CMU-CS-85-185

DYPAR-I: Tutorial and Reference Manual

December 1985

Mark Boggs, Jaime Carbonell, Marion Kee and Ira Monarch
Department of Computer Science
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

CMU-CS-85-185

DYPAR-I: Tutorial and Reference Manual

December 1985

Mark Boggs, Jaime Carbonell, Marion Kee and Ira Monarch
Department of Computer Science
Carnegie-Mellon University
Schenley Park
Pittsburgh, PA 15213

This research was supported in part by a grant from Digital Equipment Corporation

Table of Contents

List of Tables

DYPAR Parsing System

Exercises

One INTRODUCTION

1.1. What is DYPAR-I?

DYPAR-I is a rule-based natural language interpreter, adaptable to many limited-domain applications such as data-base query, command intepretation for software systems, and simple expert system command and query tasks. DYPAR-I is *not* designed to parse full, unrestricted natural language text; such systems do not yet exist. Rather, its primary objective is to serve as a high level programming tool making it possible for anyone to write a grammar for a specific application. The tutorial portion of this manual introduces the basic grammar writing skills, which are subsequently illustrated in a fully worked-out example grammar that serves as a natural language query and update interface to a simple semantic-network data base. Appendix B, a reference section includes all the basic and advanced operators and features of DYPAR-I.

DYPAR-I is implemented in FRANZ LISP, and runs under VAX UNIX or VMS/EUNICE. A COMMON LISP implementation has just been completed. DYPAR-I is a proper subset of other, more powerful parsers of the DYPAR family, culminating in the as-yet-unreleased DYPAR-IV, under development at Carnegie-Mellon University. DYPAR-I with a fair-sized grammar interprets sentences in real time on a lightly loaded VAX-11-780.

1.2. Objectives of Natural Language Interfaces

Human-computer interaction plays an increasingly significant role in the design of software systems, be they AI-based expert systems, packaged utilities, data base managers, or interactive help-systems. Natural language provides a convivial interface requiring little or no user training, providing flexibility of expression, and allowing the user to focus his or her cognitive energy towards achieving the underlying task. Unfortunately, computer interpretation of natural language has proven a difficult task, one which has in its general form resisted a comprehensive solution. Restricted forms of natural language, however, have proven tractable to computational approaches, such as the methods incorporated in DYPAR-I.

The central objective in most natural language interfaces is to provide sufficient coverage of those aspects of natural language pertinent to the range of tasks performed by the underlying system, rather than achieving full coverage of all natural language. Additional design objectives include efficient performance, some measure of portability to new domains, conciseness of expression for grammar rules and extensibility of the grammar. DYPAR-I strives to achieve these objectives, and fulfills some of them much better than previous language interpretation systems.

There are basically three types of people who work with natural language interfaces: 1) The designers and implementers of the system itself, 2) The grammar-writers and system maintainers at the installation site, and 3) The end users. As we noted earlier, it takes virtually no special training to be an end user of a language interface; therein lies its primary virtue. Designing and implementing a new natural language interpreter is a difficult time-consuming process that requires a high level of expertise. However, writing a grammar for a new domain or extending the capabilities in an existing domain requires, first and foremost, the ability to write and adapt grammar rules. It is precisely this skill that the present document is designed to impart. We strongly recommend that the reader obtain hands-on experience by working through the examples in the tutorial and typing some of them into the system, as well as experimenting with his or her own grammatical constructs. It is a fantastic feeling when a grammar is completed and the system actually takes instructions in natural language!

1.3. About This Manual

This document is a combination tutorial introduction to DYPAR-I and reference manual. To this end, there are some remarks in the tutorial section that are meant for those with backgrounds in LISP programming and other technical areas. However, the presence of such material should not alarm anyone who is not familiar with it; it is simply there for the benefit of the more experienced. We are continuing to modify the tutorial to make it more understandable to those who do not have a computer science background. On the other hand, to avoid unnecessary distraction for those who do know LISP, many explanatory comments on function names, etc., are contained in footnotes instead of being incorporated into the primary text. It is strongly suggested that serious grammar-writers who lack a knowledge of LISP acquire some familiarity with it. Being able to understand how and why DYPAR does what it does will greatly improve one's comprehension of the scope of DYPAR's abilities, as well as making it easier to write working grammars. In other words, the footnotes and other explanations of LISP functions are intended to be introductory in nature, not a blanket substitute for knowing LISP.

In general, Chapters Two and Three of the manual make up the tutorial section, and the rest comprise the reference manual. The tutorial is essentially self-contained, explaining each type of grammatical construction you can build, and showing how it is used in recognizing English input. The advanced features discussed in the reference manual should be used only after attaining a basic understanding with the tutorial part of this document. Some knowledge of LISP on the reader's part is assumed in the reference manual, and in general grammar debugging is made easier by being able to understand some of the inner workings of DYPAR-I.

1.4. The Parsing Strategies of DYPAR-I

DYPAR-I is a three-strategy parser. Those readers familiar with the natural language literature will recognize these strategies as recursive pattern-matching, semantic grammar interpretation and string transformations. Syntactic generalities can be captured by the transformations. The combination of recursive pattern matching with semantic grammar constructs yields a more powerful mechanism than either method provides in isolation.[1]

Wait! Don't put the tutorial down yet! That last paragraph was meant only to set some context for the computational linguists among you. You can build grammars and use DYPAR-I without worrying about the technical jargon, as we will use a minimal amount of it throughout this document. And, you need not be a linguist to devise your own grammars and encode them as DYPAR-I rules.

1.5. Disclaimer and Future Releases

For future releases of DYPAR-I, and newer versions of this document send ARPA mail to Marion.Kee@CAD.CS.CMU.EDU, or write to:

```
Marion Kee
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213
```

Since DYPAR is under active development, neither the authors nor Carnegie-Mellon University guarantee complete accuracy, performance, or compatibility of any DYPAR release or this manual.

[1] The newer DYPAR-II and DYPAR-IV systems add recursive case-frame instantiation at the sentential and noun-phrase levels.

Two RULES AND OPERATORS

The Basic Tools

The grammar writer instructs DYPAR to recognize and interpret English sentences by building grammars. Each grammar is a set of rules interpreted by DYPAR that enable the system to manipulate a limited set of English sentences.[2] The grammar writer decides which classes of sentences he or she needs to have DYPAR recognize, and then builds rules that recognize these and others like them. Three distinct types of rules are used in building a grammar:

- Top-level (Production) Rules

- Rewrite (Nonterminal) Rules

- Transformation Rules

Top-level and Transformation rules consist of sentence templates and associated actions that are carried out if the template matches the user's input. Nonterminal rules are the building blocks which make up the templates. Before examining complete rule structures, let us look at the basic component of all the rules: the *pattern*.

2.1. Patterns and Operators

Most of the natural language parsing systems in existence make some use of pattern matching. DYPAR makes rather substantial use of recursive[3] pattern matching. Each DYPAR rule consists of a pattern and an action. If the pattern is successfully matched, the action-side of the rule is executed. In this section we will focus on the structure and creation of recursive patterns, and in the next section we will show you how to write rules using these patterns.

Patterns are lists, which are composed of *terminal symbols* (i.e. words, numbers, and punctuation characters), special operators, and other patterns. Because the pattern is a list, it is enclosed in a set of parentheses:

```
(PATTERN)
```

The DYPAR pattern matcher takes the patterns provided by the grammar writer and matches them one at a

[2] We will use the word "sentence" rather loosely in this document to mean whatever the person using the system might type, i.e. any set of words, phrases, utterances, or full sentences which have meaning within the application domain. Also, DYPAR is not limited to use with English. With minor modifications, DYPAR-I can be used with most Indo-European languages which employ the Roman alphabet.

[3] Recursion is a method of doing something repeatedly. An understanding of recursion is not strictly necessary for the production of DYPAR grammars, but would be helpful. A recursive procedure goes about solving a problem by simplifying it a little and then calling itself on the simplified version of the problem. The call to itself is *embedded*; i.e., the procedure is not over and done with when it calls itself again. (In this way it differs from a procedure that is merely repeated over and over again, a technique known as *iteration*). A recursive function is "on hold" until it receives the results returned to it by this copy of itself. It may then perform additional steps using both its own results and those returned by the embedded copy. Of course, the copy will also call itself, and so on. Both recursive and iterative procedures include conditions that, when satisfied, will cause the operation to stop repeating itself. Otherwise they would go on repeating *ad infinitum*. As applied to pattern matching, recursion allows patterns themselves to contain other patterns embedded in them: the pattern matcher will simply call itself again for every level of the pattern.

time, in the order given, against the user's input.[4] The special operators cause the pattern matcher to behave in subtly different ways (other than straight word-for-word matching of input to terminal symbols in a pattern), thus allowing for a greater scope of expression.

2.1.1. Some Basic Operators

The simplest patterns are those containing only terminal symbols. Such patterns must be matched literally by the user's input. That is, a pattern like:

```
(the athlete)
```

will only match "the athlete," and not "the sportsman," "the ballplayer," or "the jock." If we wish to have the pattern matcher recognize all of these inputs interchangeably, we use the disjunction operator, a vertical bar '|'. Our new pattern would be:

```
(the (athlete | sportsman | ballplayer | jock))
```

and would match both the initial example and all of its variants listed above.

If we also wished to recognize inputs where the user neglected to enter the article ("the" in our example) we could use the optionality operator, a question mark '?', to write the pattern as follows:

```
(?the (athlete | sportsman | ballplayer | jock))
```

'?' has the effect of making the expression immediately following it (no space is left in between) into an optional constituent of the overall pattern. It may be used on any single terminal symbol or pattern. This means that successful matches of the pattern can occur in cases where the optional element(s) are in the user's input, and also when they are not present. Our new pattern matches everything the sample disjunction pattern matched, as well as "athlete," "sportsman," "ballplayer," and "jock."

Note that in the example above,

```
(athlete | sportsman | ballplayer | jock)
```

is an embedded pattern, and has its own set of parentheses. This is to ensure that the '|' operator is grouped unambiguously with its arguments. Most DYPAR operators require such grouping, and the grammar writer should assume it for any operator unless otherwise stated. However, the optionality operator '?' is an exception to this general rule. There is no question as to where its argument begins or ends because it works only on whatever expression follows it. The outer parentheses in our example are present to satisfy the rule that all patterns must be lists, not to set off '?the' as a separate operator-argument pair.

Sometimes all the variants that would satisfy some pattern cannot be specified in advance. For these cases, DYPAR provides a wildcard operator, represented by a dollar sign '$', which will match any word or numeral in the input. '$' is a special type of operator, one that, unlike '?' and '|', does not expect a subpattern as an argument[5]. This type of operator is referred to as a *niladic operator*. Any DYPAR operator which is prefixed with the '$' character is, by convention, a niladic operator. Here is how '$' would be used in a pattern by itself:

```
($)
```

This pattern will match any single word, but not groups of more than one word. This example is shown enclosed in parentheses because all patterns must be lists. However, when used inside another pattern, any niladic operator may stand alone without parentheses, since it has no arguments.

[4] When a pattern does not match the input, the matcher stops using that pattern and the template it was part of, and tries another template. If no template in the grammar matches the sentence, then the parse has failed and the parser will print out a message to that effect. If the entire sentence is successfully matched by a template, the parse succeeds and the parser will carry out the action side of the rule containing that template.

[5] A DYPAR subpattern can be another DYPAR pattern (in which case it is *embedded* because it is contained within the outer pattern), or one or more terminal symbols, or a wildcard operator, or some combination thereof. It is called a subpattern simply because it is inside another pattern. Most DYPAR operators take a subpattern as their argument.

Another operator, '*', is used to allow the same pattern to be matched in the user's input an arbitrary number of times (including 0). In other words, '*' tells the matcher to try the pattern over and over again until it is unable to consume any more of the user's input. Any operator, such as '*', that allows multiple matches of its pattern argument is called an *iterative operator.*[6] '*' is the same as the Kleene Star used in Regular Expressions. (If you have never heard of a "Kleene Star," don't worry. Its mathematical properties are of no relevance here.) '*' uses the form:

```
(* (war | famine | disaster))
```

This pattern would match "war war war war ...," "disaster war famine," "famine disaster," etc., and of course nothing at all. The '*' operator can be used in conjunction with the "$" operator to allow matches of arbitrary groups of words that would not otherwise be recognized by the matcher.

```
(* $)
```

Here our pattern matches anything that the user could type, and there is no restriction on the number of words matched.

2.1.2. Variable Assignment

Oftentimes it is the case that we wish to do more than recognize a user's input. We usually want to remember something of what was said for later processing by the system's backend (which, for instance, may formulate database query expressions). In DYPAR, a piece of input is saved for further use by assigning it to a variable name. This is known as *variable assignment*, because the variable name is assigned the relevant input as its *value*. Then when it is necessary to recall the data for some purpose (such as adding it to a database), the input thus saved can be called up by using the variable name, a procedure called *accessing* the value of the variable. DYPAR uses patterns like:

```
(!var := PAT)
```

to assign to the variable '!var' whatever part of the user's input was matched by the pattern 'PAT'. A more concrete example is:

```
(!jock := (athlete | sportsman | ballplayer | jock))
```

which assigns to the variable '!jock' as its value either "athlete," "sportsman," "ballplayer," or "jock," whichever of these words matched the user's input. If the user does not type any of those words, the variable is not assigned, as the match fails.

In the example above, the pattern side of the variable assignment is a subpattern using the disjunction operator '|'. It could also be a single terminal symbol; for example,

```
(!jock := sportsman)
```

in which case the variable assignment would only match the word "sportsman". In fact, the variable assignment operator's pattern argument can be any subpattern which consumes input[7]. All of the operators introduced thus far in the tutorial consume input.

Once a variable has been assigned, the value can be accessed in the LISP environment (and therefore in the context of a DYPAR rule) by using the variable name. This topic is covered, with examples, in the sections of this chapter dealing with DYPAR rules (Section 2.2.2 and Section 2.2.3). There is a special operator, '=', for accessing a variable assignment within a pattern. It is covered in Section 2.3.2. (Those who are just beginning to learn DYPAR should not feel obliged to refer to these sections until they encounter them in the normal course of the tutorial.)

[6] The pattern matcher moves through the user's input (English sentences) one word or number at a time, matching each piece of input to the patterns given it by the grammar writer. The matcher keeps track of what it has already matched by moving a marker (called a *pointer*) along the input. When the pointer is moved beyond a word or number in the user's sentence, the pattern matcher is said to have *consumed* that part of the input. Once a piece of input is consumed, it becomes invisible to the pattern matcher for the duration of the match. Normally the matcher will also forget a pattern as soon as it matches once, but the iterative operators allow a pattern to be matched to the input repeatedly.

[7] For an explanation of what we mean by "to consume input", see the footnote on page5.

You may have noticed that the variable names have been prefixed with the character '!'. This is a convention adopted in DYPAR to make variable names stand out from terminal symbols in a grammar file. (Often the variable name will be otherwise identical to one of the terminal symbols it can stand for, as in the "!jock" example above.) The '!' helps to avoid confusion between variable names and actual words encountered in the user's input . Chapter 4, beginning on page 27 of this document, delves more deeply into variable assignments and other options a grammar writer has regarding variables and their values.

An important note regarding patterns: All patterns must be written using lower-case letters only! DYPAR allows upper-case *user* input, which it handles by converting it to lower case before matching is attempted. (Further information on case sensitivity in DYPAR may be found in Section 2.2.2 on page 7.)

The set of operators we have just examined should suffice to allow us to move on and look at those rules which make use of DYPAR patterns. However, it should be noted that the operators introduced so far are *not* the complete set of operators which DYPAR recognizes. More operators will be introduced after the next section. Table 2-1 recaps the operators that have been discussed so far.

Desired Operation	Symbol(s)
Optional Element	?
Disjunct Set	| or !
Variable Assignment	:=
Wildcard	$
Repetition (including null)	*

Table 2-1: Basic DYPAR Grammar Operators

2.2. Rules in DYPAR

In this section we introduce DYPAR rule syntax. Each rule has a Left Hand Side (LHS) and a Right Hand Side (RHS) separated by a symbol which denotes rule type. Since the form of the RHS and LHS of any particular rule may differ with the rule type, we will examine each type of rule individually. Rule types are summarized in Table 2-2, on page 9.

2.1. Rewrite Rules

Rewrite rules define the building blocks of which the other types of rules are composed. A rewrite rule is used to associate some often used pattern, such as a class of words like articles ("a," "an," or "the"), with a symbol (or name for that pattern). We call these symbols *nonterminals.*

```
<article> -> (a | an | the)
```

In the above example, '<article>' is a nonterminal whose associated pattern is '(a | an | the)'. The symbol '->' serves to identify this rule to DYPAR's grammar interpreter as being a rewrite rule. Whenever the pattern matcher encounters one of these rules, the LHS (in this case '<article>') is rewritten (or expanded) to be the pattern associated with (the RHS of) that rule.[8] You may have noticed that the name of the rewrite rule ("article") has been surrounded with angle brackets ('<' and '>'). This is done to ensure that there is no confusion between nonterminals and literal or terminal symbols (much the same as the justification for the '!' used to prefix variable names). DYPAR will not recognize nonterminals unless they are enclosed in angle brackets! If the brackets are not included, the name will not be expanded to a pattern; instead, it will be treated as a terminal symbol.

[8] Internally the RHS (pattern) of the LHS (rule) is stored as the value of the "rewrite:" property of that rule. Thus, the rewrite: property of <article> would contain an internalized form of the pattern (a | an | the).

Rewrite rules can themselves contain nonterminal symbols in their RHSs. We can make up a set of rewrite rules to correspond to the examples we used while introducing the basic operators:

```
<athlete> -> (sportsman | athlete | jock | ballplayer)
<article> -> (a | an | the)
<jock> -> (?<article> (!jock := <athlete>))
```

This rule construction mechanism allows the grammar writer to build a hierarchical base for his top-level rules, and generally serves as a convenient means of avoiding very long and complex patterns. You may have surmised by now that, through the use of nonterminals, rewrite rules and DYPAR operators can be nested to arbitrary depths inside of each other. Nonterminals and operators can be used anywhere you wish inside a pattern.

Sometimes a rewrite rule is used within its own definition. Rewrite rules which reference themselves are commonly referred to as *recursive rules*. This is fine as long as the rewrite pattern does not begin with itself. For instance:

```
<repeat> -> (a ?<repeat>)
```

is an acceptable rule that matches one or more occurrences of "a" (although it can be written more efficiently as (+ a); see Section 2.3, on page 10.) However, the pattern:

```
<bad-repeat> -> (?<bad-repeat> a)
```

will not work. The reason it does not work is that the rewrite pattern for '<bad-repeat>' begins with itself, '?<bad-repeat>'. This pattern can never get around to matching "a", because it will first try to match '?<bad-repeat>', which tries to match '?<bad-repeat>', and so on ad infinitum. Patterns like '<bad-repeat>' are called *left-recursive* patterns. When DYPAR loads a grammar it generates warnings if it detects potentially left-recursive rewrite rules, and it will not try to match them. This means that if you should happen to write one by accident it won't cause the pattern matcher to enter an infinite loop (but it won't work, either.)

2.2.2. Top-level Rules

Top-level rules are central to a grammar, because they contain both the pattern sequences to match sentences and the instructions for whatever action is to be taken when a particular sentence is matched.[9] The syntax of a top-level rule is different from that of a rewrite rule. For top-level rules, the LHS is a pattern, and the RHS is an action to be taken whenever that pattern is successfully matched. The symbol which signifies that a rule is of type top-level is '=>'. Patterns used by top-level rules follow the same guidelines we have laid down for rewrite rules, except that in top-level rules the pattern occurs in the LHS instead of the RHS. The action (RHS) of a top-level rule can be any LISP code which is to be executed when that pattern is matched. (We will cover typical LISP commands in the next chapter.) We haven't defined enough nonterminals yet to let us showcase the typical top-level rule, but a simple example is:

```
(is he <jock>)
=>
(msg "You asked if he was a " !jock "?")[10].
```

Notice that the RHS of the rule can access variable assignments made in the LHS. One thing this means is that you may construct in the RHS a reply to the user which employs the same word he or she used to refer to a given thing. Also notice that in the RHS we used an upper-case alphabetic character, while we already warned you that *patterns* must be written in lower case alphabetic characters. However, messages to the user (which are not *evaluated* by FRANZ LISP) may contain capital letters, as above.This is because FRANZ

[9] When DYPAR-I is used as an integral part of a more powerful parsing system, such as DYPAR-IV, it is at this level (the *sentential level*) that its functions are replaced with those of the higher-level parser. Operators, rewrite rules and even the occasional transformation rule continue to be used by the more powerful system.

[10] 'msg' is a FRANZ LISP routine, a macro which prints verbatim that portion of its argument which is enclosed in double quotes, and prints the value of any other portion. So '!jock' evaluates to whatever we set it to in the LHS. In this case, that turns out to be whatever word the user typed in which matched the non-terminal called <athlete>. For example, if the user typed "Is he a ballplayer", the action taken by this rule would be to print out to the user "You asked if he was a ballplayer?"

LISP[11] is case sensitive, which means that it does not recognize the upper-case version of a given character as being identical to the lower-case version. In order to avoid problems with having to differentiate between capitalized and uncapitalized words typed by the user, all input is converted to lower case before pattern matching is attempted.

2.2.3. Transformation Rules

This type of rule is used to transform certain linguistic constructions into more canonical forms. The symbol '::>' tells DYPAR that it is processing a transformation rule. This proves to be a useful means of simplifying the pattern matcher and semantic grammar application process, especially when the transformed construction can occur in many different contexts. The example we give below is a rule that handles possessive constructions using "apostrophe s"; it simplifies them into "of" constructions (e.g., "John's car" becomes "car of John".) Although this construction is not generally used in English, it is semantically equivalent to the more common "apostrophe s". For purposes of DYPAR parsing, "of" is easier to handle than "apostrophe s", and in those instances where "of" *is* a normal usage (e.g., "Betty is the mother of John") the same top-level rule can match both that usage and the transformed "apostrophe s" usage.

For transformation rules the LHS is a pattern (as it is for top-level rules) and the RHS is a recipe for the reconstruction of the sentence. The RHS uses calls to LISP to generate the new sentence. (If you don't know SP, don't panic. The LISP function "append" is by far the most commonly used function for the RHS of transformation rules, and it is the subject of an extensive footnote below.) The following example serves to transform possessive constructions (e.g. "Sally's mother") into "of" constructions (e.g. "mother of Sally"):

```
((!beginning-of-sentence := (* $))
 (!possessor := $)
 %apost s
 (!possession := $)
 (!rest-of-sentence := (* $)))
::>
 (append !beginning-of-sentence !possession '(of) !possessor
  !rest-of-sentence)[12]
```

Here the LHS (pattern) consists primarily of a series of variable assignments, whose values are then accessed by the RHS to reconstruct the sentence in a different order. If you were to use this rule in a grammar, it would reconstruct all inputs containing "apostrophe s". If there is no "apostrophe s" in the input, then the transformation rule fails. For instance, given the sentence "Mary is Sally's mother, and that is why they look alike.", DYPAR would match the "apostrophe s" construction in the sentence to the "%apost s" portion of the pattern. The rest of the pattern would then be matched accordingly; that is, the word to which the "apostrophe s" was attached (*'Sally'*) would become the value of the variable !possessor, and all the words preceding it (*'Mary is'*) would become the value of the variable !beginning-of-sentence. Then the word immediately following the "apostrophe s" (*'mother'*) will be the value of !possession, and any words or punctuation left in the sentence (*', and that is why they look alike.'*) will be the value of !rest-of-sentence. When this sentence is reconstructed by the RHS of the rule, it becomes "Mary is mother of Sally, and that is why they look alike." Notice that the "apostrophe s" has disappeared entirely. We didn't need to keep it around, and since it was not assigned to a variable, it was simply matched and then thrown away by the pattern-matcher. The fact that the reconstructed sentence may ignore some rules of normal usage is irrelevant; it is for internal consumption only, and the meaning is intact.

Transformation rules, then, merely simplify sentences; they do not fill the same functions as top-level rules.

[11] FRANZ LISP is one of the LISP dialects in which DYPAR is implemented. The COMMON LISP version is not case sensitive.

[12] 'append' is a LISP function which strings together the element(s) of any number of given lists into one long list. Here the smaller lists are represented by variables (such as '!possession'), the elements are the words which the variables stand for and the resulting single list is the rearranged sentence. Note that '(of) is not a variable. LISP needs to recognize it as a list, however, so the word 'of' is enclosed in parentheses. The quote mark ' in front of the parentheses is necessary to prevent LISP from treating (of) as a function name. If you write similar transformation rules, in the RHS you will need to enclose sentence elements (words or punctuation) which are not variables in parentheses, prefaced by a single quote as in the above example. Since anything in the original user sentence that you wish to keep can be assigned a variable name, only new elements to be added in reconstructing the sentence need special treatment.

The action side of a transformation rule merely *rearranges* the words in the user's input,[13] whereas a top-level rule takes that input (sentence) and performs certain specific actions according to what the sentence meant as a whole. So any input that is matched and rearranged by a transformation rule must still be matched (in its transformed version) by a top-level rule.

This example makes use of one more feature of DYPAR:

- DYPAR reads punctuation characters in a special way. All punctuation characters are converted to another form at read time. The symbol '%apost' is used to represent the character "'" (referred to as apostrophe, acute accent or single quote.) It is important to remember to use the appropriate DYPAR punctuation symbol when you wish to include a punctuation character in a rule. A table listing all of the punctuation character to DYPAR symbol mappings is contained in Appendix A, on page 49.

Before you go off and write hundreds of transformation rules, a caveat is in order. A transformation rule should only be used when it replaces five or more top-level rules. DYPAR will only try to transform a sentence into another form after all of the top-level rules have been tried once, and have failed to match the input. This may sound quite time-consuming, but transformation rules (when properly used) can cut down considerably on the number of top-level rules you will need in a grammar, by making a given top-level rule work for more versions of a sentence. With fewer rules, the grammar will run faster overall.

Transformation rules are cyclic; i.e., the order in which they appear in the grammar files is the order in which the pattern matcher will try them. If one transformation rule (call it Rule A) is written such that it will use the result of a different transformation rule (we'll call this one Rule B), Rule A should appear *after* Rule B in the grammar file. Putting Rule A *before* Rule B would mean that a pass through the transformations will not make Rule A fire, even though Rule B fires.[14] Rewrite rules and top-level rules can be placed in any order the grammar writer desires; they are cross-referenced when the file is loaded and DYPAR will call them as needed.

Rule Type	Symbol	LHS	RHS
Rewrite	->	<name>	pattern
Top-level	=>	pattern	action
Transformation	::>	pattern	sentence transformation

Table 2-2: DYPAR Grammar Rule Types

[13] However, the grammar writer may add or take away sentence components. For instance, in our example we added the word "of", and took away the "%apost s".

[14] For example, our transformation rule to change possessive "apostrophe s" constructions will transform *any* "apostrophe s" it runs across, in its current form. This means a word such as "she's", actually a contraction for "she is", will become "of she", which is not a desirable result. We could write another transformation rule, which looked for these "is" contractions (instead of simply for "%apost s") and restructured sentences containing them in an appropriate way. Then we would place this rule *before* the "apostrophe s" rule in the grammar file, and it would replace all the "is" contractions with forms which would not trigger the "apostrophe s" rule. However, if we placed the new rule *after* the old one in the grammar file, it would never fire because all the "is" contractions would have already been transformed (incorrectly) by the original rule.

2.3. Additional Operators

In this section we introduce the rest of the DYPAR operators needed to write grammars. These operators. combined with those previously introduced (in Section 2.1.1) make up nearly the full set of DYPAR operators. A few operators are introduced later on, in the reference manual section of this document; most of them are primarily for internal use.[15]

2.3.1. Iterative Operators

The next two operators are closely related to the '*' operator introduced on page 5. '+' is exactly the same as '*' except the pattern must occur at least once in the user's input. (You may recall that '*' allows matches of its pattern any number of times, including 0. While '*' will allow the overall match to proceed whether or not its pattern is present, '+' will kill the match if the first piece of the input it looks at does not match its embedded pattern.) For instance,

```
(+ (war | famine | disaster))
```

would match "war war war war ...", "disaster famine", "disaster", etc., but it would *not* match nothing at all. '+' can be interpreted as the Kleene Plus used in Regular Expressions. So:

```
(?(+ (war | famine | disaster)))
```

is equivalent to:

```
(* (war | famine | disaster))
```

The other related operator, '↑', is used for exponentiation of a pattern. Here a pattern must occur a specified number of times in the user's input. '↑' expects two arguments, the first of which is the number of times the pattern must occur, and the second of which is the pattern. Thus:

```
(↑ 2 (war | famine | disaster))
```

would match "war war," "war famine," "war disaster," "famine war," "famine famine," "famine disaster," "disaster war," "disaster famine," and "disaster disaster," but not "war" or "disaster disaster disaster." '*', '+', and '↑' are the set of iterative DYPAR operators.

2.3.2. Variable Reference Operator

'=' is the operator used to reference the value of a variable assigned earlier in the pattern. If we wished (God knows why) to match inputs like: "sportsman sportsman" and "jock jock"; but not like: "sportsman jock" and "jock ballplayer", while still assigning a variable value, we could write a pattern:

```
((!jock := (athlete | sportsman | ballplayer | jock)) (= !jock))
```

hich would only match if the value assigned to '!jock' was repeated immediately in the user's input. Note that the following pattern will also parse "sportsman sportsman" or "athlete athlete", and will assign the same value to the variable as the pattern above does:

```
(↑ 2 (!jock := (athlete | sportsman | ballplayer | jock)))
```

However, the two patterns differ when two variants of the disjunct pattern are input. The former pattern does not parse such input; the latter pattern does, remembering only the first value of '!jock'. So if you gave the input "athlete ballplayer" to the first pattern, you would get back a message that the parse had failed; if you gave the same input to the second pattern, the match would succeed and '!jock' would have the value "athlete".

[15] These internal operators are generated by DYPAR during the cross-reference phase of grammar loading. DYPAR uses them to help speed up the matching process, and inserts them into the internal-format form of the grammar. More information about DYPAR's internal format may be found in Chapter 8.

2.3.3. 'Match-Until' Operators

The '&u' operator consumes input up to, but not including, the first successful match of its embedded pattern. Its pattern can be any legal combination of terminal symbols, nonterminals or operators. For instance,

```
<show> -> (show | list | print ?out | give)

(&u <show>)
```

would match the underlined part of "could you please show me...". As we will see later, this ability is very useful when we wish to skip over pieces of the user's input until we reach something we consider meaningful. '&u' will *ignore* any variable assignments made inside its pattern argument; since it does not consume the input which matches its pattern, it cannot assign a variable name to it.

The &ui operator is very similar to the '&u' operator, except that it consumes input up to and *including* its embedded pattern. For instance,

```
(&ui <show>)
```

would match the underlined part of "could you please show me ...". Like '&u', '&ui' is useful for skipping over pieces of user input. It accepts the same broad range of patterns as '&u'. '&ui' can be used when you want to find the meaningful part of the user's input and you don't know exactly what words that part will consist of. If you *do* know what words are likely to *precede* it, and you want to ignore them, you can just give '&ui' a pattern to match the words preceding the interesting part. It will find those words and consume them (as well as consuming any unmatched input before them.) You will then be at the part of the user's input you were interested in finding, and can do variable assignments or whatever you wish. (An example of this kind of use of '&ui' can be found in Section 5.2, under the description of '$lisp-function', beginning on page 32.)

2.3.4. Unordered Match Operator

It is often the case that the user has some discretion over the order in which he types certain word groups in his input. That is, there is often more than one way to structure what is semantically the same sentence. The '&c' operator gives the grammar writer some of this flexibility. '&c' will match each of its embedded patterns in any order they occur in the input. For our example we define two rewrite rules before actually using the '&c' operator:

```
<river> -> (over the river)
<woods> -> (through the woods)

((<river> &c and &c <woods>))[16]
```

This pattern would then match either "over the river and through the woods," or "through the woods and over the river." The difference between this and the '*' operator is that *all* of the elements of the pattern *must* occur once, and *only once* in the user's input. Think of &c as matching any permutation of its arguments. Those of you familiar with case grammars can see the utility of this operator as it provides a rudimentary case matching capability.

2.3.5. Negation and Scan Operators

If it is ever the case that we wish a match to succeed unless a certain word is present in the input, two closely related operators are provided. The first is '~'. Its usage is typically:

```
(~<article>)
```

This would mean that at this point in the match (assuming "~<article>" is a pattern element in a larger pattern) the match will fail if "a," "an," or "the" is present. The other operator '&n' works exactly the same way:

```
(&n <article>)
```

The difference between the two operators is in the effect they have on the matcher's position in the user's

[16] An alternative form for using '&c' is (&c (<river>) (and) (<woods>)).

input. '~' consumes the word it is looking at, provided that the word does not match its pattern argument (in which case the overall match will fail, as stated above.) Therefore it is most useful inside of a variable assignment statement, such as:

```
(!word := ~<article>)
```

This results in '!word' being assigned the next word in the input unless it matched '<article>'. '&n' on the other hand has no effect on the position of the matcher. It is mostly useful for filtering-out rules in conjunction with the '&u' operator.

```
((&n (&u <article>))  <sock> . . .)
```

would cause the matcher to fail if '<article>' was matched anywhere in the user's input. If '<article>' does not occur anywhere in the user's input, the matcher begins matching '<sock>' against the first word of the input.

While the negation operators covered above will kill the match if their argument pattern is *present*, the &s operator has the opposite effect. It scans the user input for its embedded pattern, and if that pattern is *absent*, the match will fail. If '&s' finds its pattern, matching will continue. For instance, the pattern

```
((&s time) . . .)
```

will scan the user input for the word "time"; if it finds the word, the match will continue, using the rest of the pattern (represented as ". . ."). If "time" is not found in the input, the match will fail immediately. Note that this operator does not consume any input, it just scans to see if its pattern is in the as-yet-unmatched part of the input. It will not move you up to the part of the input where it found the match to its pattern; all it does is allow matching to continue. This means you must match the pattern in addition to having '&s' scan for it. Because it does not consume input, '&s' is the complement of the '&n' operator (the '~' operator consumes that portion of the input which matches its embedded pattern.)

The argument pattern for '&s' may be made up of any legitimate terminals or non-terminals in the grammar, and may include any of the other DYPAR operators. However, using one '&s' inside the argument of another '&s' is inefficient.

2.3.6. Wildcard (Niladic) Operators

The '$r' operator (or remainder wildcard) will match the *rest of the input.* It is used primarily in transformation rules to capture what remains of the sentence after the rest of the pattern has matched.

```
(stop $r)
```

would match any sentence that began with the word "stop," no matter which words (including none) compose the remainder of the sentence. The astute reader will have noticed that both '&u' and '$r' can be defined by either the '*' operator or recursive '->' rules. But using '&u' and '$r' directly is clearer and much more efficient.

There is another basic wildcard operator: numeric wildcard, '$n'. '$n' is analagous to '$' except it matches only numbers.

```
($n)
```

would be used to match numbers like: "1," "3.14," or "-7."

There are three additional wildcard operators described in Section 5.2. However, beginning-level grammars are not likely to require their use.

2.4. Morphology

The morphology operator, &morph, is used to match a single root word with some set of suffixes[17]. Suppose you were expecting the user's input to contain information about the relative size of two objects--for instance, "The chair on the right is smaller" or "The chair on the right is the smallest." You could write a pattern

[17] Prefixes will be implemented in the future.

```
(smaller | smallest)
```

to handle the adjective, but if you wanted to break down the information contained in either word into the concept expressed by the root ("small") and the concept expressed by the ending (comparative or superlative), you would need to use an operator that recognized these components separately. &morph was created to fill this need.

The grammar writer tells &morph what root word to look for and what suffixes are allowed with the root. The form of a morphology pattern is a list whose first element is &morph. The rest of the pattern consists of either one or two :keyword pattern pairs. For example, a pattern to match either the word "smaller" or the word "smallest" would look like this:

```
(&morph :root small :suffix (er | est))
```

There are two :keyword pattern pairs in the example above. The first one is :root small, where :root is the keyword and small is its pattern. The second pair is :suffix (er | est), where :suffix is the keyword, and (er | est) is its pattern. Another keyword, :endings, is a synonym for :suffix.[18] These three (:root, :suffix and :endings) make up the set of allowable keywords for the '&morph' operator. You can use any DYPAR-I operator in the pattern argument to a keyword, but the most useful ones for this purpose are the disjunction operator and the variable manipulation operators. With variable assignments, the root of a word and its ending can be stored separately for later use. We will show an example of this usage later in this section.

Now that we have explained what '&morph' is for, we will examine how a morphology pattern is matched to a word in the user's input. The pattern matcher attempts to match the pattern immediately following the :root keyword to the possible root words of the current word in the input. In order to do this matching, the matcher must find those root words. The matcher's first step in determining possible root words is to take the current word and check for the following conditions:

- no endings can be taken away
- the word is less than three characters in length
- the word is in the pattern matcher's dictionary[19].

If any one of these conditions is met, the current word is considered to be the root word. Otherwise, the pattern matcher will strip an ending off the word and apply the conditions again, until it gets a word that will meet one of the requirements. '&morph' has a list of endings for the various English verb and adjective forms, and for plurals, and will look for these endings on the current word. Any endings stripped from the input word will be saved for matching to the pattern(s) following the :endings or :suffix keyword. Should it be the case that the *unaltered* form of the current input word meets one of the above conditions, it will be matched immediately.[20]

If the root word thus obtained does not match the pattern following the :root keyword, the overall match will fail here. '&morph' knows about spelling rules in English and will automatically take into account such differences as 'y' being changed to 'i', or doubled consonants before an ending. No match should fail due to such spelling changes. Assuming the root does match, the pattern matcher then tries to match the saved list of endings to the pattern immediately following either the keyword :endings or the keyword :suffix. If the match of ending(s) fails, then the overall match fails.

[18] Only one of these synonymous keywords should be used in any given morphology pattern.

[19] The matcher's dictionary is explained with the $d operator, on page32.

[20] If the word exists in the dictionary in an inflected form (i.e., with one or more endings) that matches the current input, this will satisfy the dictionary condition and the current input will be defined as the root--whether or not it actually *is* the root form. If this situation occurs, there will be no ending to match to the pattern argument of the :endings keyword. One of two things will then happen: 1: If the pattern argument of the :endings keyword is an optional one (e.g., ?(ed | ing)) the match will still succeed, since it was not required that the argument to :endings be matched. (Note, however, that the input word matched by the pattern argument to :root may not actually *be* the root.) 2: If the pattern argument of the :endingsb keyword is *not* optional (e.g., (ful | less)), the match will fail, as there is nothing to match to this pattern.

In the case that the grammar writer wishes only to check to see if a root word matches, and doesn't care about whatever endings it may have, the form to use is

```
(&morph :root small)
```

When neither :endings nor :suffix is used, then the possible lists of endings generated in the stripping-off process are not matched against a pattern. If the root word matches, the match will succeed. Conversely, if for some reason the grammar writer is only concerned with the ending(s) of a particular word in the input, the form is

```
(&morph :endings (er | est))
```

or

```
(&morph :suffix (er | est))
```

When the :root keyword is not used, then the list of possible root words is generated as above, but is not matched against a pattern. If '&morph' is used without any keywords it will fail.

Variable assignments are allowed within each keyword pattern argument (subpattern), but the variable reference operator '=' may not be used within one subpattern to access a value assigned within the other subpattern. (There should, in any case, not be any reason to attempt this feat.) Here is an example of the use ''&morph' with a variable assignment to save the root word:

```
(&morph :root (!wrd := (candy | cook)) :endings (ed | ing))
```

This pattern will match any one of the following user inputs: candied, cooked, candying, and cooking. Note that it will *not* match simply "candy" or "cook", as the :endings keyword has a pattern which must be matched with either "ed" or "ing".

2.5. Operator Summary

All of the operators introduced thus far are summarized in Table 2-3:

Desired Operation	Symbol
Optional Element	?
Disjunct Set	| or ![1]
Variable Assignment	:=
Variable Reference	=
Wildcard	$
Numeric Wildcard	$n
Repetition (including null)	*
Repetition (at least once)	+
Repetition (specified number)	↑
Up-to	&u
Up-to-and-including	&ui
Remainder (of input)	$r
Scan (rest of input)	&s
Unordered	&c
Negation(for variable assignment)	~[2]
Negation(as a filter)	&n
Morphology	&morph

[1] Either of these symbols is sufficient.

[2] Ascii value is 126; use symbol which has that value.

Table 2-3: DYPAR Operator Summary

EXERCISES

For information on how to run DYPAR-I look at sections 6.4-8.7.1, starting on page 36 of the manual. Answers to exercises can be found in Appendix E, starting on page 61.

The grammar files you write in doing the exercises must contain a top-level exit rule of the form:

```
(<exit> (* $)) => 'exit
```

Exercise 2-1: Suppose you are beginning to write a natural language interface to a database or information-retrieval system which can, for example, represent questions concerning when classes in some school meet. It turns out that at this school classes can meet from one to five days per week on any day except Saturday and Sunday.

a. Construct a rewrite rule which represents or covers all the possible days a given class meets and saves this information for later processing.

b. Using this rule, write a top-level rule which parses questions asking whether classes in the philosphy and computer science departments meet on given days at given times. Suppose that each class can be designated by a single descriptive word together with a number. For example you might write a top-level rule which recognizes questions like:

```
does phil 1 meet m w f?
does compsci 201 meet on tu th?
```

where "m w f" can be replaced by any sequence of week days completely written out or abbreviated in a variety of ways.

The action to be taken on a successful match of the pattern on the left hand side of the rule should be a message that shows which days the question specified. The message should be like the one that forms the action or right hand side of the example rule in section 2.2.2 page 7.

You may use supplementary rewrite rules to define additional nonterminals as needed to simplify your top-level rule.

Exercise 2-2: Write a grammar containing two top-level rules to recognize the different ways of asking *when* and *where* a class meets. These rules should cover inputs using the words "when" and "where" but also inputs using phrases like "at what time". You may name classes as in Exercise 2-1. The action to be taken should be a message indicating which type of question was asked.

Exercise 2-3: Write a grammar containing two top level rules: one which parses input asking whether a particular person is in a given class (name classes as above), the other asking whether anyone is in that class. For example, be sure your grammar can parse and distinguish between the following:

```
is smith taking phil 1?
is anyone taking phil 1?
```

Assume that your grammar does not contain a list of all student names. Also assume that a student's name can be made up of a first and last name and a middle name or middle initial.

Exercise 2-4: a. Write a grammar that will parse the following input.

```
does compsci 10 meet at 1pm on m w f?
is phil 101 being given tu th at 11:30?
```

b. After you get your grammar to work on the above input, supplement the grammar to cover input which begins with words and phrases that are not essential to specifying the query that needs to be performed. Some examples are:

```
Please tell me does....
Will you please say whether...
Do you know if...
Can you tell me is...
```

You could use the operator '&u' and '&ui' at the beginning of top-level rule patterns to skip over such phrases. However this way of dealing with inessential input does not make use of the cross referencer and can significantly slow down parsing speed. Find a way around this problem and implement it. Do not worry about the variety of inputs your grammar covers as long as it parses the above examples along with whatever inessential words and phrases that might precede them.

Three BUILDING A "REAL" GRAMMAR

In this section we will examine a working DYPAR grammar, and discuss some of the considerations that entered into its construction. This grammar[21] can be used to experiment with a primitive semantic network[22] and thus get some hands on experience using DYPAR. Our example grammar is by no means complete, in terms of coverage of its domain. It is presented more as a model of how to write a grammar, than as an example of the range of sentences that can be parsed using DYPAR.

3.1. The Semantic Network

The program that interfaces with our grammar is a very simple semantic network. One thing that might help us to specify a grammar to talk to the semantic network is an understanding of what that network can and cannot do. This might seem like a rather obvious point, but it does make a difference. If you don't know what you want DYPAR to understand, your grammar will not be very useful. Or, more simply, time spent learning the scope of the abilities of the back end is not wasted!

Our network is capable of storing and retrieving simple "is-a" relationships (e.g. "Mary is a painter" and answering "What is Mary?"), traversing an "is-a" hierarchy (e.g. "Fido is a dog" and "Dogs are canines," implies to the network that "Fido is a canine"), storing and retrieving named property information relating to some node in the network (e.g. "Bob's pencil is short" and "What do you know about Bob's pencil?"), deleting information (e.g. "Forget about Bob's pencil."), altering information (e.g. "Bob's pencil is long"), and making some types of inferences (e.g. given "Mary's mother is Sally" and "The opposite of mother is daughter," the system will infer that "The daughter of Sally is Mary.").

The network also has some important limitations we might want to keep in mind. It cannot handle multiple "is-a" relationships (e.g. "John is an Armenian." and "John is a dentist" are considered contradictory), or complex adjectival information (e.g. "Large furry happy brown dog" makes it gag). In fact, we didn't spend a lot of time working on the network, so it is pretty simple minded. It is, however, good enough to demonstrate the utility and function of DYPAR in action.

3.1.1. Network Functions

This section briefly examines those LISP functions which are of importance to the person writing a grammar making use of the semantic network. These are the functions that are used in the RHS or action side of the top-level rules:

ltm-ret (Long Term Memory RETrieval) is used to extract information from the semantic network.

ltm-ret-all (Long Term Memory RETrieve ALL) extracts all information about a given node from the semantic network.

ltm-store (Long Term Memory STOREage) is used to place information into the network.

ltm-forget (Long Term Memory FORGET) is the function which deletes nodes from our network.

ltm-spec (Long Term Memory SPECify) decides which interpretation of a sentence should be stored in the network.

storefile saves the current state of the network to a disk file for later restoration.

loadfile restores the state the system was in from a previously saved session with the semantic network.

[21] The grammar is reproduced, without any accompanying text, in Appendix C page 55. At CMU it is contained in [CAD]/usr/xcalibur/grammars/sem.gra.

[22] The source for the semantic network lives in the LISP file [CAD]/usr/xcalibur/dyparl/src1.11/semfns.l.

3.2. Groundwork

Now that we have a idea of what the network can do, the next step is to characterize the way those capabilities can be expressed in English sentences. A corpus of example inputs is invaluable for this task. As was previously stated, we wish to be able to recognize requests for information, assertion of new information to be stored, and a few simple commands. So our initial input corpus reflects these desires:

Assertions

Mary is a woman.
John's mother is Mary.
The inverse of mother is son.
Fido is a lazy dog.
Sugar is an ingredient of cookies.

Information Requests

What is Mary?
Is Mary a woman?
Who is John's mother?
Is Mary John's mother?
Tell me all that you know about Mary.
What is the inverse of son?

Commands

Save this session.
Load the number.gra file.
Exit the parser.
Forget about Mary.

We have informally stated our goals for the grammar in terms of the type of inputs it must recognize. Now our task is to prepare a set of sentence templates that implement these goals. However, before we can do that we must come up with a more rigorous statement of the types of inputs we need to recognize. This is not as difficult as it might appear. Certain regularities make themselves apparent from the corpus of examples.

- One class of sentences that we want to parse concerns "is-a" relationships between two objects[23].
- Another class specifies a particular property of an object.
- Questions derived from either of the two preceding forms can be asked, as well as requests for all that is known about some object.
- We wish to allow a command for saving the current "state of the world" in a file for later usage, and a command to bring one of those stored files into the environment.
- Explicitly cancelling information is also on our list.
- Most importantly, we need to be able to quit the system.

[23] We are using the word "object" to mean the focus of the input

3.3. Writing the Grammar

First, we pick one of the sentence classes and begin to write our grammar.

3.3.1. Queries

We will start with one of less complex sentence types: "What is ____" (e.g. "What is Mary"). The first try would probably look like:

```
(what is $)
```

That would be fine, except that we forgot to record the value of $ for use later on.

```
(what is (lnam :* $))
```

That looks Ok for the LHS of the rule. We also need an action half (RHS) for our rule:

```
(ltm-ret lnam 'isa: nil nil)
```

Still this rule doesn't cover many of the different ways of asking a question. Perhaps we should define nonterminals to take the place of "what" and "is" in our rule.

Taking the easier problem first, we define the rewrite rules used to conjugate the verb "be." For the present tense our rule is:

```
<be-present> -> (is | are | be | am)
```

For both past and future tenses, the forms become more complex. For example, "have been," and "has been" are both valid instances of "be." Perhaps we should digress and write the rules for recognition of forms of "have":

```
<have-present> -> (have | has)
<have-past> -> (?<have-present> had)
<have-future> -> (will have ?had)
<have> -> (<have-present> | <have-past> | <have-future>)
```

Now back to "be":

```
<be-past> -> (was | were | <have-present> been | had been)
<be-future> -> (will be | will have been)
<be> -> (<be-present> | <be-past> | <be-future>)
```

The rewrite rules we have written so far serve to point out the naturally hierarchical manner in which a grammar should be developed. The rules meant to recognize forms of "be" and "have" each have subpatterns to recognize the different tenses of those verbs that we might encounter. If later on, while defining top-level rules, it becomes important to recognize only one tense of the verb, we need only use the particular subpattern of the "<be>" rule we require. Whereas, in cases where tense is not important we can use the "<be>" nonterminal itself. Our "<be>" rule will match 11 different forms for expressing the existence of an "is-a" relationship.

The other area where some grammar primitives are needed is for the recognition of "what" forms. We will define nonterminals for words or phrases used to begin questions.

```
<q-word> -> (what | who | where | when | how | why | how much |
             how many | how come)
<www> -> (what | who | which)
```

Questions usually occur in more exotic forms, e.g. "what's," or "who is." We need to define a nonterminal to recognize the " apostrophe s" attached as a suffix to indicate contractions as well as posessives before we can handle such instances. Any time we wish to include a punctuation character as part of an expected input, we must remember that punctuation characters have specific internal names. A punctuation character table can be found in Appendix A on page 49.

```
<poss> -> (%apost s)
```

Now back to "what's" and "who is"

```
<what-q> -> (<www> <be-pres>[24] | <www> <poss>)
```

By the way--"<what-q>" can be matched 36 different ways.

There are still a few ways of asking for information that we haven't considered yet: "could you tell me what ...," "can you give me ...," etc. Words like "could" are refered to by linguists as *positive modal auxiliaries*. We will write a rule to recognize those words, and then another to attach them to the pronoun "you."

```
<pos-modal> -> (could | would | can )
<polite> -> (<pos-modal> you)
```

Since "me" and "us" can be used interchangeably (as far as we're concerned), we should group them together.

```
<me-us> -> (me | us)
```

Next we make us of the rewrite rules we have already defined to build one rewrite rule to recognize all of the ways we have discussed of formatting the initial part of the question. Again, if later on it became important to handle the different query forms in different ways. we need only use the particular nonterminal that covers that type of question. Take a few minutes to work through what each of the following rules can match.

```
<info-req1> -> (?<polite> <info-req2> ?<what-q>)
<info-req2> -> (tell <me-us> ?about | give <me-us> | print | type )
<info-req3> -> (<www> | ?<polite> <info-req2> ?<www>)
<info-req> -> (<what-q> | <info-req1>)
```

The last four rewrite rules start to add up some pretty impressive numbers for possible ways to match them.

Nonterminal	Possible Matches
<Info-req>	1122
<Info-req1>	1084
<Info-req2>	8
<Info-req3>	152

Table 3-1: Number of Possible Ways to Match Rules

The top-level rule we were working on earlier would now look like:

```
((<info-req> (!nam := $))
=>
(ltm-ret !nam 'isa: nil nil)
```

However, we still haven't covered: "What is the _____?" Both the punctuation ("?") and the determiner ("the") would be missed.

"The" is part of the group of words which, when they precede a noun, associate a quantity with that noun. These words are called *quantifiers*. Let us define a set of rewrite rules that recognize quantifiers. First, the indefinite determiners:

```
<a-an> -> (a | an)
```

Now, some more exotic forms:

```
<bulk> -> (bulk | majority | greater part)
<universal-quant> -> (?almost all | ?almost every ?one | each |
                      most | many | the <bulk> of)
```

When we put it all together:

[24] You may notice that a potential error has crept onto the scene at this point. In fact it is a real one. To identify and eliminate this and other errors when writing grammars see Section 8.7.1, page 46.

```
.<det> -> (the | <a-an> | <universal-quant>)
```

We solve the punctuation problem with a couple more rewrite rules:

```
<punct> -> (%qmark | <dpunct>)
<dpunct> -> (%period | %emark)
```

Going back to our top-level rule we plug in the optional determiner, and allow for optional punctuation at the end of the sentence.

```
(<info-req> ?<det> (!nam := $) ?<punct>)
=>
(ltm-ret !nam 'isa: nil nil)
```

This rule now can match such varied sentences as: "What is Mary?," "Could you tell me about the horse.," and "Print what's Mary."

We now move on to the more complex question forms, such as: "What is the _____ of _____," or "Could you tell me the _____ of _____." (e.g. "Could you give me the color of the horse.")

```
(<info-req> ?<det> (!prop := $) of ?<det> (!nam := $) ?<punct>)
=>
(ltm-ret !nam !prop nil nil)
```

And then there's: "What _____ is _____" (e.g. "What color is the horse?")

```
(<info-req3> (!prop := $) <be-pres> ?<det> (!nam := $) ?<punct>)
=>
(ltm-ret !nam !prop nil nil)
```

Another form is: "Is _____ a _____." (e.g. "Is blue a color?")

```
(<be-pres> ?<det> (!nam := $) ?<a-an> (!val := $) ?<punct>)
=>
(ltm-ret !nam 'isa: !val nil)
```

Also: "Is the _____ of _____ _____." (e.g. "Is the color of the horse blue?")

```
(<be-pres> ?<det> (!prop := $) of ?<det> (!nam := $) ?<det> (!val := $)
          ?<punct>)
=>
(ltm-ret !nam !prop !val nil)
```

Yet another form is: "Is _____ the _____ of _____" (e.g. "Is blue the color of the horse?"). For this example the prepositions become more complex. We might as well define rewrite rules for prepositions in general at this point. All of this group of nonterminals can be used to locate the beginning of a prepositional phrase in the input.

```
<prp> -> (of | to | for | with)
<prp-about> -> (about | on)
<prp-in> -> (on | in | into | onto | inside | within)
<tof> -> (to | of)
<ofor> -> (of | for)
```

Now our definition for the top-level form:

```
(<be-pres> ?<det> (!val := $) ?<det> (!prop := $) <tof> ?<det>
          (!nam := $) ?<punct>)
=>
(ltm-ret !nam !prop !val nil)
```

The last query form we will define is that of requests for all information about something (i.e. "Tell me all that you know about _____," or "What is everything known about _____") (e.g. "Tell me all you know about Fido.") We need rewrite rules for "know," and also for a subset of the quantifiers rules we've already defined.

```
<all> -> (all | everything | what)
<that-do> -> (that | do)
<known> -> (you <know-have> | ?is known | there is | stored
                            | in memory)
<know-have> -> (know | have)
```

The top-level rule:

```
((<info-req> <all> ?<that-do> ?<known> <prp-about>
       ?<def> (nam := $) ?<dpunct>)
=>
(ltm-ret-all !nam)
```

3.3.2. Assertions

After finishing one class of sentences we move on to the next. There are some subclasses of assertions that we want to recognize separately, i.e., there are certain relationships expressed in assertions that the semantic network makes special allowances for.

For sentences like: "____ is a name." (e.g."Minneapolis is a proper noun.") we will need rewrite rules to recognize "naming words."

```
<label> -> (word | term | name | label)
<dlabel> -> (?the <label>)
<name> -> (?proper name | ?proper noun | token ?mode)
```

Then we can write our rule. In the action side of the rule both 'token' and 'node-type:' are of special significance to the semantic network.

```
(?<dlabel> (!nam := $) <be-pres> <a-an> <name> ?<dpunct>)
=>
(ltm-store !nam 'token 'node-type: nil nil)
```

Synonomy is likewise a predefined property in the semantic network. First, the rewrite rules:

```
<same> -> (what | <same1>)
<same1> -> (?the same ?thing <as-that>)
<as-that> -> (as | that)
<means-does> -> (means | does)
```

Now we can write the top level rules to handle sentences like: "____ is a synonym for ____." (e.g. "The word pun is a synonym for the word joke.")

```
(?<dlabel> (!nam := $) <be-pres> <a-an> synonym <ofor> ?<dlabel>
              (!val := $) ?<dpunct>)
=>
(progn (ltm-store !nam !val 'synonym nil nil)
       (msg "Henceforth when you type "
            !nam " I'll interpret it "
            "as " !val t))
```

Although the previous rule illustrates that arbitrary LISP code can appear in the RHS of any top-level rule, it is preferable (for clarity's sake) to assert a single LISP function call as the action to be associated with the pattern. Another form for this type of sentence is: "____ means the same as ____ does." (e.g. "Dog means the same thing that canine does.")

```
(?<det> (!nam := $) means ?<same> (!val := $) ?<means-does>
       ?<dpunct>)
=>
(ltm-store !nam !val 'synonym nil nil)
```

Now we move back to more generic types of assertions. These are not as simple as they might at first appear. Oftentimes there are different ways to interpret an assertion. The semantic network function ltm-spec is used for such disambiguations. First, though, we will examine the unambiguous case. These are sentences like: "A ____ is a type of ____." (e.g. "A pig is a kind of animal.") Once again, we need to define rewrite rules for "type" words.

```
<typeof> -> (<type> ?of)
<type> -> (type | kind | form | instance | example)
```

And then define the top-level rule.

```
(?<a-an> (lnam := $) <be-pres> <a-an> ?<typeof>
 ?<a-an> (lval := $) ?<dpunct>)
=>
(ltm-store lnam lval 'isa: nil nil)
```

Sentences like: "____ is the ____ of the ____." (e.g. "Blue is the color of the horse.") are also unambiguous. Here's the rule for them.

```
(?<det> (lval := $) <be-pres> ?<det> (lprop := ~<type>) of
         ?<det> (lnam := $) ?<dpunct>)
=>
(ltm-store lnam lval lprop nil nil)
```

Note the use of the '~' operator to avoid a conflict with the last rule.

Our next rule must ask the semantic network to decide what is the proper interpretation of the form: "____ are ____." We could say either "Dogs are animals," or "Dogs are furry." In the first case we are making an is-a link between "dogs" and "animals." In the other, the relationship is one of "amount of hair" associated with "dogs." ltm-spec asks the user to decide on the proper interpretation.

```
(?<det> (lnam := $) <be-pres> (lvorp := $) ?<dpunct>)
=>
(ltm-spec lnam lvorp nil nil nil)
```

Assertions like: "____ is a ____ ____" (e.g. "Fido is a fat dog.") are handled by the next rule, which asks the user for the relationship between "fat" and "fido."

```
(?<det> (lnam := $) <be-pres> <a-an> (lvorp := $) (lval := $)
        ?<dpunct>)
=>
(progn (ltm-store lnam lval 'isa: nil nil)
       (ltm-spec lnam lvorp nil nil t))
```

3.3.3. Commands

When we developed our specification, we included a number of commands the system must be able to understand. Now we will define some rewrite rules that recognize the different synonyms for the various commands. This saves us the trouble of defining a different top-level rule for each synonym. Note the use of an optional wildcard in the '<exit>' rule.

```
<forget> -> (remove | delete | erase | forget ?about | wipe out)
<load> -> (load | input | read ?in | dskin)
<store> -> (save | store | output | write ?out | dskout | print ?out)
<exit> -> (quit | exit | end ?$ session | ?good bye)
<command> -> (<forget> | <load> | <store> | <exit>)
```

We can also make a rewrite rule for the any of the verb forms we have defined so far. This might be used in a transformation rule to pick up the part of the input that is a verb separately from the rest of the input.

```
<verb> -> (<command> | <be> | <info-req> | <have>)
```

One of the commands we wished to handle was of the form: "Forget the ____ of ____." (e.g. "Forget the color of the horse.")

```
(?<pos-modal> <forget> ?<det> (lprop := $) of ?<det> (lnam := $)
              ?<punct>)
=>
(ltm-forget lnam lprop)
```

This rule is for the load command (e.g."Load the file session."):

```
(?<polite> <load> ?the ?file (!fil := $) ?<punct>)
=>
(loadfile !fil)
```

Now a rule for the store command (e.g."Store the session in the file session."):

```
(?<polite> <store> ?the ?session ?<prp-in> ?the ?file (!fil := $)
 ?<punct>)
=>
(storefile !fil)
```

And as we intimated earlier, the most important rule of all, the exit rule (e.g "Exit the parser."):

```
((<exit> $r) => 'exit
```

3.3.4. Negation

While we didn't cover this earlier, it is helpful to capture some forms that the back end can't handle. This makes the person using the system less frustrated when something typed in is not understood. The semantic network doesn't understand negations, so we write a rule to warn the user of this fact. First, a rewrite rule, and we might as well write the rewrite rule for affirmation at the same time. It might become useful in the future.

```
<neg> -> (no | not | never | none | nothing | %apost t)
<pos> -> (yes | sure | indeed | certainly | certain | surely)

((&u <neg>) <neg> $r)
=>
(msg "I do not understand negations yet." (N 1))
```

3.3.5. Transformations

"Please" is a meaningless word, for purposes of understanding what the sentence means. We can write a transformation rule to flush "please" from the input. We could define a class of meaningless noise words and filter them out as well.

```
((!s1 := (&u please)) please (!s2 := $r))
::>
(append !s1 !s2)
```

Other uses for transformation rules are to change from one form of a sentence to another which has the same semantic content, but is a different syntactic construction. The next rule converts sentences of the form: "Could you tell me what the color of the horse is." to "Could you tell me what is the color of the horse." The "is" is moved so that the top-level rules we defined earlier can recognize the sentence.

```
((!s1 := (&u <q-word>)) (!q := <q-word>) (!s2 := (&u <be>))
        (!v := <be>) ?(!p := <punct>))
::>
(nconc !s1 !q !v !s2 !p)
```

This rule expands "what's" to "what is," "who's" to " who is," etc.

```
((!s1 := (&u <q-word>)) (!w1 := <q-word>) <poss> (!s2 := $r))
::>
(nconc !s1 !w1 (list 'is) !s2)
```

Sentences like "Mary's mother is Sally." are converted to "Mother of Mary is Sally."

```
((!s1 := (&u $ <poss>)) (!w1 := $) <poss> (!w2 := $) (!s2 := $r))
::>
(nconc !s1 !w2 (list 'of) !w1 !s2)
```

This one converts from "Mother of Mary is Sally" to "Sally is mother of Mary."

```
(?<det> (!w1 := $) (!prp := <prp>) ?<det> (!w2 := $) (!v := <verb>)
        (!s2 := $r))
::>
(nconc !s2 !v !w1 !prp !w2)
```

Four VARIABLE ASSIGNMENT

The method used to associate a value with a variable that was introduced in Chapter 2 is not always flexible enough to capture all of the currently pertinent information. This chapter examines the more powerful variable assignment tools provided with DYPAR.

4.1. The Repetitive Variable

If you were to write a pattern that used the same variable twice:

```
((!a := forgotten) (!a := value))
```

you would find that only the second value of the variable was available to your backend LISP functions. Looking at the example, the value of '!a' would be "value," not "forgotten" (provided, of course, that the pattern was successfully matched). This type of behavior makes it impossible to use a simple variable assignment inside of an iterative pattern (i.e. '*', '+', '↑', will not work with named variables. The same is true for recursively defined nonterminals.) DYPAR, does however, have the capability to generate new variable names for you. To do this you must name your variable '*var*'. The generic form for this is:

```
(* (*var* := <variable-value-list>) ?<other-pattern-elements>)
```

In this example the '*' operator says that the following pattern may occur any number of times. The pattern we're talking about consists of a variable assignment statement (using '*var*') and another nonterminal which we expect to occur in the input, but don't want to remember. The atom 'var1' will be assigned the first match of <variable-value-list>, 'var2' the next match, etc... A similar way of utilizing '*var*' would be in the recursive pattern:

```
<nt> -> (a b (*var* := (c | d)) ?<nt>)
```

which would match "a" followed by "b" followed by "c" or "d" an arbitrary number of times (e.g. "a b c a b d a b c" would return var1 = "c," var2 = "d," and var3 = "c")

The names of the newly generated variables are stored in a variable named '!newvars'. The value of the '!newvars' variable will be the list *(var1 var2 ...)*, where the individual *var$_i$* are the variable names that were generated during the current parse.

4.1.1. Accessing !newvars

access is a LISP function you can use to access the values associated with the DYPAR generated variable names stored in !newvars. This avoids your having to reference '!newvars' directly. It will return a list of the variable values in the order they occurred in the user's input. To use access to get at '!newvars' you must do:

```
(access !newvars)
```

access is meant for use in the RHS of Top-level rules:

```
((* (*var* := (red | blue | white | black)) (%comma ?and | and)))
=>
(msg "The colors are:  " (access !newvars))
```

Other methods for accessing '!newvars' and making use of repetitive variable assignments are interspersed throughout the rest of this chapter.

4.2. Variable Coercion

On occasion, you want to assign a variable some value other than the input segment matched by the current pattern. This is called *variable coercion* and is accomplished via the '&i' operator.

4.2.1. Map into a Value

The '&i' operator is used to allow different kinds of information to be transmitted as the value of the variable named in the nearest enclosing variable assignment pattern. This is useful for limiting the number of different values that the backend functions need to be able to process. '&i' takes two arguments, the first being the value you wish to be assigned to the variable when the second argument (any valid pattern, usually a nonterminal) successfully matches the input. For instance:

```
((!varname := (&i true <affirmative-response>)))
```

means that '!varname' will be assigned the value 'true' if the input matches '<affirmative-response>', which could consist of a large number of possible words and phrases.

```
<affirmative-response> -> (yes | true | yup | ?absolutely correct)
```

In our example the '&i' pattern was a direct sub-pattern of the variable assignment pattern. The same effect could be achieved in the following way:

```
<true> -> ((&i true <affirmative-response>))
((!variable := <true>))
```

'&i' has meaning only when it is used within the context of variable assignment. An '&i' pattern that is not caught by a := will probably cause a LISP error.[25] (This may be fixed in a future release.)

4.2.2. Function Calls

Let's take a somewhat closer look at the syntax of patterns containing '&i':

```
((&i VALUE PATTERN))
```

'VALUE' can be either a list or an atom. When 'VALUE' is an atom the behavior of '&i' is exactly as described in the last section. However, when 'VALUE' is a list DYPAR looks at the first word in that list to see if it is either the symbol '&apply', or the symbol '&funcall'. The occurrence of one of these symbols as part of 'VALUE' signifies that you wish to apply an arbitrary LISP function to some variables that you assigned within 'PATTERN'. The generic form for this is:

```
((&i (&apply fun-name (v-name1 v-name2 ...)) <var-assigns>))
```

Here 'VALUE' corresponds to:

```
(&apply fun-name (v-name1 v-name2 ...))
```

and 'PATTERN' corresponds to '<var-assigns>'. The result of this is that the value returned from the '&i' is the value returned from the call to 'fun-name' with 'v-name1', 'v-name2', ... as its arguments. This is exactly what would happen if you used a LISP call like:

```
(fun-name v-name1 v-name2 ...)
```

as part of the RHS of one of your top-level rules. One thing to keep in mind is that the values of the variables passed by '&apply' to the function are always lists themselves. The alternative form '&funcall' passes variable values to the LISP function a little bit differently. If the variable value would be passed by '&apply' as a single element list, '&funcall' will just pass the element itself. Suppose you had used the pattern:

```
((!number := $n))
```

to capture the input fragment "7". Both '&apply' and the RHS of a top-level rule would perceive the value of '!number' to be '(7)' (i.e the list of "7"). '&funcall', on the other hand, would see the value of '!number' as "7." The distinction between the two forms is solely to make it easier to use built in LISP functions that expect atoms as arguments within '&i' patterns. A more concrete example of the use of the '&i' function calling form is:

[25] When we say "caught by", we mean that at some level the '&i' pattern must be *enclosed within* a variable assignment. The two examples above demonstrate two different ways of accomplishing this. In the second example, the '&i' is more deeply embedded, but is still part of the variable assignment operator's pattern argument. The restriction on this example would be that the nonterminal <true>, since it contains an '&i', can ONLY be used when embedded in the pattern side of a variable assignment. Otherwise, if matched it would cause a LISP error. Patterns using '&i' should not be enclosed within each other without intervening variable assignment statements.

```
((!fraction := (&i (&funcall divide (!divisor !dividend))
                      <fraction-els>))
```

with the rewrite rule for <fraction-els> being:

```
<fraction-els> -> ((!divisor := $n) %slash (!dividend := $n))
```

Here we've choosen '&funcall' because the LISP function divide expects atomic arguments. '%slash' is the way that DYPAR reads a '/'. The '<fraction-els>' rewrite rule would match user inputs like: 6/7 or 1/2. The '&i' construct would take the result from '<fraction-els>' and convert it to the equivalent floating point fraction, e.g. the user input "1/2" would be converted to ".5," which would then be assigned as the value of '!fraction' by the enclosing ':='. This conversion would be accomplished as follows: '&funcall' would see the variable names '!divisor' and '!dividend' and pass the atomic values of those variables (1 and 2 respectively) to the LISP function divide, which would return ".5" as its value. This is the value that '&i' would return to be assigned to '!fraction' by ':='.

The variables '!divisor' and '!dividend' are thrown away after use, i.e. they are local to the invocation of '&i'. The rule is that any pattern variables referred to within '&apply' or '&funcall' go away. They are not visible to the RHS.. However, any variables that are assigned within 'PATTERN' that are not referred to within an '&apply' or '&funcall' will be available later in the parse.

4.2.3. Extra Variables

Remembering our generic form for '&i':

```
((&i VALUE PATTERN))
```

Another thing that '&i' will allow you to do is omit the 'PATTERN' argument. This allows you to set up flags for your backend functions based on just where in the pattern (or in which pattern) a variable assignment was successfully completed.

```
((&i VALUE))
```

When this form is encountered, '&i' will return 'VALUE', and also *not* consume any of the user's input. This means that you can assign more than one variable at the same point in the user's input. For example:

```
((!name := <first-names> <last-names>) (!full-name := (&i t)))
```

will in addition to associating a first-name/last-name list with '!name' also set the variable '!full-name' to "t" when the first part of the pattern succeeds. The short '&i' form does not have all of the options normally available for 'VALUE'. At some point in the future this form will be extended to allow you to apply functions and reference DYPAR variables here.

4.2.4. !newvars revisited

Patterns of the form:

```
(!var-name := (&i !newvars <assignment-using-*var*>))
```

group all of the values of the '*var*' generated :var as the value of '!var-name'. It also makes sure those :var don't show up in the RHS of your top-level rule. This means you can use a different '*var*' assignment sub-pattern in your pattern without getting multiple generated '*var*' variable groups confused with one another.

Patterns of the form:

```
(!var-name := (&i (&apply fun-name !newvars) <*var*-pattern>))
```

will assign to '!var-name' the value of applying the LISP function '!fun-name' to the values associated with the '*var*' generated variables, as well as removing the :var from RHS availability.

EXERCISES

Exercise 4-1: Suppose it is decided that the rule representing yes/no questions regarding meeting times, as it is written in the answer to Exercise 2-4 in Appendix E page 63, should cover cases in which commas and/or the word "and" separates the days of the week being specified. Make the necessary changes to the grammar given as the answer to Exercise 2-4 to cover these cases.

Exercise 4-2: Modify the relevant nonterminal or rewrite rules and top-level rule that you wrote for Exercise 4-1 so that the resulting grammar is able to parse input about clock times. Then write a nonterminal that translates clock times such as 1:25pm into 24-hour-time, i.e. 13:25. To facilitate your work, a Lisp function called rdtime is defined below. This function can be used with the &i operator to produce the required translation. Make the other necessary changes so that your grammar can parse the following example, changing the inputed times to 24-hour-time.

```
      Does philosophy 1 meet on m w f at 11:10am?
```

The Lisp function follows:

Function to be used with "&i" operator to translate clock-times to 24-hour-times.

```
      (defun rdtime (hr min am-pm)

          (and (null min) (setq min 0))               ; make sure min has a value

          (and (< min 10) (setq min (concat 0 min))) ; if min is less than 10
                                                     ;   stick a zero in front

          (cond ((null hr)                           ; no value for hr
                 (list hr min))                      ; quit with hr still empty

                ((eq am-pm 'am)                      ; in the morning?
                 (list hr min))                      ; we're doing OK, stop.

                ((and (eq am-pm 'pm)                 ; in the afternoon?
                      (not (eq hr 12)))              ; but not just after noon?
                 (list (+ hr 12) min))               ; add 12 to hr

                ;; am-pm must have been empty so assume the middle of the day
                ((> hr 7)                            ; high numbers before noon
                 (list hr min))

                ((< hr 8)                            ; low numbers after noon
                 (list (+ hr 12) min))               ; add 12 to hr

                (t (list nil nil))))                 ; We should never get here.
```

Five EXTENSIBILITY

5.1. User-Defined Operators

For some DYPAR applications domains, the grammar developer may wish to add operators he/she finds useful in that domain. This amounts (in a sense) to customizing DYPAR so that it will handle specific needs in an efficient manner. User-defined operators come in two varieties:

- simple extensions (e.g., niladic operators)

- complex extensions

These operators are referred to as *extensions* because they are actually pointers to external functions[26] that are called when the associated symbol is encountered in a pattern. The simple extensions that are included as a part of DYPAR are discussed below. Complex extensions will be dealt with in a later version of this manual.

The procedure for adding new niladic operators is quite straightforward. You must define a LISP function that takes one argument which will be a list of the unconsumed input. Your function should be a predicate on the car of its argument (the unconsumed input list.) One example of such an operator could be a function $q that returns t for any word beginning with the letter "q". We will develop this example further in Section 5.3, where guidelines are provided for writing functions that meet your particular needs. First, though, we shall describe some predefined extension functions that are provided with DYPAR.

5.2. Predefined Extension Functions

The code for the functions listed below can be found in the DYPAR file extend.l.

The external functions provided with DYPAR can usually be invoked by any one of a group of synonymous names. The examples provided with this listing are designed not only to demonstrate the use of these functions but also to show how using extension operators can make a grammar more flexible.

$p -- will match any punctuation character.[27] (Other names for $p are $punctuation and $punct.)
For example, returning to our sample grammar, a pattern to match the sentence, "Fred is a jock!" could look like this:

```
((!nam := $) <be> ?<det> (!jock := (athlete | jock)) ?$p)
```

(Note that this pattern will also match permutations such as "Fred has been an athlete.", "Bill was the jock", and so on.)

$w -- matches any word ($word is a synonym for $w.)
Suppose we want to write a pattern with a variable-binding that will accept any input so long as it is a word, but will not accept punctuation characters or numerals. For instance, to match the answer to a specific question (such as, "What is your favorite color?") you might want to parse quite diverse replies like "Green", "maroon.", "My favorite [color] is azure!", "I like red the best.", and so on:

[26] Such pattern invoked functions are occasionally called "daemons", since they are invoked by the presence of certain data, rather than built into the interpreter.

[27] Appendix A, on page 49, contains a full listing of the punctuation characters recognized by DYPAR, together with their Ascii values and DYPAR symbols.

```
<i-my> -> (i | my)[28]
<most> -> (?the best | ?the most)
<prefer> -> (like | prefer)
<favorite> -> (favorite ?color <be>)

(?<i-my> ?<most> ?<prefer> ?<favorite> (!val := $w) ?<most> ?$p)
```

$w is quite useful when it is necessary to differentiate between words and numbers in some input.

$d -- matches any word in the "dictionary."

The dictionary is a list of all the words[29] (terminal symbols) in a given grammar, including all the words appearing on the RHS of any rewrite rule. Suppose you wrote a top-level rule whose RHS executed a search function that would tell you where in a given grammar a certain word appeared. Then the LHS (pattern) for the rule might look like this:

```
<find> -> (find | search ?for | look ?for ?up)
<where> -> (where ?<be>)
<gra> -> (grammar | ?grammar file)

(?<polite> ?<info-req2> (+ (<find> <where>)) (!term := $d) ?<be>
        in ?the ?<gra> (!filename := $)  ?$p)
```

This pattern would match requests such as, "Could you please tell me where [word] is in the file [filename]?", "Find [word] in [filename]", "Please look up [word] in the grammar [filename]."

$lisp-function -- matches words that have function definitions in LISP.

$lisp-function checks to see whether or not its input is a LISP function name. This operator can be used to match a function name in user input and bind it to a variable: then the value of the variable could be accessed by the RHS of the rule and the LISP function executed, or manipulated in whatever way the grammar writer desires. (Another use of $lisp-function is discussed in Chapter 7, "Advanced Topics.") If parenthesis are used in the user input, they must be individually matched in the pattern using %lparen and %rparen, or an appropriate wildcard.

For example, suppose you were using DYPAR as the language interface to a tutorial program that teaches people how to do simple arithmetic in LISP. The tutorial has just asked the user,"What LISP function would you use to add 3 and 6?" (The correct answer is (PLUS 3 6).) A pattern to match the user's reply might look like this:

```
((&u* %lparen) (!lspfun := $lisp-function) (!arg1 := $n)
        (!arg2 := $n) $r)
```

This pattern assumes the user will include the parentheses which are properly part of the function and exploits their presence to find the relevant portions of the input.[30] If there is a good chance that the user will type in something that is *not* a LISP function in this kind of input, then the pattern should make use of a less-specific wildcard (such as $w) instead of $lisp-function. Otherwise the parse would fail under such circumstances.

$lisp-variable -- matches atoms that have values in LISP.

[28] Remember, only lower-case letters may be used in writing patterns, though upper-case *user* input is accepted.

[29] By "word", we mean plain English words that are present in the grammar to match the user's input, whether they are in a rewrite rule or put directly into a pattern. Nonterminals, operators and variable names are not words in this sense.

[30] If this is not assumed, the pattern must be a lot more complex, in order to match a variety of anticipated responses and pick out the function; for instance:

```
<func> -> (?lisp function)
<equals> -> (?<pos-modal> <be> | ?equals)
<answer> -> (?<det> (answer | <func> | it) ?<equals>)
<use> -> (?<pos-modal> (use | answer))

(?<i-my> ?<use> ?<answer> ?%lparen (!lspfun := $lisp-function)
        (!arg1 := $n) (!arg2 := $n) $r)
```

$lisp-variable checks to see if its input is a DYPAR variable name or (using the LISP function BOUNDP) has a value. For instance, this operator will match atoms which have been given values using the LISP function SETQ. In a situation where a list of information has been bound to some atom using SETQ, $lisp-variable could be used to match such an atom in the user's input. The RHS of the rule could then go off and retrieve the information, or perform whatever function was desired by the grammar-writer. As an example, suppose you wanted to match user input requesting a list of names, e.g., "Please print the list FileNames"; "could you give me the list filenames?"; "Please show me Filenames.", etc.:

```
<display> -> (get ?me | show ?me | <info-req2>
                | display | <find> ?me)

(?<polite> <display> ?the ?list (!nam := $lisp-variable) ?$p)
```

The '$n' operator introduced earlier is implemented as an extension function.

5.3. Simple Extensions

In this section we will provide basic guidelines, and a few examples, for defining new operators which are simple extension functions. Note that if you wish to make extensions using functions that you write yourself, they must be contained in a separate file from the grammar, and they must be explicitly loaded into the LISP environment before loading the grammar file.

Now we shall return to our hypothetical user-defined operator from Section 5.1, $q. This operator is defined as a predicate on the car of its input, which returns t for any word beginning with the letter "q". Adding this function to our sample grammar, you could match the input

```
Susan will have the quilt for us.
```

with a pattern such as

```
((!nam := $) <have> ?<det> $q <prp> <me-us> ?<punct>)
```

Following the input as it is matched to the pattern, we find that by the time we reach $q in the pattern, the input consists of "quilt for us." The car of the input is therefore "quilt", of which the first atom is "q", which satisfies the requirements of the operator $q and is matched by it.

$q is an example of a niladic operator written as a simple extension. It consists of a symbol, '$q', and an underlying LISP function. The symbol is associated with the function using the LISP function defprop. Of course, $q is a hypothetical function (although the reader may choose to define it as an exercise; see "Exercises" at the end of this chapter.) Below is an example which explains the workings of an actual built-in DYPAR operator, $p.

The function associated with the symbol '$p' is called 'recognize-punct'. It is defined as follows:

```
(defun recognize-punct (l)
    (member (car l) !!allpunct))
```

This function works by checking whether the first element in "l" (the input utterance) is a member of global list '!!allpunct', a list of all the DYPAR symbols which are equivalent to punctuation characters.[31] If the membership test is successful, the pattern element '$p' is considered matched.[32]

The symbol '$p' is attached to its function by means of defprop. The form for this is:

[31] For an explanation of how DYPAR reads punctuation characters, see page 9. See Appendix A, page 49 for a list of punctuation characters with their DYPAR equivalents.

[32] Note that this function does not return 't' if (member (car l) !!allpunct) succeeds. The LISP function member checks to see if its first argument is an item in the list which is its second argument. It either returns nil or a portion of the list, beginning with the item it searched for. What the pattern-matching function looks for is a non-nil argument, so if member succeeds, the list it returns will cause the pattern matcher to consider the match successful. All this means is that for purposes of writing simple extensions, any non-nil value is as good as a 'T' value for your function to return.

```
(defprop $p recognize-punct :function)
```

That is, the symbol '$p' is assigned 'recognize-punct' as the value of its ':function' property.

To aid the cross-referencing routine in compiling a grammar, each extension function should have an associated ':symbol' property with a value of either '$' if the function is one which is meant to be used on non-numeric input, or '$n' for those functions used for numeric inputs.

```
(defprop $p $ :symbol)
```

The values for the ':symbol' properties in the above example are solely for the use of the cross-referencer when an extension function is encountered during the loading of the grammar. If user-defined extension functions do not have one of these values associated with them, then the value will default to '$'.

To summarize, user-defined simple extension operators should:

- use a function taking one argument, which acts as a predicate on the car of its input
- attach the symbol to the function using defprop
- assign the operator a ':symbol' property, either '$' or '$n', using defprop
- have the code contained in a separate file, which is explicitly loaded into the LISP environment before the grammar file

5.4. Complex Extensions

The DYPAR operator &i, discussed in Chapter 4 on page 27, is implemented as a complex extension function. It will be described here when this section is added in a later version of the manual. (See Chapter 1, page 2 for information on obtaining newer versions.) Examining the source code for &i in the files extend.l and xmatch6.l may help the "do-it-now" developer.

EXERCISES

Exercise 5-1: In this chapter we have used a theoretical operator, Sq, as an example of a simple extension. Sq matches any word beginning with the letter q. Write and test Sq. (You may want to review the guidelines and look at the code for other simple extension functions that is contained in the DYPAR file extend.l)

Exercise 5-2: Write a function 'Spe' that matches the punctuation marks period, exclamation point and question mark ('.' '!' '?'). Appendix A, page 49, contains a table of the internal DYPAR symbols for each punctuation character.

Six

INSTALLATION

6.1. Source Files

The LISP source code for DYPAR is broken up into a number of files each dealing with a particular aspect of the parsing system. The files are:

- ✓ extend.l contains extension functions which have been provided with the parser in addition to those "hard wired" into the pattern matcher.

- fload.l contains a fast grammar loader for precompiled grammars.

- ✓ general.l contains utility functions used throughout DYPAR.

- ✓ macros.l contains utility macros used throughout DYPAR.

- readl.l contains DYPAR I/O routines, including the function which does punctuation expansion.

- semfns.l contains the semantic network used as a sample backend for illustrative purposes in this document.

- ✓ var.l is a file containing global variable initializations.

- ✓ xload.l contains the grammar loader.

- ✓ xmatch6.l contains the pattern matcher.

- ✓ xpar.l contains the DYPAR top-level.

- ✓ xrefnew.l contains the DYPAR cross-referencer, which is used to build the discrimination network DYPAR uses to decide which rule(s) to try.

- sem.gra contains the sample grammar described in this document.

- rundemo.l contains a boot file for starting a DYPAR image with the sem grammar included as well as the semantic network from semfns.l.

- boot.l contains a boot file for starting up a DYPAR image with no preloaded grammar.

- dybanner.l contains version information and a startup message for using dumplisps of the DYPAR-I system.

6.2. Compiling the Sources

You should read the file "Readme" which has been shipped with the DYPAR release you have.

All of the LISP files compromising DYPAR-I are compilable using version 6.0 of the FRANZ LISP compiler with the CMU environment loaded. You should compile the macros.l file before proceeding to any of the others. After macros.l has been compiled, the order in which you compile the rest of the files is unimportant.

6.3. Installing DYPAR

Once the sources have been compiled, you need only start up a FRANZ LISP image, and when you see the LISP prompt do:

```
(load 'rundemo)
```

or

```
(load 'boot)
```

depending on whether or not you wish to use the sample system and grammar provided with DYPAR (rundemo.l) or to develop your own grammar (boot.l).

If you have disk space available on your machine, you might want to dump a core image of DYPAR and save the time you would normally spend waiting for the system to load. To do this type:

```
(dumplisp dypar)
```

to the LISP prompt[33]. (You need not call the dumped image "dypar", if you prefer another name for it. A dumped LISP will preserve everything in the core image; since you have already loaded DYPAR into the LISP environment, it will be there no matter what you call the file it is stored in.)

6.4. Loading DYPAR Grammars

6.4.1. Loadgra

The command used to load a DYPAR grammar is:

```
(loadgra 'grammar-file-name)
```

This function calls both the grammar loader and the cross-referencer. You can also give multiple file names to loadgra, in order to load more than one grammar file:

```
(loadgra 'file1 'file2 'file3 . . .)
```

6.4.2. Loading Saved Grammars

The command described in this section will not work under VMS, and the UNIX version is not completely debugged.

Loading grammars with the loadgra command can be time consuming. Relatively stable grammars can be loaded more quickly by using the functions savegra and faslgra. To do so enter DYPAR-I and load a grammar. After the grammar has been loaded, type:

```
savegra root-file-name
```

and a file called root-file-name.gra.flg will be created. This saves a copy of the cross-referenced grammar which is more quickly loadable. At any future time this .flg file can be loaded into a LISP image by typing:

```
faslgra root-file-name.flg
```

[33] This works with FRANZ LISP only under UNIX. For FRANZ LISP under VMS, type (savelisp dypar).

6.5. Running DYPAR

Once the DYPAR package has been loaded, it can be invoked simply by calling the function 'parse'. 'Parse' will prompt the user for an input, attempt to match that input against its loaded grammar, and return the value of that parse to its calling function (or the top-level, if that's where it was called from). 'Parse' should be used if you wish to call the DYPAR system from another function.

Another function, 'parser', is useful for grammar development, or for systems where DYPAR is to be used as the top-level function. It will parse utterances in a loop until it is explicitly told to exit. There must be an exit rule contained in the grammar for this to work properly:

```
(exit | quit | ?good bye)
=>
'exit
```

If for some reason you have forgotten to include an exit rule, you can bail out by typing an interrupt character to the DYPAR prompt. This will generate a LISP error and put you into a break loop.

6.6. Variables to Control Behavior

- '!ptrace' will, when non-nil, enable primitive tracing of the parse in progress. Its default setting is t; to turn off the tracing, type (setq ! ptrace nil) before using the parser.

- '!failure-flag', when non-nil, will cause failed parses to be recorded in a file in the directory [CAD]/usr/xcalibur/comments. This feature is only implemented in the FRANZ LISP version of DYPAR running under UNIX. Its default setting is nil; to turn it on type (setq !failure-flag t)[34]

6.7. Using Dypar

This section is a transcript of an actual DYPAR-I session. User's input is italicized. A LISP image containing DYPAR-I has been saved under the name "dypar1", so our first command (to UNIX) is to run DYPAR.

```
%dypar1

Welcome to DYPAR
1.
```

The next thing we must do now that we have a LISP prompt is to load a grammar into our environment. We will use the grammar developed earlier in this document.

[34] CMU local capability only on CMU-CS-CAD.

```
1.(loadgra 'sem.gra)
Loading Grammar File: sem2.gra
::>::>::>::>::>->->->->->->->->->->->->->->->->->->->->->->->
->->->->->->->->->->->->->->->->->->->->->->->->->->->->->->-
>->->->->->=>=>=>=>=>=>=>=>=>=>=>=>=>=>=>=>=>=>=>=>=>=>=>=>=>
=>=>=>=>
Semantic Grammar Loaded.

X-referencing => top-level rules:
RoRRRonononoooononononnonoonoRonnononoononononoonRonononnnnoonnonRonnnnno
oooonRonnnnonRonnnonnoonnonRonnnnonnoononRooRoooRoooRoooRooRooRonnonooo
nonoonnooooRRooooRonconooooonoononononnononooonRononoooRnooooRonnoooonooon
oononRonnoooonoRonnooonnonnonRononoRnonononnnnonnonnoonoonoonoononRnon
nonnnoonnonRnonnonnonnonRnonnonnonRnononnonnnonnonRnonnonnonRnonnonRnon
no

X-referencing ::> transformation rules:
RonoonoonoononononnnnnnnonnonnooooonRooonooooonRoonoonnooRoononoononoonRoo
oo

X-referencing -> rewrite rules:
RonoRononoonoonnoRoonnoRoRoRoRoRonRnono
Following[35] non-terminals have been seen, but have no definitions:
((<find>))

Indexing => top-level rules:
F-F-L-L-F-L-F-L-F-F-F-F-F-F-F-F-F-F-L-L-L-L-F-F-F-F-F-F-F-F-F-L-L-L-L
-L-L-F-F-F-F-F-F-L-L-L-L-F-F-F-F-F-F-L-L-L-L-L-L-L-F-F-F-F-F-F-F-F-F-
L-L-L-L-F-F-F-F-F-F-F-F-F-F-L-L-L-L-F-F-F-L-L-L-L-F-L-F-F-F-L-F-F-F-F-L
-L-L-L-F-F-F-F-L-F-L-F-L-F-F-F-F-F-F-F-F-F-F-L-F-L-F-F-F-L-F-F-F-F-F-F-
F-F-F-L-F-L-L-L-F-F-F-F-F-L-L-L-L-L-F-F-F-F-F-F-F-F-F-L-L-L-L-L-F-F-F-F
-F-F-F-L-L-F-F-F-F-F-F-F-F-L-L-L-L-L-F-F-F-F-F-F-F-L-L-L-L-L-L-L-F-F-F-
F-F-F-F-F-F-F-L-L-L-L-F-F-F-F-L-L-L-L-L-F-F-F-F-L-L-L-L-L-F-F-F-F-L-L-L
-L-L-F-F-F-F-F-F-F-F-F-F-L-L-L-L-L-F-F-F-F-F-F-F-F-F-F-L-L-L-L-L-F-F-F-
F-F-F-F-L-L-L-L-L-F-F-F-L-L-
Cross-referencing Completed.
t
2.
```

This output provides information about the grammar loading process. First DYPAR loads each rule in the grammar, printing out a table of the rule-type symbol associated with each rule it encounters in the grammar file (e.g., ::>::>::>::>::>->->->->->=>=>=>=>=>). When the entire file is loaded, DYPAR prints "Semantic Grammar Loaded", and calls the cross-referencer on the rules in the file. It works on the top-level rules first, then the transformation rules, then the rewrite rules. After all the rules have been cross-referenced, the cross-referencer will return a list of any nonterminals used in the rules which were not defined in a rewrite rule. In a complete grammar, there should not be any instances of undefined nonterminals, but in a grammar which is still under development, this information can be very useful. Lastly, the sentence templates in all the top-level rules are indexed, using a system which records the first and last elements of each template. The table thus formed is used by DYPAR to look up matches for English inputs: it will only try to match an input to those templates whose first and last elements match the corresponding elements in the input. (This procedure speeds up the matching process by eliminating the need to attempt complete matching of every top-level rule until the correct template is found.) When the cross-referencer is finished, it prints the message "Cross-referencing Completed" and returns a t. Now we are left at a LISP prompt and can call the parser.

6.7.1. Inputting Sentences

Now we can invoke DYPAR and type some sentences to it. During this session the value of the !ptrace variable was t.

[35] <find> is a gratuitous undefined nonterminal which was added to our sample grammar in order to demonstrate this feature of DYPAR; see the text which follows this example for an explanation.

```
2. parser

You are talking to the top-level Semantic Parser.
English sentences will be parsed in a loop.


+ mary is an architect
I will try rules:  (rul23 rul19 rul15 rul9 rul14 rul16 rul18 rul20)
Parse is:

Rule rul14
Action will be: (ltm-store !nam !val 'isa: nil nil)
With bindings:
        (!val architect)
        (!nam mary)

Storing assertion in semantic net:   mary is an architect.
Inference:   the concept-type of architect is generic.


+ mary's mother is sally
I will try rules:  (rul23 rul19 rul15 rul9 rul14 rul16 rul18 rul20)

transforming  (mary %apost s mother is sally)
Into ::>      (mother of mary is sally)

transforming  (mother of mary is sally)
Into ::>      (sally is mother of mary)
I will try rules:  (rul23 rul19 rul15 rul9 rul14 rul16 rul18 rul20)
Parse is:

Rule rul15
Action will be: (ltm-store !nam !val !prop nil nil)
With bindings:
        (!nam mary)
        (!prop mother)
        (!val sally)

Adding new assertion:   the mother of mary is sally.


+ tell me all about mary
I will try rules:  (rul8 rul3 rul2 rul4)
Parse is:

Rule rul8
Action will be: (ltm-ret-all !nam)
With bindings:
        (!nam mary)

 mary is an architect.
 the mother of mary is sally.

+ who is mary
I will try rules:  (rul8 rul3 rul1 rul4)
Parse is:

Rule rul1
Action will be: (ltm-ret !nam 'isa: nil nil)
With bindings:
        (!nam mary)

 mary is an architect.
```

```
+ the inverse of husband is wife
I will try rules:  (rul20 rul18 rul16 rul15 rul19)

transforming  (the inverse of husband is wife)
Into ::>      (wife is inverse of husband)
I will try rules:  (rul23 rul19 rul15 rul9 rul14 rul16 rul18 rul20)
Parse is:

Rule rul16
Action will be: (ltm-store !nam !val !prop nil nil)
With bindings:
        (!nam husband)
        (!prop inverse)
        (!val wife)

Adding new assertion:   the inverse of husband is wife.



+ mary's husband is bob
I will try rules:  (rul23 rul19 rul15 rul9 rul14 rul16 rul18 rul20)

transforming  (mary %apost s husband is bob)
Into ::>      (husband of mary is bob)

transforming  (husband of mary is bob)
Into ::>      (bob is husband of mary)
I will try rules:  (rul23 rul19 rul15 rul9 rul14 rul16 rul18 rul20)
Parse is:

Rule rul15
Action will be: (ltm-store !nam !val !prop nil nil)
With bindings:
        (!nam mary)
        (!prop husband)
        (!val bob)

Adding new assertion:   the husband of mary is bob.
Asserting inverse relation:
Adding new assertion:   the wife of bob is mary.



+ exit
I will try rules:  (rul13)
Parse is:

Rule rul13
Action will be: 'exit
With bindings:


Leaving natural language interface.

  Back to LISP.
t
6.
```

Seven MULTIPLE MATCHES

7.1. Non-Deterministic Parsing

The approach to parsing taken by DYPAR is non-deterministic. What this means is that even when the parser has found a solution (i.e. successfully matched a pattern) the parser will continue to try other solution paths until there are no more to try. That isn't quite as bad as it sounds because the cross-referencer limits the paths the parser is allowed to try. However, what does happen is the parser sometimes finds more than one solution to a parse. A parse with more than one solution is *ambiguous*. When a single top-level rule can match a user's input in more than one way, it is *internally ambiguous*. When two or more top-level rules can match the same user's input they are *externally ambiguous*. Multiple matches for the same input can also be referred to as *match collisions*.

7.2. Internally Ambiguous Matches

Most of the time the existence of multiple matches will not have any effect on the parse, as the variable bindings will be the same for all of the solutions. As long as the variable bindings are the same DYPAR just removes the redundancy from the parse solution set. Sometimes, though, the variable values for the solutions are different. DYPAR must then decide which solution is the *correct* one. DYPAR tries to resolve the conflict using the following collision resolution strategies, stopping when the application of the strategies reduces the number of solutions to one:

1. Pick the solution(s) that contains the largest number of variables assigned.

2. Pick the solution(s) that contains the largest number of variables with values other than nil.

3. Pick the solution(s) that contains the largest part of the user's input as the values of variables.

4. Pick the first solution. (This is guaranteed to make a choice!)

7.3. Externally Ambiguous Matches

This condition is a little more serious than the internal variety. Here, we have two different top-level rules (probably with different actions) vying for the same user input. Again, DYPAR tries to make a choice as to which solution is the correct one. The same collision resolution strategies that are used for internally ambiguous matches are use to recover from externally ambiguous matches, with the exception of the last strategy. Instead of picking the first solution, DYPAR will ask the user which of the solutions is the correct one.

7.4. Changing Default Behaviour

If this behaviour is unacceptable, DYPAR allows you to modify the built-in collision resolution mechanism. There are two ways to do this: You can reorder the way the existing strategies are used; You can write LISP function(s) describing entirely new collision strategy(ies). Both methods are accomplished using the meta commenting facility for DYPAR rules. The keyword for internally ambiguous match strategies is:

```
:internal-strategy
```

and its use is described below.

7.4.1. Reordering Methods

Each of the existing strategies has a name. When you make a meta comment you simply list the strategies in the order you wish them to be tried. The names of the strategies are, respectively:

- most-vars
- most-non-nil
- most-input
- arbitrary

Say for example, that you wish to choose the possibility which consumed the most input, and otherwise you don't care. You would place the meta-comment

```
(:internal-strategy (most-input arbitrary))
```

just before the pattern definition for your rule.

7.4.2. New Strategies

For this you should write a LISP function which will make the decision. The data structure it must chew on looks like:

```
(((lvariable1 value1) (lvariable2 value2) ...)
 ((lvariable1 value3) (lvariable2 value4) ...)
 ...)
```

1st possibility

2nd possibility

To tell DYPAR about your new strategy your meta-comment should look like:

```
(:internal-strategy my-function-name)
```

Note that when you reorder the existing strategies DYPAR is expecting you to surround the new list with parentheses, and when you define your own function you should omit the parentheses around the function name when you give it to "internal strategy:".

Eight ADVANCED TOPICS

8.1. About This Chapter

In this chapter we explain more about the inner workings of DYPAR, and introduce three more operators. It is presupposed that the reader knows LISP. The topics covered here are those which were not extensive enough to warrant separate chapters in the reference manual; however, much of this material will be helpful in writing efficient patterns. The section on debugging grammars is not meant to be exhaustive, but rather to give specific examples of some general procedures.

8.2. Deterministic Disjunction

One operator which hasn't been covered yet is the deterministic disjunction operator '!!'. Its syntax is the same as that of '|'. The difference is that '!!' stops after the first successful match of one of its arguments. It is easy for someone to get into trouble by using '!!' when he means '!'[36] For instance,

```
(a (b !! (* $)) d)
```

will fail on "a b c d" because the b matches locally, but it fails on the next element in the pattern. Of course:

```
(a (b | (* $)) d)
```

will work. However, '!!' is more efficient when properly used.

8.3. Deterministic Optionality

The deterministic optionality operator &o forces an optional element to be consumed wherever possible. It works in the same way as '?', except that the match does not branch when a successful match is made. In order to understand the difference between them, we need to examine how '?' works.

When the pattern argument to '?' succeeds, two branches are created, one with the pattern argument matched to the next section of input, and one with the pattern argument matched to nothing at all. The second branch has the effect of causing the match to succeed without consuming any input, *even if input existed which matched the pattern argument to '?'.* Following the first branch, we find the pattern matcher moving on and attempting to match the next piece of the pattern with the next piece of input, while following the second branch, it attempts to match the next piece of the pattern with the same piece of input it used for the previous pattern (which contained the '?' operator.) In most cases, it is the first branch that succeeds in completing the parse, and the second one will fail at this point. However, consider the pattern

```
(?very very much)
```

If this pattern is matched to the input "very very much", the first branch of the match is followed through, and the second, since it fails to consume both repetitions of the word "very", fails. But if we match this pattern to the input "very much", it will succeed by following the *second* branch of the match. If this second branch did not exist, the '?' operator's pattern argument would cause the word "very" to be consumed, and the match would fail when the next, nonoptional, pattern element "very" failed to match the next piece of input, "much". You can see that this multiple branching is necessary to cause the '?' operator to work as it should under all conditions.

Now we are ready to consider the deterministic optionality operator, '&o'. Its syntax is the same as that of most non-niladic DYPAR-I operators, which means that the operator is enclosed in parentheses along with its pattern argument:

```
(&o very)
```

[36] '!' is a synonym for '|', as well as being used to mark variable names. See Table 2-2, on page 15.

It should be noted that this is *not* the syntax used with the '?' operator, which simply works on the expression immediately following it. The pattern above will match the word "very", or nothing at all, as you would expect from the '?' operator as well. However,

```
((&o very) very much)
```

will match "very very much", but *not* "very much". The &o very portion of the pattern is forced to consume the first "very" to which it is matched, and if there is no second "very" to be matched to the rest of the pattern, the match will fail.

The '&o' operator is most useful in constructions such as

```
((&o word) (!var := $))
```

to ensure that !var does not end up with some optional constituent as its value in an essentially spurious parse. With the use of niladic operators, and particularly their use in conjunction with an iterative operator, that portion of the pattern following a use of '?' might be flexible enough to cause both branches of the match to succeed, when only the first was intended to succeed. If there is a variable involved, it could be assigned the wrong value. In such situations, it is safer to use '&o'.

8.4. Internal Grammar Representation

When a grammar rule is read into the LISP environment by the grammar loader, it is converted from the form used by the grammar writer to a more LISP-like syntax. In this section, we will explain the meaning of the internal format, which is what you see if you examine a rule or pattern after you load it into a LISP core image.

When you have a pattern of the form:

```
(?an example)
```

it will be stored as:

```
((? an) example)
```

Complex disjunct patterns like:

```
(a | <foo> | b c)
```

would be converted to:

```
(! (a) (<foo>) (b c))
```

Simple disjuncts, e.g.

```
(a | b | c)
```

are converted to:

```
(&m a b c)
```

We haven't covered '&m' before, because it is an *internal operator*. Its purpose is to make the match on a disjunct set, where all of the options are terminal atoms, more efficient.

The case operator '&c' when used in the form:

```
(<over> &c and &c <through>)
```

gets converted to:

```
(&c (<over>) (and) (<through>))
```

Variable assignments change from:

```
(!color := <color>)
```

to:

```
(:= !color <color>)
```

Patterns using the not operator '~' like:

```
    (~a dog)
```

convert to:

```
    ((~ a) dog)
```

Invocations of '*', '+', '↑', '=', '&u', '&n' and '&i' retain the same form. The *niladic operators* '$', '$n', and '$r' take no arguments and also retain the same form. Generally what happens is that any operators which are expressed in infix form in the grammar file are converted to a prefix notation, and precedence of operators is made explicit.

8.5. Pattern Storage

Patterns are stored on property lists. LISP stores the information contained in a DYPAR grammar, as well as that generated by the cross-referencer on property lists. This information is accessed using the LISP function *get*. Top-level and transformation rules generally use the DYPAR-generated rule name as the key to which the other information is attached. Information for rewrite rules is attached to the LHS (nonterminal) of the rule. The following properties are used by DYPAR:

- Top-level and Transformation Rules:
    - pattern: LHS or pattern of the rule
    - action: RHS or action for the rule
- Rewrite Rules:
    - rewrite: RHS or pattern
- Extension Functions:
    - function: name of the function to invoke
    - symbol: token to be used by the cross-referencer
- Terminal Symbols:
    - where: top-level rules that this symbol can be the first element for
    - lwhere: top-level rules that this symbol can be the last element for
- All Rules:
    - first: set of words that can be the first elements for this rule
    - last: set of words that can be the last elements for this rule
    - opt: flag as to whether this rule can succeed while matching nothing

8.6. Cross-Referencing

DYPAR uses a lookup table to decide which rules might successfully match the input utterance.

- The table is indexed by terminal symbols (words) which can start or end a pattern.
- The current word of the input is compared against the table entries for all rules that are currently active.
- Only those rules that have index entries equal to the current input word receive further processing.
- For each of these rules the index entry for its ending words is retrieved and intersected with the input string.
- Only rules which return non-nil intersections are expanded by the matcher for a typical, moderately-sized grammar.

This forward pruning results in a factor of 10 speedup in the performance of the matcher over iterating through all the rules.

8.7. Punctuation in Depth

DYPAR treats punctuation characters typed by the user in a special way. No punctuation characters are read in verbatim. They are converted to special symbols, usually prefixed with a percent '%' sign.

One reason for specially treating punctuation characters is to avoid conflicts between what the user types and what LISP thinks of as special characters (such as ';' and ','). Also, a person typing input to a system like DYPAR seldom types spaces between punctuation characters and the adjacent words. This approach allows user inputs like "$450" to be treated as the two words "%dollar 450", which is much easier to parse. If some of these conversions are found to be intolerable, they can be turned off by modifying the contents of the global variable !!delimiters, so that it doesn't contain that character.

```
(setq !!delimiters (delete 45 !!delimiters))
```

would remove '-' (ascii 45, %dash) from the set of punctuation characters that are normally expanded. !!delimiters contains a list of the base 10 ascii values of the pertinent punctuation characters. A list of all of the punctuation characters affected by DYPAR can be found in Appendix A, on page 49.

8.7.1. Debugging Grammars

It turns out that the grammar described in Chapter 3, starting on page 17, and reproduced without accompanying text in Appendix C, page 55, has a bug in it. The file loaded above was not the one developed in the manual, but a very similar one previously written. This section describes some LISP functions which are useful in testing and debugging grammars that are accessible in the LISP image containing DYPAR-I. They will be introduced by being used to debug the grammar described in the manual.

Suppose that the grammar described in the manual has been loaded into a LISP image containing DYPAR-I, and the parser has been called. On typing the first input, as shown above, the following result is obtained:

```
+ mary is an architect
I will try rules:  (rul19 rul13 rul11 rul9 rul10 rul12 rul14)
No parses found for:
(mary is an architect)
```

In fact none of the input in the previous section will be parsed by the grammar as it is presently written. This is somewhat puzzling, since an earlier version of this grammar was able to parse all of these sentences.

Whatever error has been made, it seems to have permeated through much of the grammar. In order to check the grammar rules more directly, it is necessary to exit the parser and get back to LISP. After doing this it must be decided which rule pattern or part thereof should be tested. Testing the top-level rule pattern which should have parsed the input tried above is as good a place as any to start. The LISP functions useful in debugging grammars are able to identify top-level rules by their numerical place in the grammar file. Looking at the grammar as reproduced in Appendix C, the top-level rules can be counted beginning on page 55 and ending on page 57. The top-level rule to be tested is rule 11. It appears on page 56. The first step to be taken is to see what rule 11 presently looks like. It can be accessed by typing ppl or plist in the following way with the following result.

```
6.ppl rul11
rul11
pattern: ((? <a-an>) (:= !nam $) <be-pres> <a-an> (? <typeof>)
          (:= !val $) (? <dpunct>))
action: (ltm-store !nam !val 'isa: nil nil)
first:  $
last:   $
rul11
```

As can be seen, the rule pattern is in a somewhat different form than the one typed into the grammar file. This is discussed in Section 8.4 starting on page 44. It can also be seen that for the input *mary is an architect* only two nonterminals in the rule pattern come into play: '<be-pres>' and '<a-an>'. Taken together these should parse the input "is a." This can be tested using the LISP function 'xmatch' in the following way:

```
6.(xmatch '(is a) '(<be-pres> <a-an>))
nil
```

An entire top-level rule pattern could be tested in this way, either by typing the pattern as it is printed after a ppl or by specifying the rule number in the following way:

```
(xmatch '(mary is an architect %period) (get 'rul11 'pattern:))
nil
```

Notice that punctuation characters cannot appear in 'xmatch', only the corresponding special symbol for the punctuation character.[37] Getting back to the problem at hand, what has been shown so far is that '<a-an>' or '<be-pres>' are not covering the input they are supposed to cover. If they are tested separately, only '<be-pres>' does not do its job.

```
7.(xmatch '(a) '(<a-an>))
((1))
8.(xmatch '(is) '(<be-pres>))
nil
```

The "((1))" above indicates that one element has been matched by the pattern. If more elements were matched, the number would be correspondingly greater. In any case the problem has now been narrowed to finding out what is wrong with '<be-pres>'. Using a good screen editor or even just perusing a printed version of the grammar will quickly lead to the discovery that '<be-pres>' has not been defined. Rather, the nonterminal defined in the grammar file to do this job is '<be-present>'. Since '<be-pres>' has been used where '<be-present>' should have been throughout the grammar file, it becomes understandable why so much parsable input would not be parsed by the grammar described in the document. After '<be-pres>' is changed to '<be-present>', the grammar parses input as it is supposed to.

A primitive tracing mechansim exists, which is enabled by setting the variable !show-expanded to t in the LISP environment. This trace will print nonterminal names as they are expanded, and show at what level of recursion the expansion took place. (It can, of course, be turned off again by setting the variable to nil.)

Lists of a grammar's top-level rules, nonterminals, and transformation rules can be accessed through the evaluation of the three corresponding s-expressions: '!!patrules', '!!nonterms', and '!!pattrans'. This can be helpful in debugging grammars, especially in the case where a grammar file fails to load. This usually occurs because of a missing or misplaced parenthesis. The grammar file will load into a LISP environment until it becomes confused by a missing or misplaced parenthesis. Such a problem can be traced to where the file stops loading. For example, if top level rule 14 was missing a parenthesis, then typing '!!patrules' would have the following effect:

[37] See Section 8.7, page 46 and Appendix A, page 46.

```
9.||patrules
(rul1 rul2 rul3 rul4 rul5 rul6 rul7 rul8 rul9 rul10 rul11 rul12 rul13)
```

Appendix A
PUNCTUATION CHARACTERS

DYPAR symbol	Character	Ascii Value
%colon	:	58
%dash	-	45[1]
%slash	/	47
%apost	'	39
%hash	#	35
%comma	,	44
%lparen	(	40
%rparen	)	41
%star	*	42
%bquote	`	96
%rsbrack	[	91
%lsbrack	]	93
%bslash	\	92
%vbar	|	124
%semicolon	;	59
%dquote	"	34
%lcbrack	{	123
%rcbrack	}	125
%labrack	<	60
%rabrack	>	62
%amper	&	38
%percent	%	37
%dollar	$	36
%plus	+	43
%equal	=	61
%underbar	–	95
%upcaret	↑	94
%atsign	@	64
%tilde	~	126
%emark	!	33
%qmark	?	63
%cr	↑M	13[2]
%lf	↑J	10[3]
%period	.	46

[1] This conversion is commonly disabled.

[2] Also known as "carriage return."

[3] Also known as "linefeed."

Table A-1: Punctuation Character Mappings

Appendix B
DYPAR QUICK REFERENCE GUIDE

<table>
<tr><th>Desired Operation</th><th>Symbol</th><th>Function</th><th>Section</th></tr>
<tr><td>Disjunct List</td><td>|</td><td>Match any pattern in a list of patterns, separated by the disjunction operator:
(is | are | be | am)</td><td>2.1.1</td></tr>
<tr><td>Deterministic Disjunction</td><td>!!</td><td>Same as '|', but stops after first successful match of its arguments:
(is !! are !! be !! am)</td><td>8.2</td></tr>
<tr><td>Quick Disjunction</td><td>&m</td><td>Makes matches on disjunct sets containing terminal atoms:
(&m is are be am)</td><td>8.4</td></tr>
<tr><td>Optional Element</td><td>?</td><td>Optionally match the pattern immediately following the optionality operator:
(?(?will have | has) had)</td><td>2.1.1</td></tr>
<tr><td>Deterministic Optionality</td><td>&o</td><td>Same as '?', but match will not branch when it succeeds:
((&o thank you) very much)</td><td>8.3</td></tr>
<tr><td>Repetition (including null)</td><td>*</td><td>Matches a pattern an arbitrary number of times (including 0):
(?on (!days := (* <week-days>)))</td><td>2.1.1</td></tr>
</table>

Desired Operation	Symbol	Function	Section
Repetition (at least once)	+	Same as '*' but pattern must match at least once: `(?on (!days := (+ <week-days>)))`	2.3.1
Repetition (specified number)	↑	Matches a pattern a specified number of times: `(?on (!days := (↑ 3 <week-days>)))`	2.3.1

Unordered	&c	Matches each of its embedded patterns in any order:
		`(does compsci $n meet`
		`    (&c`
		`        (?on (!days := (+ <weekdays>)))`
		`        (?at (!time := <hrs-minutes>)`
		`             (!apm := (pm | am)))))`
			2.3.4

Variable Assignment	:=	Match pattern to right of assignment operator and store it in a variable whose name begins with '!': `(?on (!day := <week-days>))`	2.1.2

Variable Reference	=	Matches a value of a variable assigned earlier in a pattern:
		`(a (!var := (rose | hawk | stone))`
		`    is a (= !var) is a (= !var))`
			2.3.2

Variable Repetition	*var*	Records an arbitrary number of matches of different segments of the input to the same pattern:
		`((* (*var* := <week-days>)`
		`    ?%comma ?and))`
			4.1

Variable Coercion	&i	Maps a set of possible inputs into a single value which can be assigned to a variable:
		`((!pm := (&i pm`
		`        (afternoon | evening | night))))`
			4.2.1

Desired Operation	Symbol	Function	Section

Wildcard	$	Matches any atom in the input:
		`(a (!var := $)`
		`    is a (= !var) is a (= !var))`
			2.1.1

Numeric Wildcard	$n	Matches only numbers: `(where does compsci $n meet)`	2.3.6
Symbolic Wildcard	$w	Matches any word in the input except numbers: `(add $w $w to compsci $n)`	5.2
Dictionary Wildcard	$d	Matches any terminal element in a given grammar: `(is $d a terminal element in this grammar)`	5.2
Remainder (of input)	$r	Matches the rest of an input or nothing: `(does compsci $n meet in room $n $r)`	2.3.6

Up-to	&u	Matches input upto, but not including the first successful match of its embedded pattern:
		`(where (&u compsci) compsci $n`
		`     (&u (meet | meeting))`
		`     (meet | meeting) %qmark)`
			2.3.3

Up-to-and-including	&ui	Matches input up to and including the first successful match of its embedded pattern:
		`(where (&ui compsci) $n`
		`     (&ui (meet | meeting)) %qmark)`
			2.3.3

<table>
<tr><th>Desired Operation</th><th>Symbol</th><th>Function</th><th>Section</th></tr>
<tr><td>Scan
(does not move pointer)</td><td>&s</td><td>Scans input for its embedded pattern; matching will continue if pattern is present and fail if pattern is not present:

(&s <weekdays>)</td><td>2.3.5</td></tr>
<tr><td>Negation
(moves pointer)</td><td>~</td><td>Matches if embedded pattern does not match and moves pointer to next element in the input:

(does compsci $n meet
      (* ~($n | <week-days>)))</td><td>2.3.5</td></tr>
<tr><td>Negation
(does not move pointer)</td><td>&n</td><td>Same as '~', but does not move pointer:

(add (&n (↑ 2 <week-days>))
     $w $w (&u compsci) compsci $n $r)</td><td>2.3.5</td></tr>
<tr><td>Function Application</td><td>&apply</td><td>Applies function to variable list:

(!nopunct := (&i (&apply append !newvars)
                 (* (*var* := ~$p) ?$p)))</td><td>4.2.2</td></tr>
<tr><td>Function Application</td><td>&funcall</td><td>Applies function to variable list:

(!sum := (&i (&funcall plus !newvars)
             (* (*var* := $n))))</td><td>4.2.2</td></tr>
<tr><td>Morphology</td><td>&morph</td><td>Matches single root words with suffixes:

(&morph :root (duck|prince) :endings ling)</td><td>2.4</td></tr>
</table>

Appendix C
SAMPLE GRAMMAR

```
<be-present> -> (is | are | be | am)

<have-present> -> (have | has)
<have-past> -> (?<have-present> had)
<have-future> -> (will have ?had)
<have> -> (<have-present> | <have-past> | <have-future>)

<be-past> -> (was | were | <have-present> been | had been)
<be-future> -> (will be | will have been)
<be> -> (<be-present> | <be-past> | <be-future>)

<q-word> -> (what | who | where | when | how | why | how much |
             how many | how come)
<www> -> (what | who | which)

<poss> -> (%apost s)

<what-q> -> (<www> <be-pres> | <www> <poss>)

<pos-modal> -> (could | would | can )
<polite> -> (<pos-modal> you)

<me-us> -> (me | us)

<info-req1> -> (?<polite> <info-req2> ?<what-q>)
<info-req2> -> (tell <me-us> ?about | give <me-us> | print | type )
<info-req3> -> (<www> | ?<polite> <info-req2> ?<www>)
<info-req> -> (<what-q> | <info-req1>)

<a-an> -> (a | an)

<all> -> (all | everything | what)
<bulk> -> (bulk | majority | greater part)
<universal-quant> -> (?almost all | ?almost every ?one | each |
                      most | many | the <bulk> of)

<det> -> (the | <a-an> | <universal-quant>)

<punct> -> (%qmark | <dpunct>)
<dpunct> -> (%period | %emark)

(<info-req> ?<det> (!nam := $) ?<punct>)
=>
(ltm-ret !nam 'isa: nil nil)

(<info-req> ?<det> (!prop := $) of ?<det> (!nam := $) ?<punct>)
=>
(ltm-ret !nam !prop nil nil)

(<info-req3> (!prop := $) <be-pres> ?<det> (!nam := $) ?<punct>)
=>
(ltm-ret !nam !prop nil nil)

(<be-pres> ?<det> (!nam := $) ?<a-an> (!val := $) ?<punct>)
=>
(ltm-ret !nam 'isa: !val nil)

(<be-pres> ?<det> (!prop := $) of ?<det> (!nam := $) ?<det> (!val := $)
           ?<punct>)
=>
(ltm-ret !nam !prop !val nil)
```

```
<prp> -> (of | to | for | with)
<prp-about> -> (about | on)
<prp-in> -> (on | in | into | onto | inside | within)
<tof> -> (to | of)
<ofor> -> (of | for)

((<be-pres> ?<det> (!val := $) ?<det> (!prop := $) <tof> ?<det>
            (!nam := $) ?<punct>)
=>
(ltm-ret !nam !prop !val nil)

<known> -> (you <know-have> | ?is known | there is | stored
                            | in memory)
<know-have> -> (know | have)

((<info-req> <all> ?<that-do> ?<known> <prp-about> (!nam := $)
        ?<dpunct>)
=>
(ltm-ret-all !nam)

<label> -> (word | term | name | label)
<dlabel> -> (?the <label>)
<name> -> (?proper name | ?proper noun | token ?mode)

(?<dlabel> (!nam := $) <be-pres> <a-an> <name> ?<dpunct>)
=>
(ltm-store !nam 'token 'node-type: nil nil)

<same> -> (what | <same1>)
<same1> -> (?the same ?thing <as-that>)
<as-that> -> (as | that)
<that-do> -> (that | do)
<means-does> -> (means | does)

(?<dlabel> (!nam := $) <be-pres> <a-an> synonym <ofor> ?<dlabel>
            (!val := $) ?<dpunct>)
=>
(progn (ltm-store !nam !val 'synonym nil nil)
       (msg "Henceforth when you type "
            !nam " I'll interpret it "
            "as " !val t))

(?<det> (!nam := $) means ?<same> (!val := $) ?<means-does>
        ?<dpunct>)
=>
(ltm-store !nam !val 'synonym nil nil)

<typeof> -> (<type> ?of)
<type> -> (type | kind | form | instance | example)

(?<a-an> (!nam := $) <be-pres> <a-an> ?<typeof> (!val := $) ?<dpunct>)
=>
(ltm-store !nam !val 'isa: nil nil)

(?<det> (!val := $) <be-pres> ?<det> (!prop := $) of ?<det> (!nam := $)
        ?<dpunct>)
=>
(ltm-store !nam !val !prop nil nil)

(?<det> (!nam := $) <be-pres> (!vorp := $) ?<dpunct>)
=>
(ltm-spec !nam !vorp nil nil nil)

(?<det> (!nam := $) <be-pres> <a-an> (!vorp := $) (!val := $)
        ?<dpunct>)
=>
(progn (ltm-store !nam !val 'isa: nil nil)
       (ltm-spec !nam !vorp nil nil t))
```

```
<inverse-rel> -> (<inverse> ?<rel> ?name)
<inverse> -> (inverse | opposite | <backpointer>)
<backpointer> -> (back pointer | back %dash pointer | backpointer)
<rel> -> (relation | property | attribute | pointer | arc)

<forget> -> (remove | delete | erase | forget ?about | wipe out)
<load> -> (load | input | read ?in | dskin)
<store> -> (save | store | output | write ?out | dskout | print ?out)
<exit> -> (quit | exit | end ?$ session | ?good bye)
<command> -> (<forget> | <load> | <store> | <exit>)

<verb> -> (<command> | <be> | <info-req> | <have>)

(?<pos-modal> <forget> ?<det> (!prop := $) ?<prop> of (!nam := $)
                ?<punct>)
=>
(ltm-forget !nam !prop)

(?<polite> <load> ?the ?file (!fil := $) ?%period)
=>
(loadfile !fil)

(?<polite> <store> ?<prp-in> ?the ?file (!fil := $) ?<punct>)
=>
(storefile !fil)

((<exit> $r) => 'exit

<neg> -> (no | not | never | none | nothing | %apost t)
<pos> -> (yes | sure | indeed | certainly | certain | surely)

((&u <neg>) <neg> $r)
=>
(msg "I do not understand negations yet." (N 1))

((!s1 := (&u please)) please (!s2 := $r))
::>
(append !s1 !s2)

((!s1 := (&u <q-word>)) (!q := <q-word>) (!s2 := (&u <be>))
        (!v := <be>) ?(!p := <punct>))
::>
(nconc !s1 !q !v !s2 !p)

((!s1 := (&u <q-word>)) (!w1 := <q-word>) <poss> (!s2 := $r))
::>
(nconc !s1 !w1 (list 'is) !s2)

((!s1 := (&u $ <poss>)) (!w1 := $) <poss> (!w2 := $) (!s2 := $r))
::>
(nconc !s1 !w2 (list 'of) !w1 !s2)

(?<det> (!w1 := $) (!prp := <prp>) ?<det> (!w2 := $) (!v := <verb>)
        (!s2 := $r))
::>
(nconc !s2 !v !w1 !prp !w2)
```

Appendix D

BNF FOR DYPAR-I--Internal Form

CAPS means definition external to BNF
<angle-brackets> means BNF nonterminal
lower-case means terminal quoted node

```
<pattern> ::= nil
<pattern> ::= (<pattern1>)
<pattern1> ::= <term>
<pattern1> ::= <term> <pattern1>

<term> ::= ATOMIC-CONSTANT                  ; terminal (lexical) symbol
<term> ::= $                                ; one-place wildcard
<term> ::= $n                               ; one-place numeric wildcard
<term> ::= $w                               ; one-place word-only
                                            ;   wildcard
<term> ::= $d                               ; match any word in pattern
                                            ;   matcher's dictionary
<term> ::= $r                               ; match remainder of input
<term> ::= NON-TERMINAL                     ; non-term grammar symbol
<term> ::= FUNCTION                         ; hook for user-defined match
<term> ::= ( ↑ <n> <pattern1> )             ; matches <pattern1> <n> times
<term> ::= ( ? <pattern1> )                 ; optional match
<term> ::= ( &o <pattern1> )                ; deterministic optional match
<term> ::= ( * <pattern1> )                 ; kleene *, match <pattern1>
                                            ;   0 or more times
<term> ::= ( + <pattern1> )                 ; kleene +, match <pattern1>
                                            ;   1 or more times
<term> ::= ( ! <pattern-list> )             ; disjunction
<term> ::= ( !! <pattern-list> )            ; deterministic disjunction
<term> ::= ( &s <pattern1> )                ; scan input and continue
                                            ;   match if <pattern1>
                                            ;   present
<term> ::= ( &n <pattern1> )                ; fail match if <pattern1>
                                            ;   is next input seg
<term> ::= ( ~ <pattern1> )                 ; fail match if <pattern1>
                                            ;   is next input seg,
                                            ;   else consume next seg
<term> ::= ( &u <pattern1> )                ; match until <pattern1>
<term> ::= ( &ui <pattern1> )               ; match through <pattern1>
<term> ::= ( &c <pattern-list> )            ; unordered case match
<term> ::= ( := VARIABLE-NAME <pattern1>)   ; assign var to result of
                                            ;   match
<term> ::= ( &i VALUE <pattern1> )          ; return VALUE instead
                                            ;   of input seg if
                                            ;   <pattern1> matches
```

```
<term> ::= ( &apply LISP-FUNCTION <variable-list>)          ; applies function to var list
<term> ::= ( = VARIABLE-NAME )                              ; reference previous match var
<term> ::= ( &in <list> )                                   ; memb[input word, list]
<term> ::= ( &funcall LISP-FUNCTION <variable-list> )
<term> ::= ( &morph <morph-terms> )

<morph-terms> ::= <morph-term>
<morph-terms> ::= <morph-term> <morph-terms>
<morph-term> :: = <morph-key-word> <pattern>
<morph-key-word> ::= :root | :endings | :suffix

<n> ::= 1 | 2 | 3 | ...                                     ; any positive integer

<list> ::= ATOMIC-CONSTANT
<list> ::= ATOMIC-CONSTANT <list>

<pattern-list> ::= ( <pattern-list1> )
<pattern-list1> ::= <pattern>
<pattern-list1> ::= <pattern> <pattern-list1>

<variable-list> ::= VARIABLE-NAME
<variable-list> ::= VARIABLE-NAME <variable-list>
```

Definitions external to BNF

ATOMIC-CONSTANT ::= A LISP atom without a REWRITE: or FUNCTION: property and NOT eq to $.

NON-TERMINAL ::= A LISP identifier (i.e. a LITATOM) with a REWRITE: property whose value is a <pattern> defined above. Recursive pattern definitions are allowed. Pure left recursion, however, will be detected and will not expand in the matcher.

VARIABLE-NAME ::= A LISP identifier used in output assoc list of bindings as the name referencing the matched segment (or value returned by &i).

FUNCTION ::= A LISP identifier with a FUNCTION: property whose value is the name of a 1-argument user-defined FEXPR. This fexpr is used to match the input at the current location and must return a value compatible with the matcher.

VALUE ::= A LISP S-expression (or value returned by &apply)

LISP-FUNCTION ::= A LISP EXPR, MACRO, or LEXPR, which can be a built-in function or user-defined. This should return a value that can be meaningfully assigned to a variable.

Appendix E
ANSWERS TO EXERCISES

NOTE

The answers to questions are not intended to be exhaustive but rather to cover the main thrust of the question. It would be useful for the reader to try to supplement any of the answers.

Exercise 2-1

The following is a grammar file which represents a possible way to answer exercise 2-1.

;;; rewrite rules for non-terminals

;;; possible days

```
<days> -> (?on (!days := (* <disj-days>)))
<disj-days> -> (<mon> | <tues> | <wed> | <thurs> | <fri>)
<mon> -> (m | mon | mon %period | monday)
<tues> -> (tu | tues | tues %period | tuesday)
<wed> -> (w | wed | wed %period | wednesday)
<thurs> -> (th | thurs | thurs %period | thursday)
<fri> -> (f | fri | fri %period | friday)
```

;;; representing class names

```
<class> -> (<department> $n)
<department> -> (phil | compsci)
```

;;; question verbs

```
<isq> -> (does | is | will)
```

;;; verbs meaning or similar in meaning to "meet"

```
<meet> -> (meet | meets | meeting | take place | takes place | taking
           place | occur | occurs | occurring | held | taught | given)
```

;;; auxiliary verb

```
<auxv> -> (going to | going to be | scheduled | scheduled to
           | scheduled to be | be | being)
```

;;; exit words

```
<exit> -> (exit | bye | quit | end)
```

;;; top-level rule

;;; yes/no question regarding meeting times of classes

```
((<isq> (!class := <class>) ?<auxv> <meet> ?%qmark)
=>
(msg "You asked whether " !class " meets on " !days ".")
```

;;; exit rule

```
((<exit>) => 'exit
```

Exercise 2-2

The following is what a grammar file that answers exercise 2-2 might look like.

;;; rewrite rules for non-terminals

;;; words and phrases asking for meeting time (of a class)

```
<when> -> (when | ?(at | on) (what | which) <and-time-word>)
<and-time-word> -> (<time-word> ?and ?the ?<time-word>)
<time-word> -> (time | times | day | days | hour | hours)
```

;;; words and phrases asking for meeting place (of a class)

```
<where> -> (where | ?(at | in) (what | which) <and-place-word>)
<and-place-word> -> (<place-word> ?and ?the ?<place-word>)
<place-word> -> (place | room | building | hall)
```

;;; representing class names

```
<class> -> (<department> $n)
<department> -> (phil | compsci)
```

;;; verbs meaning or similar in meaning to "meet"

```
<meet> -> (meet | meets | meeting | take place | takes place | taking
           place | occur | occurs | occurring | held | taught | given)
```

;;; auxiliary verbs

```
<auxv1> -> (does | is | will)
<auxv2> -> (going to ?be | scheduled ?to ?be | be | being)
```

;;; exit words

```
<exit> -> (exit | bye | quit | end)
```

;;; top-level rules

;;; when does <class> meet?

```
(<when> ?<auxv1> (!class := <class>) ?<auxv2> ?<meet> ?%qmark)
=>
(msg "You asked when " !class " meets?")
```

;;; where does <class> meet?

```
(<where> ?<auxv1> (!class := <class>) ?<auxv2> ?<meet> ?%qmark)
=>
(msg "You asked where " !class " meets?")
```

;;; exit rule

```
(<exit>) => 'exit
```

Exercise 2-3

Here is a grammar which is one possible answer to exercise 2-3. Note, especially, the use of '~' in <name> and of '(&n (&u...)) in the first top level rule.

;;; rewrite rules for non-terminals

;;; representing class names

```
<class-var -> (!class := <class>)
<class> -> (<department> $n)
<department> -> (phil | compsci)
```

;;; relative pronoun

```
<who-that> -> (who | that)
```

;;; question verbs

```
<has-is-q> -> (has | does | is | will)
```

;;; verbs meaning or similar in meaning to "registered for"

```
<registered-for> -> (registered (in | for) | taking | take |
                     enrolled (in | for) | in)
```

;;; auxiliary verbs

```
<auxv1> -> (is | will ?be)
<auxv2> -> (going to ?be | scheduled ?to ?be | be)
```

;;; name

```
<name-var> -> (!name := <name>)
<name> -> (~$n ?~$n ?~$n)
```

;;; pronouns and pronoun-like phrases meaning any/some one

```
<any-some-one> -> (anyone | someone | anybody | somebody |
                   (any | some) (student | person))
```

;;; exit words

```
<exit> -> (exit | bye | quit | end)
```

;;; top-level rules

;;; rule covering whether a particular student in a given class

```
((&n (&u <any-some-one>)) <has-is-q> <name-var> ?<auxv>
 <registered-for> <class-var> ?%qmark)
=>
(msg "you asked whether " !name " is registered for " !class)
```

;;; rule covering whether there any students in a given class

```
(<has-is-q> ?there <any-some-one> ?<who-that> ?<auxv1> ?<auxv2>
 <registered-for> <class-var> ?%qmark)
=>
(msg "you asked whether anyone is registered for " !class)
```

;;; exit rule

```
(<exit>) => 'exit
```

Exercise 2-4

The way around the problem is to use a transformation rule to skip over the inessential words and phrases at the beginning of the given inputs. The following is a grammar which covers the requirements of the exercise 2-4 but not much more.

;;; rewrite rules for non-terminals

;;; representing class names

```
<class> -> (<department> $n)
<department> -> (phil | compsci)
```

;;; possible days

```
<days> -> (?on (!days := (* <disj-days>)))
<disj-days> -> (<mon> | <tues> | <wed> | <thurs> | <fri>)
<mon> -> (m | mon | mon %period | monday)
<tues> -> (tu | tues | tues %period | tuesday)
<wed> -> (w | wed | wed %period | wednesday)
<thurs> -> (th | thurs | thurs %period | thursday)
<fri> -> (f | fri | fri %period | friday)
```

;;; possible hours

```
<hour> -> (?at (!hr := $n) ?%colon (!min := ?$n) ?<o-clock>
           (!am-pm := ?(<am> | <pm>)))

<am> -> (am | in the morning)
<pm> -> (pm | in the (evening | afternoon) | at night)
<o-clock> -> (o %apost clock)
```

;;; verbs meaning or similar in meaning to "meet"

```
<meet> -> (meet | meets | meeting | take place | takes place
                    | taking place | occur | occurs | occurring
                    | held | taught | given)
```

;;; question verbs

```
<isq> -> (does | is | will | whether | if)
```

;;; auxiliary verb

```
<auxv1> -> (does | is | will)
<auxv2> -> (going to ?be | scheduled ?to ?be | be | being)
```

;;; exit words

```
<exit> -> (exit | bye | quit | end)
```

```
;;; transformation rule to skip over inessential words and phrases

;;; the + operator assures there is input to be skipped over before the
;;; essential input; the essential input is saved in !first-element and
;;; !r which are passed to the parser for further processing

    ((+ ~$n (!first-element := <isq>) (!r := $r))
    ::>
    (nconc !first-element !r)
```

;;; top-level rules

;;; queries

;;; yes/no question regarding meeting times of classes

```
(<isq> (!class-nam := <class>) ?<auxv1> ?<aux2> <meet> (&c (<hour>)
(<days>)) ?%qmark)
=>
(msg "You asked if " !class-nam " meets at " !hr !min !am-pm !days
 ".")
```

Exercise 4-1

Here is a likely solution to exercise 4-1.

Two new nonterminals must be defined to handle possible days:

```
<st-disj-days> -> (* (*var* := <disj-days>) ?<conj-syntax>)
<conj-syntax> -> (%comma | and | %comma and).
```

Then <days> has to be modified to:

```
<days> -> (?on <st-disj-days>).
```

Finally every top-level rule containing !days on the RHS must be changed so that !days is substituted for by (access !newvars) on the RHS.

Further, additional patterns and modifications must be introduced for '<am>' and '<pm>':

```
<am> -> (am | (&i am <am-pat>))
<am-pat> -> (in tho morning)
<pm> -> (pm | (&i pm <pm-pat>))
<pm-pat> -> (noon | in the (afternoon | evening) | at night)
```

Alternatively <st-disj-days> could also be defined as:

```
<st-disj-days> -> ((*var* := <disj-days>) ?<conj-syntax>
                   ?<st-disj-days>)
```

Exercise 4-2

This is an answer to exercise 4-2. First, a new non-terminal must be added to the file given as answer to exercise 2-4 under ";;; possible hours."

```
<twenty-four-hour-time> -> (!time := (&i (&funcall rdtime
                           (!hr !min !am-pm)) <hour>))
```

Then, the non-terminal <hour> must be expanded to look like the following:

```
<hour> -> (?at (!hr := $n) ?%colon (!min := ?$n) ?<o-clock>
          (!am-pm := ?(am | pm | (&i am <am-pat>) |
          (&i pm <pm-pat>))))
```

The non-terminals <am> and <pm> are no longer necessary in support of <hour>, but <am-pat> and <pm-pat> still are.

<twenty-four-hour-time> is substituted for <hour> in the yes/no question regarding meeting times, and !time is substituted for !hr !min !ap-pm in its RHS. The new top level rule would look like the following:

```
(isq> (!class-nam := <class>) ?<auxv1> ?<auxv2> <meet> (&c
(<twenty-four-hour-time>) (<days>)) ?%qmark)
=>
(msg "you asked whether " !class-nam " meets at " !time
 (access !newvars) ".")
```

Exercise 5-1

This is the answer to Exercise5-1. There is very little room for variance in writing an answer to this exercise, so if your answer works it should look a lot like this one. This answer consists entirely of LISP code which should be put into a file of its own and loaded separately from the grammar file.

The exercise is to write the operator '$q'. First, you need to write a function which takes the list of the unconsumed input string as its argument. (Don't worry about how you will get the list--DYPAR will plug it in for you when it calls your function.) You need to make your function a predicate on the car of this list:

```
(defun $qcheck (l)
       (equal (car (explode (car l))) 'q))
```

This function returns a "t" if the first letter of the first word of the input string is equal to "q". Now you must attach the function $qcheck to its DYPAR symbol, '$q'. You do this using defprop to assign '$q' a ':function' property whose value is '$qcheck':

```
(defprop $q $qcheck :function)
```

Finally, you must give $q a ':symbol' property for use by the cross-referencer. Since $q is designed to be used on non-numeric input, the value of the ':symbol' property is set to '$':

```
(defprop $q. $ :symbol)
```

Exercise 5-2

This is the answer for Exercise 5-2. Your primary LISP function may vary from the one given here, but the auxiliary defprop functions should be identical to the ones below (except that you may have given the primary function a name other than '$special-punct'.)

Write the primary function:

```
(defun $special-punct (l)
        (member (car l) !!somepunct))
```

!!somepunct is a global list, set by using the FRANZ LISP function dv, which is equivalent to a setq. It contains the special DYPAR symbol equivalents of '.' '!' and '?' This is because all user punctuation input is converted to these symbols when it is read, so in order to match the user's input, the function must employ the DYPAR symbols:

```
(dv !!somepunct (%period %emark %qmark))
```

Then add the auxiliary functions using defprop:

```
(defprop $pe $special-punct :function)

(defprop $pe $ :symbol)
```

Appendix F
HISTORY & CREDITS

The first version of DYPAR was designed and implemented by Jaime Carbonell in the UCI dialect of LISP on a TOPS10 operating system in the summer of 1980. A later more sophisticated version was developed by Carbonell in the spring of 1981. This version was translated to FRANZ LISP running under UNIX by Mark Boggs, who added the cross referencer, and who is essentially responsible for making sure the current implementation is kept in good working order. To this point most of the design considerations have been the province of Carbonell (with occasional suggestions from Boggs), and the implementation details have been the province of Boggs (with significant contributions from Carbonell, including the original pattern matcher). Steven Romig did some work on the oldest version of FRANZ LISP DYPAR which has carried over to the present version. Boggs and Robert Frederking worked on a DYPAR application grammar for a factory scheduling system off and on during the last half of 1981. This grammar served as a testbed for DYPAR development throughout the following year. It recognized on the order of a quarter of a million inputs. A spelling corrector written by Frederking, and modified by Boggs and Steven Minton was added during the summer of 1982.

During the summer of 1982 Carbonell developed yet another version of DYPAR (MCDYPAR) which contained some new operators. Those operators ('&u', '&c', and '&m') were converted to be non-deterministic and added to DYPAR by Steve Morrisson and Boggs. During the summer of 1983 at the instigation of the new members of the project (Ira Monarch, Scott Safier, and Jim Washburn), Boggs added some new operators ('~', '$r', '&n'). Much of the DYPAR source code was rewritten during 1983 by Carbonell and Boggs so as to be both more efficient and more robust. Morrisson added the'&morph' operator.

In the spring of 1984, Safier converted DYPAR to COMMON LISP and added an incremental cross referencer. Also at that time, Nic Easton (on loan from DEC) added the '&s' operator and Marion Kee added the '&ui' operator. During the summer of 1984 Demetri Silas worked with Boggs and Safier on a better incremental cross-referencer, which was added to DYPAR in the fall of 1984.

This manual was originally written by Carbonell and Boggs; Ira Monarch added the exercises for Chapters 2, 3, and 4; the exercises for Chapter 5 were written by Marion Kee. Beth Byers worked with Boggs on the tables and on generally cleaning up the manuscript. Kee re-wrote Chapter 5 and Chapter 2, added material in the other chapters, and improved the index. Steve Morrisson helped to write the section on the operator '&morph' in chapter two. In October 1984, the manual underwent a major review by group members and others involved with the project. Boggs, Carbonell, Monica Cellio, Nic Easton, Kee, Monarch and Morrisson all made valuable suggestions leading to improvements in the manual.

Chapter 7 was added in the summer of 1985 by Boggs and Kee. During the fall of that year, some revisions were made to the other chapters to keep abreast of developments in DYPAR. Kee and Boggs put this manual into final CMU technical report format in December, 1985.

We would like to thank the Carnegie-Mellon Computer Science community for their support and aid during the development of DYPAR, most specifically Don Kosy and Phil Hayes, for their enlightening insights into some of the problems of natural language parsing, and Mark Fox and Gary Strohm of the Intelligent Systems Lab, for giving us our first real domain.

Index

DYPAR symbol	Character	Ascii Value
%colon	:	58
%dash	-	45[1]
%slash	/	47
%apost	’	39
%hash	#	35
%comma	,	44
%lparen	(	40
%rparen	)	41
%star	*	42
%bquote	‘	96
%rsbrack	[	91
%lsbrack	]	93
%bslash	\	92
%vbar	|	124
%semicolon	;	59
%dquote	"	34
%lcbrack	{	123
%rcbrack	}	125
%labrack	<	60
%rabrack	>	62
%amper	&	38
%percent	%	37
%dollar	$	36
%plus	+	43
%equal	=	61
%underbar	_	95
%upcaret	↑	94
%atsign	@	64
%tilde	~	126
%emark	!	33
%qmark	?	63
%cr	↑M	13[2]
%lf	↑J	10[3]
%period	.	46

[1] This conversion is commonly disabled.

[2] [illegible] "carriage return."

Appendix E
ANSWERS TO EXERCISES

NOTE

The answers to questions are not intended to be exhaustive but rather to cover the main thrust of the question. It would be useful for the reader to try to supplement any of the answers.

Exercise 2-1

The following is a grammar file which represents a possible way to answer exercise 2-1.

igual que gensym — crean las variables var₁ var₂ var₃ . . .

todas las variables se guardan en un cajón para poder recuperarla posteriormente

∃ *vars*

```
;;; rewrite rules for non-terminals

;;; possible days

     <days> -> (?on (!days :• (• .<disj-days>)))
     <disj-days> -> (<mon> | <tues> | <wed> | <thurs> | <fri>)
     <mon> -> (m | mon | mon %period | monday)
     <tues> -> (tu | tues | tues %period | tuesday)
     <wed> -> (w | wed | wed %period | wednesday)
     <thurs> -> (th | thurs | thurs %period | thursday)
     <fri> -> (f | fri | fri %period | friday)

;;; representing class names

     <class> -> (<department> $n)
     <department> -> (phil | compsci)

;;; question verbs

     <isq> -> (does | is | will)

;;; verbs meaning or similar in meaning to "meet"

     <meet> -> (meet | meets | meeting | take place | takes place | taking
                place | occur | occurs | occurring | held | taught | given)

;;; auxiliary verb

     <auxv> -> (going to | going to be | scheduled | scheduled to
                | scheduled to be | be | being)

;;; exit words

     <exit> -> (exit | bye | quit | end)

;;; top-level rule

;;; yes/no question regarding meeting times of classes

     ((<isq> (!class :• <class>) ?<auxv> <meet> ?%qmark)
     =>
     (msg "You asked whether " !class " meets on " !days ".")

;;; exit rule

     ((<exit>) => 'exit
```

(*days* := <days>)

<days>

(access !newvar)

cachear el valor de todas las variables

Exercise 2-2

The following is what a grammar file that answers exercise 2-2 might look like.

;;; rewrite rules for non-terminals

;;; words and phrases asking for meeting time (of a class)

```
<when> -> (when | ?(at | on) (what | which) <and-time-word>)
<and-time-word> -> (<time-word> ?and ?the ?<time-word>)
<time-word> -> (time | times | day | days | hour | hours)
```

;;; words and phrases asking for meeting place (of a class)

```
<where> -> (where | ?(at | in) (what | which) <and-place-word>)
<and-place-word> -> (<place-word> ?and ?the ?<place-word>)
<place-word> -> (place | room | building | hall)
```

;;; representing class names

```
<class> -> (<department> $n)
<department> -> (phil | compsci)
```

;;; verbs meaning or similar in meaning to "meet"

```
<meet> -> (meet | meets | meeting | take place | takes place | taking
           place | occur | occurs | occurring | held | taught | given)
```

;;; auxiliary verbs

```
<auxv1> -> (does | is | will)
<auxv2> -> (going to ?be | scheduled ?to ?be | be | being)
```

;;; exit words

```
<exit> -> (exit | bye | quit | end)
```

;;; top-level rules

;;; when does <class> meet?

```
(<when> ?<auxv1> (!class := <class>) ?<auxv2> ?<meet> ?%qmark)
=>
(msg "You asked when " !class " meets?")
```

;;; where does <class> meet?

```
(<where> ?<auxv1> (!class := <class>) ?<auxv2> ?<meet> ?%qmark)
=>
(msg "You asked where " !class " meets?")
```

;;; exit rule

```
(<exit>) => 'exit
```

Exercise 2-3

Here is a grammar which is one possible answer to exercise 2-3. Note, especially, the use of '~' in <name> and of '(&n (&u...)) in the first top level rule.

;;; rewrite rules for non-terminals

;;; representing class names

```
<class-var -> (!class := <class>)
<class> -> (<department> $n)
<department> -> (phil | compsci)
```

;;; relative pronoun

```
<who-that> -> (who | that)
```

;;; question verbs

```
<has-is-q> -> (has | does | is | will)
```

;;; verbs meaning or similar in meaning to "registered for"

```
<registered-for> -> (registered (in | for) | taking | take |
                     enrolled (in | for) | in)
```

;;; auxiliary verbs

```
<auxv1> -> (is | will ?be)
<auxv2> -> (going to ?be | scheduled ?to ?be | be)
```

;;; name

```
<name-var> -> (!name := <name>)
<name> -> (~$n ?~$n ?~$n)
```

;;; pronouns and pronoun-like phrases meaning any/some one

```
<any-some-one> -> (anyone | someone | anybody | somebody |
                   (any | some) (student | person))
```

;;; exit words

```
<exit> -> (exit | bye | quit | end)
```

;;; top-level rules

;;; rule covering whether a particular student in a given class

```
((&n (&u <any-some-one>)) <has-is-q> <name-var> ?<auxv>
 <registered-for> <class-var> ?%qmark)
=>
(msg "you asked whether " !name " is registered for " !class)
```

;;; rule covering whether there any students in a given class

```
(<has-is-q> ?there <any-some-one> ?<who-that> ?<auxv1> ?<auxv2>
 <registered-for> <class-var> ?%qmark)
=>
(msg "you asked whether anyone is registered for " !class)
```

;;; exit rule

```
((<exit>) => 'exit
```

Exercise 2-4

The way around the problem is to use a transformation rule to skip over the inessential words and phrases at the beginning of the given inputs. The following is a grammar which covers the requirements of the exercise 2-4 but not much more.

;;; rewrite rules for non-terminals

;;; representing class names

```
<class> -> (<department> $n)
<department> -> (phil | compsci)
```

;;; possible days

Durán

(:= !days (* <disj-days>))

```
<days> -> (?on (!days := (* <disj-days>)))
<disj-days> -> (<mon> | <tues> | <wed> | <thurs> | <fri>)
<mon> -> (m | mon | mon %period | monday)
<tues> -> (tu | tues | tues %period | tuesday)
<wed> -> (w | wed | wed %period | wednesday)
<thurs> -> (th | thurs | thurs %period | thursday)
<fri> -> (f | fri | fri %period | friday)
```

;;; possible hours

```
<hour> -> (?at (!hr := $n) ?%colon (!min := ?$n) ?<o-clock>
           (!am-pm := ?(<am> | <pm>)))

<am> -> (am | in the morning)
<pm> -> (pm | in the (evening | afternoon) | at night)
<o-clock> -> (o %apost clock)
```

;;; verbs meaning or similar in meaning to "meet"

```
<meet> -> (meet | meets | meeting | take place | takes place
                | taking place | occur | occurs | occurring
                | held | taught | given)
```

;;; question verbs

```
<isq> -> (does | is | will | whether | if)
```

;;; auxiliary verb

```
<auxv1> -> (does | is | will)
<auxv2> -> (going to ?be | scheduled ?to ?be | be | being)
```

;;; exit words

```
<exit> -> (exit | bye | quit | end)
```

```
;;; transformation rule to skip over inessential words and phrases

;;; the + operator assures there is input to be skipped over before the
;;; essential input; the essential input is saved in !first-element and
;;; !r which are passed to the parser for further processing

    ((+ ~$n (!first-element := <isq>) (!r := $r))
    ::>
    (nconc !first-element !r)
```

equipara los patrones en cualquier orden

((-) => ( )

;;; top-level rules

;;; queries

;;; yes/no question regarding meeting times of classes

```
((<isq> (!class-nam := <class>) ?<auxv1> ?<aux2> <meet> (&c (<hour>)
((<days>)) ?%qmark)
=>
(msg "You asked if " !class-nam " meets at " !hr !min !am-pm !days
 ".")
```

Exercise 4-1

Here is a likely solution to exercise 4-1.

Two new nonterminals must be defined to handle possible days:

```
<st-disj-days> -> (* (*var* := <disj-days>) ?<conj-syntax>)
<conj-syntax> -> (%comma | and | %comma and).
```

Then <days> has to be modified to:

```
<days> -> (?on <st-disj-days>).
```

Finally every top-level rule containing !days on the RHS must be changed so that !days is substituted for by (access !newvars) on the RHS.

Further, additional patterns and modifications must be introduced for '<am>' and '<pm>':

```
<am> -> (am | (&i am <am-pat>))
<am-pat> -> (in the morning)
<pm> -> (pm | (&i pm <pm-pat>))
<pm-pat> -> (noon | in the (afternoon | evening) | at night)
```

Alternatively <st-disj-days> could also be defined as:

```
<st-disj-days> -> ((*var* := <disj-days>) ?<conj-syntax>
                    ?<st-disj-days>)
```

Exercise 4-2

This is an answer to exercise 4-2. First, a new non-terminal must be added to the file given as answer to exercise 2-4 under ";;; possible hours."

```
<twenty-four-hour-time> -> (!time := (&i (&funcall rdtime
                            (!hr !min !am-pm)) <hour>))
```

Then, the non-terminal <hour> must be expanded to look like the following:

```
<hour> -> (?at (!hr := $n) ?%colon (!min := ?$n) ?<o-clock>
           (!am-pm := ?(am | pm | (&i am <am-pat>) |
           (&i pm <pm-pat>))))
```

The non-terminals <am> and <pm> are no longer necessary in support of <hour>, but <am-pat> and <pm-pat> still are.

<twenty-four-hour-time> is substituted for <hour> in the yes/no question regarding meeting times, and !time is substituted for !hr !min !ap-pm in its RHS. The new top level rule would look like the following:

```
(isq> (!class-nam := <class>) ?<auxv1> ?<auxv2> <meet> (&c
(<twenty-four-hour-time>) (<days>)) ?%qmark)
=>
(msg "you asked whether " !class-nam " meets at " !time
 (access !newvars) ".")
```

Exercise 5-1

This is the answer to Exercise5-1. There is very little room for variance in writing an answer to this exercise, so if your answer works it should look a lot like this one. This answer consists entirely of LISP code which should be put into a file of its own and loaded separately from the grammar file.

The exercise is to write the operator 'Sq'. First, you need to write a function which takes the list of the unconsumed input string as its argument. (Don't worry about how you will get the list--DYPAR will plug it in for you when it calls your function.) You need to make your function a predicate on the car of this list:

```
(defun Sqcheck (l)
        (equal (car (explode (car l))) 'q))
```

This function returns a "t" if the first letter of the first word of the input string is equal to "q". Now you must attach the function Sqcheck to its DYPAR symbol, 'Sq'. You do this using defprop to assign 'Sq' a ':function' property whose value is 'Sqcheck':

```
(defprop $q $qcheck :function)
```

Finally, you must give Sq a ':symbol' property for use by the cross-referencer. Since Sq is designed to be used on non-numeric input, the value of the ':symbol' property is set to 'S':

```
(defprop $q $ :symbol)
```

Exercise 5-2

This is the answer for Exercise 5-2. Your primary LISP function may vary from the one given here, but the auxiliary defprop functions should be identical to the ones below (except that you may have given the primary function a name other than 'Sspecial-punct'.)

Write the primary function:

```
(defun $special-punct (l)
        (member (car l) !!somepunct))
```

!!somepunct is a global list, set by using the FRANZ LISP function dv, which is equivalent to a setq. It contains the special DYPAR symbol equivalents of '.' '!' and '?' This is because all user punctuation input is converted to these symbols when it is read, so in order to match the user's input, the function must employ the DYPAR symbols:

```
(dv !!somepunct (%period %emark %qmark))
```

Then add the auxiliary functions using defprop:

```
(defprop $pe $special-punct :function)

(defprop $pe $ :symbol)
```