comment (after the semicolon) that indicates the contents of the labeled line where the key was pressed.

Display equivalents `<enter>`, `<return>`: The `<enter>` and `<return>` keys, when pressed on a labeled line, execute the `Display` command taking as its argument the label of that line. When pressed on a line with a prompt but no label (i.e., one with a command that has been previously entered) the command is reinvoked. Thus, the user can reexecute a command merely by scrolling back to that command, placing the cursor anywhere on the line and hitting return.

Help equivalent `<help>`: The `<help>` key, when pressed anywhere, regardless of whether the line has a prompt, a label, or neither, causes execution of the `Help` command.

Read equivalent <c-m-r>: When pressed on a labeled line (which should contain a string or a pathname) executes the Read command with the label as its argument.

Install equivalent <c-m-i>: When pressed on a labeled line (which should contain a string or a pathname) executes the Install command with the label as its argument.

Generate equivalent <c-m-g>: When pressed on a labeled line (which should contain an s-expression) executes the Generate command with the label as its argument.

Parse equivalent <c-m-p>: When pressed on a labeled line (which should contain a string or edge) executes the Parse command with the label as its argument.

Tree equivalent <c-m-t> : When pressed on a labeled line (which should contain an edge) executes the Tree command with the label as its argument.

11.1.2 Alternative Grammar Updating

The Macintosh implementation makes available an alternative method for updating the grammar beyond the Update command in the normal interface. The keystroke <c-m-u> used within any Fred window updates the grammar according to the selected material in the buffer.

11.1.3 Intended Usage of the Macintosh Interface

The Macintosh interface was intended to be used primarily by moving the insertion cursor around the buffer with the mouse, and using keystroke equivalents to replace explicit typing of commands. In this way, the user can interact with the system with a minimum of typing. Although not as convenient as a true window/menu-based graphical interface, this style of interaction provides many of the same advantages while retaining portability.

At the same time, the grammar being used can be opened in separate Fred windows. As changes are made to the grammar, the changed rules can be selected and <c-m-u> can be used to immediately incorporate the changes without rereading the files.

11.2 Extensions for Running Under GNU Emacs

A similar system of extensions have been developed for running the system on UNIX systems in which GNU EMACS is available. Again, functionality is added by allowing for keystroke equivalents for certain of the special commands. The GNU EMACS interface runs in an inferior LISP buffer which is created with the command M-x run-patr. This buffer works exactly like a normal edit buffer except that certain keys are rebound to make interaction with the interface easier. Among the capabilities of EMACS windows that the interface window inherits are scrolling, searching for strings, and saving of the buffer contents in a file.

11.2.1 GNU EMACS Keystroke Equivalents

Typically, the rebound keys, when pressed while the insertion cursor is anywhere on a labeled line, execute a special command taking as its argument the label of that line. The command thus invoked is also printed after the prompt at the end of the buffer, so that the interaction appears as if the command had actually been typed in. Finally, the typed command includes a comment (after the semicolon) that indicates the contents of the labeled line where the key was pressed.

Display equivalents `<line feed>`, `<return>`: The `<line feed>` and `<return>` keys, when pressed on a labeled line, execute the `Display` command taking as its argument the label of that line. When pressed on a line with a prompt but no label (i.e., one with a command that has been previously entered) the command is reinvoked. Thus, the user can reexecute a command merely by scrolling back to that command, placing the cursor anywhere on the line and hitting return.

Read equivalent `<c-m-r>`: When pressed on a labeled line (which should contain a string or a pathname) executes the `Read` command with the label as its argument.

Install equivalent `<c-m-i>`: When pressed on a labeled line (which should contain a string or a pathname) executes the `Install` command with the label as its argument.

Generate equivalent `<c-m-g>`: When pressed on a labeled line (which should contain an s-expression) executes the `Generate` command with the label as its argument.

Parse equivalent `<c-m-p>`: When pressed on a labeled line (which should contain a string or edge) executes the `Parse` command with the label as its argument.

Tree equivalent `<c-m-t>`: When pressed on a labeled line (which should contain an edge) executes the `Tree` command with the label as its argument.

11.2.2 Alternative Grammar Updating

The GNU EMACS implementation makes available an alternative method for updating the grammar beyond the `Update` command in the normal interface. The keystroke `<c-m-u>` updates the grammar according to the selected material in the buffer.

> **Known Bug: Grammar Updating Not Implemented..** The alternative grammar updating is not yet implemented.

11.2.3 Intended Usage of the GNU EMACS Interface

The GNU EMACS interface was intended to be used primarily by moving the insertion cursor around the buffer with EMACS cursor control commands or the mouse (if running directly on the Sun console rather than a terminal), and using keystroke equivalents to replace explicit typing of commands. In this way, the user can interact with the system with a minimum of typing. Although not as convenient as a true window/menu-based graphical interface, this style of interaction provides many of the same advantages while retaining portability.

At the same time, the grammar being used can be viewed in separate GNU EMACS buffers. As changes are made to the grammar, the changed rules can be selected and <c-m-u> can be used to immediately incorporate the changes without rereading the files.

Chapter 12

The CL-PATR Architecture

This chapter is a reprint of the SRI Technical Note 437, "A Uniform Architecture for Parsing and Generation". The research was supported in part by a contract with the Nippon Telephone and Telegraph Corporation.

12.1 Introduction

The use of a single grammar for both parsing and generation is an idea with a certain elegance, the desirability of which several researchers have noted. Of course, judging the correctness of such a system requires a characterization of the meaning of grammars that is independent of their use by a particular processing mechanism—that is, the formalism in which the grammars are expressed must have an abstract semantics. As a paradigm example of such a formalism, we might take any of the various logic- or unification-based grammar formalisms.

As described by Pereira and Warren [11], the parsing of strings according to the specifications of a grammar with an independent logical semantics can be thought of as the constructive proving of the string's grammaticality: parsing can be viewed as logical deduction. But, given a deductive framework that can represent the semantics of the formalism abstractly enough to be independent of processing, the generation of strings matching some criteria can equally well be thought of as a deductive process, namely, a process of constructive proof of the existence of a string that matches the criteria. The difference rests in which information is given as premises and what the goal is to be proved. This observation opens up the following possibility: not only can a single grammar be used by different processes engaged in various "directions" of processing, but one and the same language-processing architecture can be employed for processing the grammar in the various modes. In particular, parsing and generation can be viewed as two processes engaged in by a single parameterized theorem prover for the logical interpretation of the formalism.

We will discuss our current implementation of such an architecture, which is parameterized in such a way that it can be used either for parsing or generation with respect to grammars written in a particular grammar formalism which has a logical semantics, the PATR formalism. Furthermore, the architecture allows fine tuning to reflect different processing strategies, including parsing models intended to mimic psycholinguistic phenomena. This tuning allows the parsing system to operate within the same realm of efficiency as previous architectures for parsing alone, but with much greater flexibility for engaging in other processing regimes.

12.2 Language Processing as Deduction

Viewed intuitively, natural-language-utterance generation is a nondeterministic top-down process of building a phrase that conforms to certain given criteria, e.g., that the phrase be a sentence and that it convey a particular meaning. Parsing, on the other hand, is usually thought of as proceeding bottom-up in an effort to determine what properties hold of a given expression. As we have mentioned, however, both of these processes can be seen as variants of a single method for extracting certain *goal* theorems from the deductive closure of some given *premises* under the rules or constraints of the grammar. The various processes differ as to what the premises are and which goal theorems are of interest. In generation, for instance, the premises are the lexical items of the language and goal theorems are of the form "expression α is a sentence with meaning M" for some given M. In parsing, the premises are the words α of the sentence to be parsed and goal theorems are of the form "expression α is a sentence (with properties P)". In this case, α is given a priori.

This deductive view of language processing clearly presupposes an axiomatic approach to language description. Fortunately, most current linguistic theory approaches the problem of linguistic description axiomatically, and many current formalisms in use in natural-language processing, especially the logic grammar and unification-based formalisms follow this approach as well. Consequently, the results presented here will, for the most part, be applicable to any of these formalisms. We will, however, describe the system schematically—without relying on any of the particular formalisms, but using notation that schematizes an augmented context-free formalism like definite-clause grammars or PATR. We merely assume that grammars classify phrases under a possibly infinite set of structured objects, as is common in the unification-based formalisms. These structures—terms in definite-clause grammars, directed graphs in PATR, and so forth—will be referred to generically as nonterminals, since they play the role in the augmented context-free formalisms that the atomic nonterminal symbols fulfill in standard context-free grammars. We will assume that the notion of a *unifier* of such objects and *most general unifier* (*mgu*) are well defined; the symbol θ will be used for unifiers.

Following Pereira and Warren, the lemmas we will be proving from a grammar and a set of premises will include the same kind of conditional information encoded by the *items* of Earley's parsing algorithm. In Earley's algorithm, the existence of an item (or *dotted rule*) of the form

$$[N \rightarrow V_1 \cdots V_{m-1} \bullet V_m \cdots V_n, i]$$

in state set $j \geq i$ makes a claim that, for some string position $k \geq j$, the substring between i and k can be classified as an N if the substring between j and k can be decomposed into a sequence of strings classified, respectively, under $V_m, \ldots, V_n$. We will use a notation reminiscent of Pereira and Warren's[1] to emphasize the conditional nature of the claim and its independence from $V_1, \ldots, V_{m-1}$, namely,

$$[i, N \leftarrow V_m \cdots V_n, j] \quad .$$

[1] Later, in the sections containing examples of the architecture's operation, we will reintroduce $V_1, \ldots, V_{m-1}$ and the dot marker to aid readability.

12.2.1 Terminology

We digress here to introduce some terminology. If $n = 0$, then we will leave off the arrow; $[i, N, j]$ then expresses the fact that a constituent admitted as a nonterminal N occurs between positions i and j. Such items will be referred to as *nonconditional* items; if $n > 0$, the item will be considered *conditional*. In the grammars we are interested in, rules will include either all nonterminals on the right-hand side or all terminals. We can think of the former as grammar rules proper, the latter as lexical entries. Nonconditional items formed by immediate inference from a lexical entry will be called *lexical* items. For instance, if there is a grammar rule $NP \rightarrow$ *sonny*, then the item $[0, NP, 1]$ is a lexical item. A *prediction* item (or, simply, a prediction) is an item with identical start and end positions.

12.2.2 Rules of Inference

The two basic deduction steps or rules of inference we will use are—following Earley's terminology—*prediction* and *completion*.[2]

The inference rule of prediction is as follows:

$$\frac{[i, A \leftarrow BC_1 \cdots C_m, j] \qquad B' \rightarrow D_1 \cdots D_n \qquad \theta = mgu(B, B')}{[j, B'\theta \leftarrow D_1\theta \cdots D_n\theta, j]}$$

This rule corresponds to the logically valid inference consisting of instantiating a rule of the grammar as a conditional statement.[3]

The inference rule of completion is as follows:

$$\frac{[i, A \leftarrow BC_1 \cdots C_m, j] \qquad [j, B', k] \qquad \theta = mgu(B, B')}{[i, A\theta \leftarrow C_1\theta \cdots C_m\theta, k]}$$

This rule corresponds to the logically valid inference consisting of linear resolution of the conditional expression with respect to the nonconditional (unit) lemma.

12.3 Parameterizing a Theorem-Proving Architecture

This characterization of parsing as deduction should be familiar from the work of Pereira and Warren. As they have demonstrated, such a view of parsing is applicable beyond the context-free grammars by regarding the variables in the inference rules as logical variables and using unification of B and B' to solve for the most general unifier. Thus, this approach is applicable to most, if not all, of the logic grammar or unification-based formalisms.

[2] Pereira and Warren use the terms *instantiation* and *reduction* for their analogs to these rules.

[3] As noted previously [14], this rule of inference can lead to arbitrary numbers of consequents through repeated application when used with a grammar formalism with an infinite [structured] nonterminal domain. The solution proposed in that paper is to restrict the information passed from the predicting to the predicted item, corresponding to the rule

$$\frac{[i, A \leftarrow BC_1 \cdots C_m, j] \qquad B' \rightarrow D_1 \cdots D_n \qquad \theta = mgu(B\restriction\Phi, B')}{[j, B'\theta \leftarrow D_1\theta \cdots D_n\theta, j]}$$

where $B\restriction\Phi$ is a nonterminal with a bounded subset of the information of B. This inference rule is the one actually used in the implemented system. The reader is directed to the carlier paper for further discussion.

In particular, Pereira and Warren construct a parsing algorithm using a deduction strategy which mimics Earley's algorithm. We would like to generalize the approach, so that the deduction strategy (or at least portions of it) are parameters of the deduction system. The parameterization should have sufficient generality that parsers and generators with various control strategies, including Pereira and Warren's Earley deduction parser, are instances of the general architecture.

We start the development of such an architecture by considering the unrestricted use of these two basic inference rules to form the deductive closure of the premises and the goals. The exhaustive use of prediction and completion as basic inference rules does provide a complete algorithm for proving lemmas of the sort described. However, several problems immediately present themselves.

First, proofs using these inference rules can be redundant. Various combinations of proof steps will lead to the same lemmas, and combinatorial havoc may result. The traditional solution to this problem is to store lemmas in a table, i.e., the well-formed-substring table or chart in tabular parsing algorithms. In extending tabular parsing to non-context-free formalisms, the use of subsumption rather than identity in testing for redundancy of lemmas becomes necessary, and has been described elsewhere [10].

Second, deduction is a nondeterministic process and the order of searching the various paths in the proof space is critical and differs among processing tasks. We therefore parameterize the theorem-proving process by a priority function that assigns to each lemma a priority. Lemmas are then added to the table in order of their priority. As they are added, further lemmas that are consequences of the new lemma and existing ones in the table may be deduced. These are themselves assigned priorities, and so forth. The technique chosen for implementing this facet of the process is the use of an agenda structured as a priority queue to store the lemmas that have not yet been added to the table.

Finally, depending on the kind of language processing we are interested in, the premises of the problem and the types of goal lemmas we are searching for will be quite different. Therefore, we parameterize the theorem prover by an initial set of axioms to be added to the agenda and by a predicate on lemmas that determines which are to be regarded as satisfying the goal conditions on lemmas.

The structure of the architecture, then, is as follows. The processor is an agenda-based tabular theorem prover over lemmas of the sort defined above. It is parameterized by

- The initial conditions,
- A priority function on lemmas, and
- A predicate expressing the concept of a successful proof.

By varying these parameters, the processor can be used to implement language parsers and generators embodying a wide variety of control strategies.

12.4 Instances of the Architecture

We now define some examples of the use of the architecture to process with grammars.

12.4.1 Parser Instances

Consider a processor to parse a given string built by using this architecture under the following parameterization:

- The *initialization* of the agenda includes axioms for each word in the string (e.g., $[0, sonny, 1]$ and $[1, \mathit{left}, 2]$ for the sentence 'Sonny left') and an initial prediction for each rule whose left-hand side matches the start symbol of the grammar (e.g., $[0, S \leftarrow NP\ VP, 0]$).[4]
- The *priority function* orders lemmas inversely by their end position, and for lemmas with the same end position, in accordance with their addition to the agenda in a first-in-first-out manner.
- The *success criterion* is that the lemma be nonconditional, that its start and end positions be the first and last positions in the string, respectively, and that the nonterminal be the start nonterminal.[5]

Under this parameterization, the architecture mimics Earley's algorithm parsing the sentence in question, and considers successful those lemmas that represent proofs of the string's grammaticality with respect to the grammar.[6]

Alternatively, by changing the priority function, we can engender different parsing behavior. For instance, if we just order lemmas in a last-in-first-out manner (treating the agenda as a stack) we have a "greedy" parsing algorithm, which pursues parsing possibilities depth-first and backtracks when dead-ends occur.

An interesting possibility involves ordering lemmas as follows:

1. Highest priority are prediction items, then lexical items, then other conditional items, then other nonconditional items.
2. If (1) does not order items, items ending farther to the right have higher priority.
3. If (1) and (2) do not order items, items constructed from the instantiation of longer rules have higher priority.

This complex ordering implements a quite simple parsing strategy. The first condition guarantees that no nonconditional items will be added until conditional items have been computed. Thus, items corresponding to the *closure* (in the sense of LR parsing) of the nonconditional items are always added to the table. Unlike LR parsing, however, the closure here is computed at run time rather than being precompiled. The last two conditions correspond to disambiguation of shift/reduce and reduce/reduce conflicts in LR parsing respectively. The former requires that shifts be preferred to reductions, the latter that longer reductions receive preference.

In sum, this ordering strategy implements a sentence-disambiguation parsing method that has previously been argued [13] to model certain psycholinguistic phenomena—for instance, right association and minimal attachment [3]. However, unlike the earlier characterization in terms of LR disambiguation, this mechanism can be used for arbitrary logic or unification-based grammars, not just context-free grammars. Furthermore, the architecture allows for fine tuning of the disambiguation strategy beyond

[4] For formalisms with complex structured nonterminals, the start "symbol" need only be unifiable with the left-hand-side nonterminal. That is, if S is the start nonterminal and $S' \rightarrow C_1 \cdots C_n$ is a rule and $\theta = mgu(S, S')$, then $[0, S'\theta \leftarrow C_1\theta \cdots C_n\theta, 0]$ is an initial prediction.

[5] Again, for formalisms with complex structured nonterminals, the start symbol need only subsume the item's nonterminal.

[6] Assuming that the prediction inference rule uses the restriction mechanism, the architecture actually mimics the variant of Earley's algorithm previously described in [14].

that described in earlier work. Finally, the strategy is complete, allowing "backtracking" if earlier proof paths lead to a dead end.[7]

12.4.2 A Parsing Example

As a demonstration of the architecture used as a parser, we consider the Earley and backtracking-LR instances in parsing the ambiguous sentence:

> Castillo said Sonny was shot yesterday.

Since the operation of the architecture as a parser is quite similar to that of previous parsers for unification-based formalisms, we will only highlight a few crucial steps in the process.

The Earley parser assigns higher priority to items ending earlier in the sentence. The highest-priority initialization items are added first.[8]

$$\begin{array}{ll} [0, S \rightarrow \bullet NP\ VP, 0] & \text{‘’} \\ [0, NP \rightarrow \mathit{castillo} \bullet, 1] & \text{‘Castillo’} \end{array}$$

By Completion, the item

$$\begin{array}{ll} [0, S \rightarrow NP \bullet VP, 1] & \text{‘Castillo’} \end{array}$$

is generated, which in turn predicts

$$\begin{array}{ll} [1, VP \rightarrow \bullet VP\ X, 1] & \text{‘’} \\ [1, VP \rightarrow \bullet V, 1] & \text{‘’} \\ [1, VP \rightarrow \bullet VP\ AdvP, 1] & \text{‘’} \end{array}$$

The highest-priority item remaining on the agenda is the initial item

$$\begin{array}{ll} [1, V \rightarrow \mathit{said} \bullet, 2] & \text{‘said’} \end{array}$$

Processing progresses in this manner, performing all operations at a string position before moving on to the next position until the final position is reached, at which point the final initial item corresponding to the word 'yesterday' is added. The following flurry of items is generated by completion.[9]

[7] Modeling of an incomplete version of the shift-reduce technique is also possible. The simplest method, however, involves eliminating the chart completing, and mimicking closure, shift, and reduction operations as operations on LR states (sets of items) directly. Though this method is not a straightforward instantiation of the architecture of Section 12.3 (since the chart is replaced by separate state sets), we have implemented a parser using much of the same technology described here and have successfully modeled the garden path phenomena that rely on the incompleteness of the shift-reduce technique.

[8] The format used in displaying these items reverts to one similar to Earley's algorithm, with a dot marking the position in the rule covered by the string generated so far, so as to describe more clearly the portion of each grammar rule used. In addition, the string actually parsed or generated is given in single quotes after the item for convenience.

[9] The four instances of 'said Sonny was shot yesterday' arise because of lexical ambiguity in the verb 'said' and adverbial-attachment ambiguity. Only the finite version of 'said' is used in forming the final sentence.

	...	
	$[5, AdvP \rightarrow yesterday \bullet, 6]$	'yesterday'
(2)	$[1, VP \rightarrow VP\ AdvP \bullet, 6]$	'said Sonny was shot yesterday'
(3)	$[3, VP \rightarrow VP\ AdvP \bullet, 6]$	'was shot yesterday'
	$[4, VP \rightarrow VP\ AdvP \bullet, 6]$	'shot yesterday'
	$[1, VP \rightarrow VP \bullet AdvP, 6]$	'said Sonny was shot yesterday'
(4)	$[0, S \rightarrow NP\ VP \bullet, 6]$	'Castillo said Sonny was shot yesterday'
	$[3, VP \rightarrow VP \bullet AdvP, 6]$	'was shot yesterday'
(5)	$[2, S \rightarrow NP\ VP \bullet, 6]$	'Sonny was shot yesterday'
	$[4, VP \rightarrow VP \bullet AdvP, 6]$	'shot yesterday'
(6)	$[1, VP \rightarrow VP\ S \bullet, 6]$	'said Sonny was shot yesterday'
	$[1, VP \rightarrow VP \bullet AdvP, 6]$	'said Sonny was shot yesterday'
(7)	$[0, S \rightarrow NP\ VP \bullet, 6]$	'Castillo said Sonny was shot yesterday'

Note that the first full parse found (4) is derived from the high attachment of the word 'yesterday' (2) (which is composed from (1) directly), the second (7) from the low attachment (6) (derived from (5), which is derived in turn from (3)).

By comparison, the shift-reduce parser generates exactly the same items as the Earley parser, but in a different order. The crucial ordering difference occurs in the following generated items:

	...	
(1)	$[5, AdvP \rightarrow yesterday \bullet, 6]$	'yesterday'
	...	
(3)	$[3, VP \rightarrow VP\ AdvP \bullet, 6]$	'was shot yesterday'
	$[3, VP \rightarrow VP \bullet AdvP, 6]$	'was shot yesterday'
(5)	$[2, S \rightarrow NP\ VP \bullet, 6]$	'Sonny was shot yesterday'
(6)	$[1, VP \rightarrow VP\ S \bullet, 6]$	'said Sonny was shot yesterday'
	$[1, VP \rightarrow VP \bullet AdvP, 6]$	'said Sonny was shot yesterday'
(7)	$[0, S \rightarrow NP\ VP \bullet, 6]$	'Castillo said Sonny was shot yesterday'
(8)	$[2, S \rightarrow NP\ VP \bullet, 5]$	'Sonny was shot'
	$[1, VP \rightarrow VP\ S \bullet, 5]$	'said Sonny was shot'
	$[1, VP \rightarrow VP \bullet AdvP, 5]$	'said Sonny was shot'
(2)	$[1, VP \rightarrow VP\ AdvP \bullet, 6]$	'said Sonny was shot yesterday'
	$[1, VP \rightarrow VP \bullet AdvP, 6]$	'said Sonny was shot yesterday'
(4)	$[0, S \rightarrow NP\ VP \bullet, 6]$	'Castillo said Sonny was shot yesterday'
	...	

Note that the reading of the sentence (7) with the low attachment of the adverb—the so-called "right association" reading—is generated before the reading with the higher attachment (4), in accordance with certain psycholinguistic results [3]. This is because item (3) has higher priority than item (8), since (3) corresponds to the shifting of the word 'yesterday' and (8) to the reduction of an *NP* and *VP* to *S*. The second clause of the priority definition orders such shifts before reductions. In summary, this instance of the architecture develops parses in right-association/minimal-attachment preference order.

12.4.3 Generator Instances

As a final example of the use of this architecture, we consider using it for generation by changing the initialization condition as follows:

- The *initialization* of the agenda includes axioms for each word in the lexicon at each position (e.g., $[0, sonny, 1]$ and $[0, \mathit{left}, 1]$ and $[1, \mathit{left}, 2]$, and so on) and an initial prediction for each rule whose left-hand side is the start symbol of the grammar (e.g., $[0, S \leftarrow NP\ VP, 0]$). In the case of a grammar formalism with more complex information structures as nonterminals, e.g., definite-clause grammars, the "start symbol" might include information about, say, the meaning of the sentence to be generated. We will refer to this as the *goal meaning.*
- The *success criterion* is that the nonterminal be subsumed by the start nonterminal (and therefore have the appropriate meaning).

Under this parameterization, the architecture serves as a generator for the grammar, generating sentences with the intended meaning. By changing the priority function, the order in which possibilities are pursued in generation can be controlled, thereby modeling depth-first strategies, breadth-first strategies, and so forth.

Of course, as described, this approach to generation is sorely inadequate for several reasons. First, it requires that we initially insert the entire lexicon into the agenda at arbitrary numbers of string positions. Not only is it infeasible to insert the lexicon so many times (indeed, even once is too much) but it also leads to massive redundancy in generation. The same phrase may be generated starting at many different positions. For parsing, keeping track of which positions phrases occur at is advantageous; for generation, once a phrase is generated, we want to be able to use it in a variety of places.

A simple solution to this problem is to ignore the string positions in the generation process. This can be done by positioning all lemmas at a single position. Thus we need insert the lexicon only once, each word being inserted at the single position, e.g., $[0, sonny, 0]$.

Although this simplifies the set of initial items, by eliminating indexing based on string position we remove the feature of tabular parsing methods such as Earley's algorithm that makes parsing reasonably efficient. The generation behavior exhibited is therefore not goal-directed; once the lexicon is inserted many phrases might be built that could not contribute in any way to a sentence with the appropriate meaning. In order to direct the behavior of the generator towards a goal meaning, we can modify the priority function so that it is partial; not every item will be assigned a priority and those that are not will never be added to the table (or agenda) at all. The filter we have been using assigns priorities only to items that might contribute semantically to the goal meaning. In particular, *the meaning associated with the item must subsume some portion of the goal meaning.*[10] This technique, a sort of indexing on meaning, replaces the indexing on string position that is more appropriate for parsing than generation.

As a rule, filtering the items by making the priority function partial can lead to incompleteness of the parsing or generation process.[11] However, the subsumption filter described here for use in generation does not yield incompleteness of the generation algorithm under one assumption about the grammar, which we might call *semantic monotonicity.* A grammar is semantically monotonic if, for every phrase admitted by the grammar, the semantic structure of each immediate subphrase subsumes some portion of the semantic structure of the entire phrase. Under this condition, items which do not subsume part of the goal meaning can be safely ignored, since any phrase built from them will also not subsume part of

[10] Since the success criterion requires that a successful item be subsumed by the start nonterminal and the priority filter requires that a successful item's semantics subsume the start nonterminal's semantics, it follows that successful items match the start symbol exactly in semantic information; overgeneration in this sense is not a problem.

[11] Indeed, we might want such incompleteness for certain cases of psycholinguistically motivated parsing models such as the simulated LR model described above.

the goal meaning and thus will fail to satisfy the success criterion. Thus the question of completeness of the algorithm depends on an easily detectable property of the grammar. Semantic monotonicity is, by intention, a property of the particular grammar we have been using.

12.4.4 A Generation Example

As an example of the generation process, we consider the generation of a sentence with a goal logical form

$$\mathit{passionately}(\mathit{love}(\mathit{sonny}, \mathit{kait})) \quad .$$

The example was run using a toy grammar that placed subcategorization information in the lexicon, as in the style of analysis of head-driven phrase-structure grammar (HPSG). The grammar ignored tense and aspect information, so that, for instance, auxiliary verbs merely identified their own semantics with that of their postverbal complement.[12]

The initial items included the following:

(1)	$[0, NP \rightarrow \mathit{sonny} \bullet, 0]$	'Sonny'
(2)	$[0, NP \rightarrow \mathit{kait} \bullet, 0]$	'Kait'
	$[0, V \rightarrow \mathit{to} \bullet, 0]$	'to'
	$[0, V \rightarrow \mathit{was} \bullet, 0]$	'was'
	$[0, V \rightarrow \mathit{were} \bullet, 0]$	'were'
	...	
	$[0, V \rightarrow \mathit{loves} \bullet, 0]$	'loves'
	$[0, V \rightarrow \mathit{love} \bullet, 0]$	'love'
	$[0, V \rightarrow \mathit{loved} \bullet, 0]$	'loved'
	$[0, AdvP \rightarrow \mathit{passionately} \bullet, 0]$	'passionately'
(3)	$[0, S \rightarrow \bullet NP\ VP, 0]$	''

Note that auxiliary verbs were included, as the semantic structure of an auxiliary is merely a variable (coiindexed with the semantic structure of its postverbal complement), which subsumes some part (in fact, every part) of the goal logical form.[13] Similarly, the noun phrases 'Sonny' and 'Kait' (with semantics *sonny* and *kait*, respectively) are added, as these logical forms each subsume the respective innermost arguments of the goal logical form. Several forms of the verb 'love' are considered, again because the semantics in this grammar makes no tense/aspect distinctions. But no other proper nouns or verbs are considered (although the lexicon that was used contained them) as they do not pass the semantic filter.

The noun phrase 'Sonny' can be used as the subject of the sentence by combining items (1) and (3) yielding

(4) $[0, S \rightarrow NP \bullet VP, 0]$ 'Sonny' .

(The corresponding item with the subject 'Kait' will be generated later.) Prediction yields the following chain of items.

[12] For reference, the grammar is similar in spirit to the third sample grammar in [15].

[13] It holds in general that closed-class lexical items—case-marking prepositions, function verbs, etc.—are uniformly considered initial items for purposes of generation because of their vestigial semantics. This is as desired, and follows from the operation of semantic filtering, rather than from any ad hoc techniques.

$[0, VP \rightarrow \bullet\, VP\ AdvP, 0]$	‘’
$[0, VP \rightarrow \bullet\, V, 0]$	‘’

The various verbs, including the forms of ‘love’, can complete this latter item.

	$[0, VP \rightarrow V \bullet, 0]$	‘to’
	$[0, VP \rightarrow V \bullet, 0]$	‘is’
	$[0, VP \rightarrow V \bullet, 0]$	‘was’
	$[0, VP \rightarrow V \bullet, 0]$	‘were’
(5)	$[0, VP \rightarrow V \bullet, 0]$	‘loves’
	$[0, VP \rightarrow V \bullet, 0]$	‘love’
	$[0, VP \rightarrow V \bullet, 0]$	‘love’
	$[0, VP \rightarrow V \bullet, 0]$	‘loved’

The passive form of the verb ‘loved’ might be combined with the adverb.

$[0, VP \rightarrow VP \bullet AdvP, 0]$	‘loved’
$[0, VP \rightarrow VP\ AdvP \bullet, 0]$	‘loved passionately’
...	

The latter item might be used in a sentence ‘Kait was loved passionately.’ This sentence will eventually be generated but will fail the success criterion because its semantics is insufficiently instantiated.

Prediction from item (4) also yields the rule for adding complements to a verb phrase.

$[0, VP \rightarrow \bullet\, VP\ X, 0]$	‘’

Eventually, this item is completed with items (5) and (2).

...	
$[0, VP \rightarrow VP \bullet NP, 0]$	‘loves’
$[0, VP \rightarrow VP\ NP \bullet, 0]$	‘loves Kait’

The remaining items generated are

$[0, VP \rightarrow VP \bullet AdvP, 0]$	‘loves Kait’
$[0, VP \rightarrow VP\ AdvP \bullet, 0]$	‘loves Kait passionately’
$[0, S \rightarrow NP\ VP \bullet, 0]$	‘Sonny loves Kait passionately’

This final item matches the success criterion, and is the only such item. Therefore, the sentence ‘Sonny loves Kait passionately’ is generated for the logical form *passionately*(*love*(*sonny*, *kait*)).

Looking over the generation process, the set of phrases actively explored by the generator included ‘Kate is loved’, ‘Kate is loved passionately’, ‘were loved passionately’ and similar passive constructions, ‘Sonny loves Kait’, and various subphrases of these. However, other phrases composed of the same words, such as ‘Kait loves Kait’, ‘Sonny is loved’, and so forth, are eliminated by the semantics filter. Thus, the the generation process is, on the whole, quite goal-directed; the subphrases considered in the generation process are “reasonable”.

12.5 The Implementation

The architecture described above has been implemented for the PATR grammar formalism in a manner reminiscent of object-oriented programming. Instances of the architecture are built as follows. A general-purpose processor-building function, taking a priority function and success criterion function as arguments, returns an object that corresponds to the architecture instance. The object can be sent initialization items as arbitrary lemmas of the usual form. Whenever a successful lemma is constructed (according to the success criterion) it is returned, along with a continuation function that can be called if further solutions are needed. No processing is done after a successful lemma has been proved unless further solutions are requested.

Using this implementation, we have built instances of the architecture for Earley parsing and the other parsing variants described in this paper, including the shift/reduce simulator. In addition, a generator was built that is complete for semantically monotonic grammars. It is interesting to note that the generator is more than an order of magnitude faster than our original PATR generator, which worked purely by top-down depth-first backtracking search, that is, following the Prolog search strategy.

The implementation is in Common Lisp and runs on Symbolics 3600, Sun, and Macintosh computers. It is used (in conjunction with a more extensive grammar) as the generation component of the GENESYS system for utterance planning and production.

12.6 Precursors

Perhaps the clearest espousal of the notion of grammar reversability was made by Kay [9], whose research into functional grammar has been motivated by the desire to "make it possible to generate and analyze sentences with the same grammar." Other researchers have also put such ideas into effect. Jacobs's PHRED system [6] "operates from a declarative knowledge base of linguistic knowledge, common to that used by PHRAN", an analyzer for so-called phrasal grammars. Jacobs notes that other systems have shared at least part of the linguistic information for parsing and generation; for instance, the HAM-ANS [17] and VIE-LANG [16] systems utilize the same lexical information for both tasks. Kasper has used a system for parsing grammars in a unification-based formalism (SRI's Z-PATR system) to parse sentences with respect to the large ISI NIGEL grammar, which had been previously used for generation alone.

Nonetheless, all of these systems rely on often radically different architectures for the two processes. Precedent for using a single architecture for both tasks is much more difficult to find. The germ of the idea can be found in the General Syntactic Processor (GSP) designed for the MIND system at Rand. Kaplan and Kay proposed use of the GSP for parsing with respect to augmented transition networks and generation by transformational grammars [7]. However, detailed implementation was apparently never carried out. In any case, although the proposal involved using the same architecture, different formalisms (and hence grammars) were presupposed for the two tasks. Running a definite-clause grammar (DCG) "backwards" has been proposed previously, although the normal Prolog execution mechanism renders such a technique unusable in practice. However, alternative execution models might make the practice feasible. As mentioned above, the technique described here is just such an execution model, and is directly related to the Earley deduction model of Pereira and Warren [11]. Hasida and Isizaki [5] present another method for generating and analyzing using a DCG-like formalism, which they call dependency propagation. The technique seems to entail using dataflow dependencies implicit in the grammar to control processing in a coroutining manner. The implementation status of their method and its practical utility are as yet unclear.

The use of an agenda and scheduling schemes to allow varying the control structure of a parser also finds precedent in the work of Kaplan [7] and Kay [8]. Kay's "powerful parser" and the GSP both employed an agenda mechanism to control additions to the chart. However, the "tasks" placed on the agenda were at the same time more powerful (corresponding to unconstrained rewrite rules) and more procedural (allowing register operations and other procedural constructs). This work merely applies the notion in the context of the simple declarative formalisms presupposed, and provides it with a logical foundation on which a proof of correctness can be developed.[14] Because the formalisms are simpler, the agenda need only keep track of one type of task: addition of a chart item.

12.7 Further Research

Perhaps the most immediate problem raised by the methodology for generation introduced in this paper is the strong requirement of semantic monotonicity, which serves as yet another instance of the straitjacket of compositionality. The semantic-monotonicity constraint allows the goal logical form to be systematically decomposed so that a dynamic-programming generation process can be indexed by the parts of the decomposition (the subformulas), just as the constraint of string concatenation in context-free grammars allows a goal sentence to be systematically decomposed so that a dynamic-programming parsing process can be indexed by the subparts of that decomposition (the substrings). And just as the concatenation restriction of context-free grammars can be problematic, so can the restriction of semantic monotonicity. Finding a weaker constraint on grammars that still allows efficient processing is thus an important research objective.

Even with the semantic-monotonicity constraint, the process of indexing by the highly structured logical forms is not nearly so efficient as indexing by simple integer string positions. Partial match retrieval or similar techniques from the Prolog literature might be useful here.

Nothing has been said about the important problem of guaranteeing that the syntactic and semantic goal properties will actually be realized in the sentence generated. The success criterion for generation described here would require that the logical form for the sentence generated be identical to the goal logical form. However, there is no guarantee that the other properties of the sentence include those of the goal; only compatibility is guaranteed. Researchers at the University of Stuttgart have proposed solutions to this problem based on the type of existential constraint found in lexical-functional grammar. We expect that their methods might be applicable within the presented architecture.

Finally, on a more pessimistic note, we turn to a widespread problem in all systems for automatic generation of natural language, which Appelt [2] has discussed under the rubric "the problem of logical-form equivalence". The mapping from logical forms to natural-language expressions is in general many-to-one. For instance, the logical forms $red(x) \wedge ball(x)$ and $ball(x) \wedge red(x)$ might both be realized as the nominal 'red ball'. However, most systems for describing the string-LF relation declaratively do so by assigning a minimal set of logical forms to each string, with each logical form standing proxy for all its logical equivalents. The situation is represented graphically as Figure 1.

The problem is complicated further in that, strictly speaking, the class of equivalent logical forms from the standpoint of generation is not really closed under logical equivalence. Instead, what is actually

[14] Such a proof is currently in preparation.

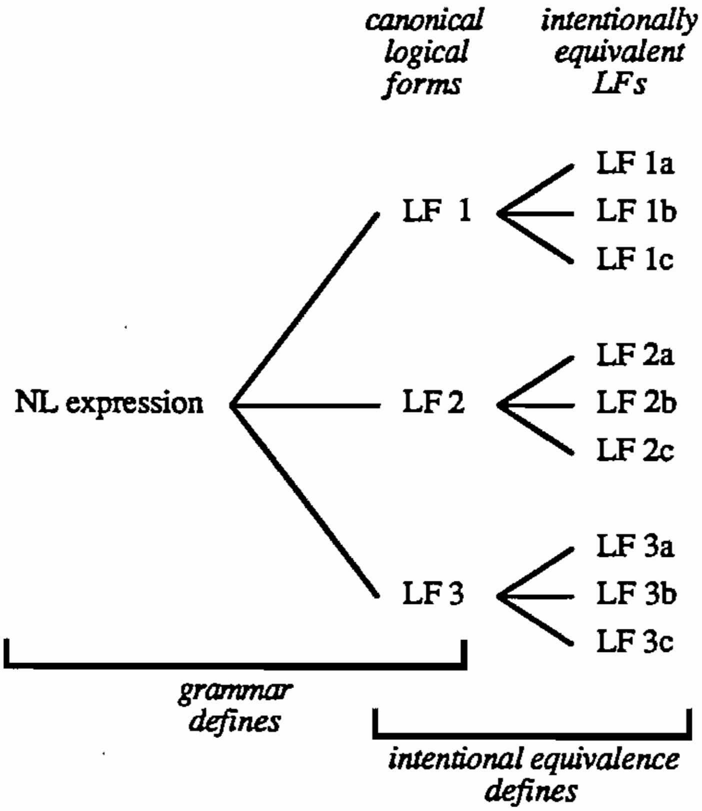


Figure 12.1: Canonical Logical Forms

required is a finer-grained notion of ***intentional equivalence***, under which, for instance, p and $p \wedge (q \vee \neg q)$ would not necessarily be intentionally equivalent; they might correspond to different utterances, one about p only, the other about both p and q.

In such a system, merely using the grammar per se to drive generation (as we do here) allows for the generation of strings from only a subset of the logical-form expressions that have natural-language relata, that is, LF1, LF2, and LF3 in the figure. We might call these the *canonical logical forms.* Even if the grammar is reversible, the problem remains, because logical equivalence is in general not computable. And even in restricted cases in which it is computable, for instance a logic with a confluence property under which all logically equivalent expressions do have a canonical form, the problem is not solved *unless* the notion of canonical form implicit in the logic corresponds exactly to that of the natural-language grammar.

It should be noted that this kind of problem is quite deep. Any system that represents meanings in some way (not necessarily with logical expressions) must face a correlate of this problem. Furthermore, although the issue impinges on syntax because it arises in the realm of grammar, it is primarily a semantic problem, as we will shortly see.

There are two apparent possible approaches to a resolution of this problem. We might use a logic in which logical equivalence classes of expressions are all trivial, that is, any two distinct expressions mean something different. In such a logic, there are no artifactual syntactic remnants in the syntax of the logical language. Furthermore, expressions of the logic must be relatable to expressions of the natural language with a reversible grammar. Alternatively, we could use a logic for which canonical forms, corresponding exactly to the natural language grammar's logical forms, do exist.

The difference between the two approaches is only an apparent one, for in the latter case the equivalence classes of logical forms can be identified as logical forms of a new logical language with no artifactual distinctions. Thus, the second case reduces to the first. The central problem in either case, then, is discovery of a logical language which exactly and uniquely represents all the meaning distinctions of natural language utterances and no others. This holy grail, of course, is just the goal of knowledge representation for natural language in general; we are unlikely to be able to rely on a full solution soon.

However, by looking at approximations of this goal, suitably adapted to the particular problems of generation, we can hope to achieve some progress. In the case of approximations, it does not hold that the two methodologies reduce one to another; in this case, we feel that the second approach—designing a logical language that approximates in its canonical forms those needed for grammatical applications—is more likely to yield good incremental results.

Chapter 13

Sample Symbolics 3600 System Files

Symbolics system software requires that certain files reside in the sys:site; directory before performing any operations on a user-defined system, such as restoring it from a distribution tape or loading it. Thus, the following four files (two each for the CL-PATR and ParEn systems) must be set up in this directory before the Symbolics 3600 distribution tape is restored.

```
sys:site;cl-patr.system
sys:site;cl-patr.translations
sys:site;paren.system
sys:site;paren.translations
```

Below, we give sample contents for these four files.

13.1 Contents of sys:site;cl-patr.system

```
;;; -*- Mode: LISP; Package: USER -*-

(fs:make-logical-pathname-host "cl-patr")

(sct:set-system-source-file "cl-patr"
    "cl-patr:patr;patr-system-3600.lisp")
```

13.2 Contents of sys:site;cl-patr.translations

```
;;; -*- Mode: LISP; Package: USER -*-
```

```
(fs:set-logical-pathname-host "cl-patr"
      :physical-host "hostname"
      :translations
      '(("**;" ">cl-patr>**>")))
```

13.3 Contents of sys:site;paren.system

```
;;; -*- Mode: LISP; Package: USER -*-

(fs:make-logical-pathname-host "paren")

(sct:set-system-source-file "paren"
    "paren:paren;lr-system-3600")
```

13.4 Contents of sys:site;paren.translations

```
;;; -*- Mode: LISP; Package: USER -*-

(fs:set-logical-pathname-host "paren"
      :physical-host "hostname"
      :translations
      '(("paren;**;" ">cl-patr>paren>**>")))
```

Bibliography

[1] James Allen. *Natural Language Understanding.* Benjamin/Cummings, Menlo Park, California, 1987.

[2] Douglas E. Appelt. Bidirectional grammars and the design of natural language generation systems. In *Theoretical Issues in Natural Language Processing—3*, pages 185–191, Las Cruces, New Mexico, 7–9 January 1987. New Mexico State University.

[3] Lyn Frazier and Janet Dean Fodor. The sausage machine: a new two-stage parsing model. *Cognition*, 6:291–325, 1978.

[4] Jr. Guy L. Steele. COMMON LISP*: The Language.* Digital Press, 1984.

[5] Kôiti Hasida and Syun Isizaki. Dependency propagation: a unified theory of sentence comprehension and generation. In *Proceedings of AAAI-87*, pages 664–670, Seattle, Washington, 13–17 July 1987.

[6] Paul S. Jacobs. PHRED: a generator for natural language interfaces. *Computational Linguistics*, 11(4):219–242, October–December 1985.

[7] Ronald M. Kaplan. A general syntactic processor. In Randall Rustin, editor, *Natural Language Processing*, pages 193–241. Algorithmics Press, New York, 1973.

[8] Martin Kay. Experiments with a powerful parser. In *Proceedings of the Second International Conference on Computational Linguistics*, August 1967.

[9] Martin Kay. Syntactic processing and functional sentence perspective. In *Theoretical Issues in Natural Language Processing—Supplement to the Proceedings*, pages 12–15, Cambridge, Massachusetts, 10–13 June 1975.

[10] Fernando C. N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis*, volume 10 of *CSLI Lecture Note Series.* Center for the Study of Language and Information, Stanford, California, 1987.

[11] Fernando C. N. Pereira and David H. D. Warren. Parsing as deduction. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, Cambridge, Massachusetts, 15–17 June 1983. Massachusetts Institute of Technology.

[12] Stanley J. Rosenschein and Stuart M. Shieber. Translating English into logical form. In *Proceedings of the 20th Annual Meeting of the Association for Computational Linguistics*, pages 1–8, Toronto, Ontario, Canada, 16-18 June 1982. University of Toronto.

[13] Stuart M. Shieber. Sentence disambiguation by a shift-reduce parsing technique. In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics*, pages 113–118, Cambridge, Massachusetts, 15–17 June 1983. Massachusetts Institute of Technology.

[14] Stuart M. Shieber. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics*, pages 145–152, Chicago, Illinois, 8–12 July 1985. University of Chicago.

[15] Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar*, volume 4 of *CSLI Lecture Note Series*. Center for the Study of Language and Information, Stanford, California, 1986.

[16] Ingeborg Steinacker and Ernst Buchberger. Relating syntax and semantics: the syntactico-semantic lexicon of the system VIE-LANG. In *Proceedings of the First Conference of the European Chapter of the Association for Computational Linguistics*, pages 96–100, Pisa, Italy, 1–2 September 1983.

[17] Wolfgang Wahlster, Heinz Marburger, Anthony Jameson, and Stephan Busemann. Overanswering yes-no questions: Extended responses in a natural language interface to a vision system. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 643–646, Karlsruhe, West Germany, 8–12 August 1983.

[18] Terry Winograd. *Language as a Cognitive Process—Volume 1: Syntax*. Addison-Wesley, Reading, Massachusetts, 1983.