3. Labels

How Sparser analyses a text is determined by a grammar of user-defined phrase structure rules (sometimes known as "rewrite rules" or "productions"). These rules are based on patterns of adjacent constituents, where the patterns are defined by the constituents' *labels*—the terminal and non-terminal vocabulary of the grammar.

Sparser supports a larger set of label types than many parsing systems do, and its technical definitions of otherwise standard label types are likely to be richer than most users may be accustomed to. The purpose of this section is to provide the information necessary to understand the types of labels Sparser supports, their properties, and how they are defined and managed. The next section (§4) describes how they are used to define phrase structure rules, and a later section (§6) describes how they are introduced in the course of an analysis.

3.1 Words

The notion of a 'word' is given a technical definition in Sparser. Every token returned by the tokenizer—without exception—is represented as a structured object (defstruct) of type `word`. Since the tokenizing algorithm (§6.3) is conservative and does not filter out any character sequences, this means that besides being used for words in the ordinary sense, the "word" data structure is used for digit sequences, punctuation characters, and sequences of spaces.

The tokenizer recognizes as separate tokens all contiguous sequences of alphabetic characters, sequences of digits, sequences of spaces, individual punctuation characters, tab, newline, and the non-printing ascii characters. The string "$47.2 million", for example, is seen as six tokens: the punctuation "$", the digits "42", the punctuation ".", a single space, and the word "million". The assembly of the full number and its appreciation as an amount of money left to chart-level processing.

Every code in the ascii character sequence is appreciated by the tokenizer. Those not defined as alphabetic, digit, punctuation, or specially designated for internal use by Sparser (see §3.1.5 below) are designated as "meaningless". The presence of a meaningless character in the input can initiate a Lisp "break" as described below. The interpretation of nearly all ascii characters is table-driven and can be changed by the user.

3.1.1 Case-sensitivity

Words are individuated by the sequence the characters that comprise them. Every instance of a given character sequence delimited by the tokenizer will be represented by the identical word object, with the exception that the case of alphabetic tokens is treated specially.

By default, the case of the characters in an alphabetic token is ignored for purposes of identifying the word object to which the token corresponds. For example, unless explicitly defined to be otherwise, the tokens "the", "The", "THE", "tHe", etc. will all be

represented by the same word object. This object—what we could call the "canonical version" of the word—is given in lowercase. Every word is returned by the tokenizer only in its lowercase canonical version, with matches to any other versions one later as needed (see §6.4).. *This means that if the case of a word does not matter to you, you should define the word in lowercase.*

The grammar writer has the option, however, to explicitly define a case-sensitive version of a word (e.g. to distinguish "President" from "president" since these variants pattern differently and that may be significant in a given grammar). The rule is that if a fully lowercase spelling is used when a word is defined, then it will be understood as case insensitive. However, if any of a word's characters are given in uppercase in the definition, then that specific pattern of upper and lowercase letters will be given its own word object and checked for in the application of phrase structure rules and hooks; it will also be cross-linked with the word object for its case-insensitive variant as described below in the presentation of the fields of the word object.

Case is always noticed by the tokenizer and recorded in the positions of the chart. This is described at length in §6.4.

3.1.2 Defining words

A number of words are predefined by Sparser (see below) including words to indicate the beginning and end of the character stream being analyzed and all of the punctuation characters defined by Common Lisp's standard interpretation of the ascii character set.

Words are defined automatically when they are mentioned in the "def-cfr" form for defining phrase structure rules (§4.1.4). This is usually the easiest way to construct the terminal vocabulary of one's grammar. Check the spelling of the strings in rules carefully since there is no check that a word in a rule is already defined and any misspellings will go through unnoticed.

If a word is useful independently of phrase structure rules, then the following form can be used. This same routine is used internally to define words automatically when they are mentioned in a rule.

`define-word (string)` `function`

> Returns a word object. If the string is already defined as a word, then the same object will be returned. The string must consist of one or more characters of a single type, i.e. alphabetic or digit. If the type is punctuation the string may be only one character long. (Spaces are an exception; see §3.1.5.4) Capitalization in alphabetic strings is treated specially, see §3.1.1. The string should be given in all lowercase if capitalization does not matter.

As a side effect of each definition, a internal symbol is created in the `word` package and bound to the newly defined word object. This symbol is proclaimed 'special'. Its pname is the same as the input string. Thus is one defines the word "the", it can be accessed via the symbol `word::|the|`. Note the use of vertical bars to designate a symbol with a mixed or lowercase print name.

Part of the process of defining a word is the calculation of its capitalization pattern and some simple morphological processing to note whether it ends in a grammatically significant suffix like “ed”. The results of this calculation are stored in the word’s fields as indicated below.

If the word is not lowercase, then it will be listed as one of the ‘capitalization-variants’ of its lowercase equivalent. This allows the rule-instantiating mechanism (§6.6) to make the connection between the canonical lowercase version of the word (the version that the tokenizer will enter into the chart) and any non-lowercase version which should be considered instead depending on the capitalization pattern that occurs in a given instance. If no lowercase equivalent word object exists at the time that a capitalized word is defined then it will be created.

Since some ascii characters either do not normally print or can be awkward to refer to as a string (e.g. tab), a special definition form is provided for them that assigns the word to symbol which can then be used to refer to it in rules. The symbols already defined for punctuation are given shortly. You can assign additional, alternative names (symbols) to them by using this definition form:

`define-punctuation (symbol character)` `macro`

> Returns three values, the word object defined for the character by this action (or the object previously defined for it), the symbol in the word package for the character taken naturally as a single-character string, and a symbol in the word package with the same pname as the one passed in as an argument. Both symbols are bound to the word object. Both are proclaimed ‘special’. Characters can be given in Common Lisp notation, e.g `#\!`.

Words created using define-punctuation have the keyword `:punctuation` on their property lists, as well as the keyword `:use-symbol-when-printing`. This second keyword, as its name suggests, is noticed by the built-in routines that print word objects.

3.1.3 Accessing words

The word corresponding to a string can be accessed with the following function:

`word-named (string)` `function`

> Returns the word object whose pname is identical (`equal`) to the string given as its argument. If no such word is defined then `nil` is returned.

This access function uses the same mechanism as the tokenizer does, so its results are guaranteed to match those that will obtain during the course of a parse (but see the discussion of ‘unknown’ words below). This mechanism uses the Lisp symbol table, using the symbol created for the word when it was defined as the basis of the lookup.

Since not all ascii characters have a representation as a string that one can type into Lisp, this alternative access routine is provided.

`punctuation-named (character)` `function`

Returns the word object whose pname is a string consisting of the single character given as input. If no such word is defined then `nil` is returned.

A list of all the words that have been defined is kept in this global. The words are listed from the most recently defined to the earliest.

`*words-defined*` `defparameter`

3.1.4 The components of the "word" object

Words are implemented as defstructs of type `word`. Their components can be accessed by the following functions, which all take word objects as their single argument, and correspond to a field in the structure in the usual fashion. As a Lisp object a word prints in sharp sign notation as, e.g. `#<word "the">`, where the string is as given in the object's pname field. Words based on on-printing strings print using their symbol, e.g. `#<word SOURCE-START>`.

`word-pname (word)` `access-function`

Returns a string. This capitalization-sensitive string establishes the identify of this word to the tokenizer and the definition routine.

`word-rules (word)` `access-function`

Returns one of these values: either a 'rule-set' object, the keyword symbol `:whitespace` (§3.1.6), or else `nil` when the word has no rules associated with it. See §4.6 for a description of rule sets.

`word-morphology (word)` `access-function`

Returns a keyword or `nil`. The keyword describes any affixes the word has. The current possibilities are `:ends-in-ed`, `:ends-in-ing`, `:ends-in-s`, and `:space`.

`word-capitalization (word)` `access-function`

Returns a keyword or `nil`. The keyword describes the pattern of upper and lower case characters in the word. The current possibilities are

```
:lower-case
:mixed-case
:all-caps
:initial-letter-capitalized
:single-capitalized-letter
:digits
```

`word-capitalization-variants` `access-function`

This field is populated only for a lowercase word where one or more of its capitalized variant forms has also been defined. This access function returns a list all such variant words that have been defined. When the word is itself a capitalized variant this field contains `nil`.

`word-plist (word)` `access-function`

Returns a list of paired objects that is intended to be employed as a "property list" in the usual sense.

`word-symbol (word)` `access-function`

Returns the symbol that was constructed and bound to this word at the time it was defined. It is in the `word` package. The symbol-name of the symbol is the same string as the pname of the word, which means that most references to this symbol will have to be enclosed in vertical bars so that Lisp references the intended case sensitive string, e.g. `word::|the|`.

3.1.5 Predefined words

Fifty four words are predefined in every version of Sparser. Thirty six punctuation characters, two words for reserved control characters, and 16 words consisting of just spaces. These are all described below, along with the notion of a 'meaningless character'.

3.1.5.1 Punctuation

Sparser predefines words for all the ascii punctuation characters and for certain ascii control characters that it uses for markers in the course of its parsing operations. (Predefining punctuation make the operation of the tokenizer more efficient.) The list below is the actual calls to define-punctuation used for their definition. Note that the symbols can be used to name punctuation characters in rules and will be used in printing references to them, e.g. `#<word tab>`. The ascii code corresponding to each punctuation character is listed after the semicolons with each definition. The characters themselves are named according to the Common Lisp conventions using "`#\`". Numbers are base 10.

```
(define-punctuation tab  #\tab)             ;; ascii 9
(define-punctuation linefeed #\Linefeed)    ;; 10
(define-punctuation new-page #\Page)        ;; 12

(define-punctuation newline  #\newline)     ;; 13
```

Note that "newline" is a pseudo character that Common Lisp supports to transparently represent whatever character the operating system uses to indicating new line, i.e. the combination of a 'return' action that moves the cursor to the beginning of the line and a 'linefeed' action that moves the cursor to the next line. On older versions of the Macintosh newline corresponds to the character `#\return` (13), while in Unix it is the character `#\linefeed` (12).

```
(define-punctuation  one-space       #\space )   ;; 32

(define-punctuation  exclamation-point   #\! )   ;; 33
(define-punctuation  double-quote        #\" )   ;; 34
(define-punctuation  sharp-sign          #\# )   ;; 35
(define-punctuation  dollar-sign         #\$ )   ;; 36
(define-punctuation  percent-sign        #\% )   ;; 37
(define-punctuation  and-sign            #\& )   ;; 38
(define-punctuation  single-quote        #\' )   ;; 39
(define-punctuation  open-paren          #\( )   ;; 40
```

```
(define-punctuation  close-paren         #\) )   ;; 41
(define-punctuation  asterisk            #\* )   ;; 42
(define-punctuation  plus-sign           #\+ )   ;; 43
(define-punctuation  comma               #\, )   ;; 44
(define-punctuation  hyphen              #\- )   ;; 45
(define-punctuation  period              #\. )   ;; 46
(define-punctuation  forward-slash       #\/ )   ;; 47

(define-punctuation  colon               #\: )   ;; 58
(define-punctuation  semi-colon          #\; )   ;; 59
(define-punctuation  open-angle-bracket  #\< )   ;; 60
(define-punctuation  equal-sign          #\= )   ;; 61
(define-punctuation  close-angle-bracket #\> )   ;; 62
(define-punctuation  question-mark       #\? )   ;; 63
(define-punctuation  ampersand           #\@ )   ;; 64

(define-punctuation  open-square-bracket   #\[ )   ;; 91
(define-punctuation  backward-slash        #\\ )   ;; 92
(define-punctuation  close-square-bracket  #\] )   ;; 93
(define-punctuation  caret                 #\^ )   ;; 94
(define-punctuation  under-bar             #\_ )   ;; 95
(define-punctuation  back-quote            #\` )   ;; 96

(define-punctuation  open-curly-bracket  #\{ )   ;; 123
(define-punctuation  vertical-bar        #\| )   ;; 124
(define-punctuation  close-curly-bracket #\} )   ;; 125
(define-punctuation  tilde               #\~ )   ;; 126
```

3.1.5.2 'Meaningless' characters

All of the other non-alphabetic, non-digit ascii characters except the three just below are identified by the tokenizer's default table as 'meaningless'. These the characters with codes 0, 3, 5, 6, 7, 8, 11, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, and 127.

Since the inclusion of one of these characters in a text is usually a mistake (for example the result of line noise when retrieving a file from an online news service), the default action when the tokenizer encounters a meaningless character is to go into a Lisp 'break'. This break can be inhibited by setting the symbol `*break-on-meaningless-characters*` to `nil`.

3.1.5.3 Reserved words

Three control characters have been reserved for Sparser's internal use in marking the beginning and end of the text source and the end of a character buffer (see §6.2.2). These are ascii 1 (^A), 2 (^B), and 4 (^D) respectively. The source start and end are also represented by word objects since they are entered into the chart in the course of a parse. The symbols below are provided for referencing them:

`source-start` `defconstant`

> This symbol is bound to the word object for the ascii character ^A (1). The parser places this word in the first position of the chart (token position 0) as part of its initializations when a call to the parser is made.

`end-of-source` `defconstant`

This symbol is bound to the word object for the ascii character ^B (2). The parser places this word in the chart at position immediately following the position of the final token of the string being analyzed.

3.1.5.4 Spaces

In addition, words have been predefined to represent each contiguous sequence of spaces from length one through 16. (It is important to keep track of the number of spaces in a sequence when processing such things as paragraphs or tables.) The word for one space is named `one-space` (i.e. it is bound to that symbol in the `word` package), the others are named `2-spaces` through `16-spaces`. The property lists of these words record the number of spaces they contain after the tag `:number-of-spaces`. It also includes the symbol :use-symbol-name-when-printing.

If a larger number of contiguous space characters occurs in the text and tokenizer has been directed to define any unknown word it encounters, then a word object will be constructed for the sequence, which will have a symbol on the same pattern as those already defined, i.e. the number of spaces followed by "-SPACES". If you want to define some specific number of spaces this form is provided:

`define-number-of-spaces (symbol string-of-spaces)` `macro`

Defines and returns the word whose pname is the indicated string. Marks the word-object as whitespace and to be printed using the indicated symbol (e.g. 37-spaces) rather than its pname.

A number of other functions for accessing, and testing for spaces are also provided.

`number-of-spaces-named (integer)` `function`

Returns the word object whose pname consists of the indicated number of spaces, or nil if no such word has been defined.

`spaces-word? (word)` `functionn`

A predicate that returns `T` if the word was defined with Define-number-of-spaces, and `Nil` otherwise.

`number-of-spaces (spaces-word)` `function`

Returns the number of spaces in the spaces word as an integer. Signals an error if any other kind of word is passed to it.

3.1.6 Whitespace

Spaces are the most obvious instance of a class of words (in Sparser's sense) that we can call "whitespace". This notion of whitespace is important in understanding what tokens become chart terminals. Margins, between-word spaces, carriage returns, blank lines, and the like are whitespace in the usual sense—portions of the printed page that are literally white. These parts of the text are rarely significant linguistically. In a character-based representation of a text, however, these parts are of course represented by

sequences of characters just like normal words and punctuation, and the Tokenizer will recognize them as word objects.

Since the parser is driven by the adjacency of labeled constituents (edges)—which ultimately comes down to the adjacency of words—we have to ensure that 'real' words, which we want to treat as logically adjacent, are in fact placed in adjacent locations in the chart, while the whitespace words are left out.

To capture this, a word can be explicitly marked as "whitespace". This has already done within the parser for spaces, tab, and the newline character. One can mark *any* word as whitespace by using the function below. This could be done for verbal interjections like "uh", or even for punctuation or determiners if they played no role in one's grammar. Once marked, these words become virtually invisible to the parser. They are retained in the chart however, see §6.4.2.

`define-to-be-whitespace (string)` `function`

> The `rules` field of the word with the input string as its pname is set to the keyword `:whitespace`, replacing that field's prior value, which is not saved, thereby disconnecting the word from any phrase structure rules that it may have been incorporated in earlier. No check is made to find such rules, which will simply fail to complete.

By the same token, one can remove the keyword `:whitespace` from a word to which it is assigned by default and it will now become visible to rules just like any other kind of word. This must be done explicitly; one cannot simply refer to a whitespace word in defining a rule, as this will signal an error. Because one often wants to change the visibility of newlines (as when parsing headers or tables), there is a facility already provided for this; see §6.4.3.

To test whether a given word is currently whitespace use this function:

`whitespace (word)` `function`

> This function is a predicate that tests whether the word object it has been passed is 'whitespace' and so will not be seen as a terminal by the default routine (§6.4.1). It returns `t` if the word is whitespace and `nil` if it is not.

3.1.7 Deleting words

To 'delete' a word is to expunge the word object from the indexes by which Sparser's tokenizer recognizes it. From that point onwards the tokenizer will regard it as 'unknown' and act accordingly (see §6.3.2.3).

`delete/word (word)` `function`

> The word object passed in as the function's argument is unlinked from the indexes that make it known to the tokenizer. The symbol associated with the word is made "unbound". Returns the deleted word object.

If one subsequently (re-)defines the same string, the new word object that is created will *not* be `EQ` to the earlier object. Also, as presently defined, the act of deleting a word

does *not* affect any phrase structure rules or hooks that reference it. They will continue to refer to the old word object even though it will not be returned by the tokenizer. Any such rules will have effectively been deleted as well since they will never be completed even when tokens with the deleted word's characters appear in the text.

Consequently, the usual reason for wanting to delete a word is to clear out part of the grammar, rather than to make some subsequent modification to the word.

3.1.8 Limits on the length of words

For efficiency there is always a limit on the number of characters in a word. The default value of this limit is 100. If one attempts to define a longer word or if a longer sequence of characters of the same type is seen by the tokenizer (e.g. a line with 101 spaces on it) then an error occurs.

One can, however, reset the limit to whatever value one likes. (Experience with a several megaword newswire corpus has shown that while 100 is often too small, a limit of 1,000 has not yet been exceeded.)

`*word-lookup-buffer-length*` `defparameter`

The value of this symbol is the current limit on the length of a word.

`resize-word-lookup-buffer (integer)` `function`

Changes the limit on the length of a word to the value of its argument.

3.2 Polywords

A "polyword" is a sequence of contiguous words that the grammar writer has chosen to treat as though it were an atomic unit just like a normal word. (The term "polyword" was introduced by Joe Becker at the 1975 TINLAP meeting.)

The notation for a polyword is a sequence of words in quotations, e.g. `"United States"`. As with words, if all of the alphabetic characters in the string are lowercase then case will not be considered when Sparser is determining whether an instance of the polyword as occurred; if any of the characters are given in uppercase, then that precise capitalization pattern must occur.

Polywords are defined uniquely by the sequence of words that comprise them. If the same sequence is defined more than once, then the same polyword object will always be returned.

The criteria for whether there is deemed to be one or several words in a given string is identical to the one used by the tokenizer (§6.3). Each sequence of characters of the same type will constitute a token (word); however only non-whitespace words within the character string will be considered in defining a polyword. If, for example, one defined the string:

```
"July   4th"
```

as a polyword, the fact that there happen to be three spaces between "July" and "4" would be ignored since the word `spaces-3` is marked as whitespace. A text with any

number of spaces, tabs, newlines, etc. between any of the words will be accepted as an instance of the polyword, including the case where there is no space, e.g. `"July4 th"` would work.

Polywords are atomic units within the parsing process. The completion of a polyword precludes any other analysis of the words that constitute it. No rules can re-partition the words of the polyword into subphrases or combine them with other words or phrases to either side. Should one want these possibilities, then one can use the alternative of defining a rule that combines the words as individuals rather than as a unitary group. That is, rather than writing a phrase structure rule in terms of a polyword, as in

```
holiday -> "July 4th"
```

one could define it in terms of three separate words:

```
holiday -> "July" "4" "th"
```

These two formulations are equivalent in so far as they lead to the same constituent being formed over those three words. They are different in the respect that the polyword will be looked for first before any phrase structure rules are applied to that portion of the text (see §6.5) and that it will preempt any alternative analysis. (Suppose for example one had a rule that rewrote "4th" as a constituent labeled 'ordinal': Because of the polyword his rule would not complete and would not even fire.)

As with words, the usual way to create polywords is as a side-effect of mentioning them in a phrase structure rule. However, if one wishes to define polywords independently of rules, this form is provided:

`define-polyword (string)` `function`

> Returns the polyword object corresponding to the sequence of words in the string. If that polyword has already been defined, the same object is returned. It is an error for the string to consist of characters all of the same type, i.e. to correspond to a word rather than a polyword.
>
> An internal symbol in the `*word*` package will be created and its value set to the polyword object. This symbol will be case sensitive. The word print names embedded in the symbol will be separated from each other by a single space, e.g. `|July 4 th|`.

To check whether a string will define a polyword rather than a word one can use the following function, which consults the current values in the tokenizer's character table (§6.2.4) to ensure that the correct interpretations are made.

`not-all-same-character-type (string)` `function`

> Returns `t` if at least one of the characters in the string is of a different type than the others as defined by the values in the character table at the time the function is executed. If the characters are all of the same type it returns `nil`.

If it does not matter whether a string is rendered as a word or as a polyword, then the following function can be used to make the definition.

`resolve-string-to-word/make (string)` `function`

Defines and returns either a `word` or a `polyword` object depending on whether the characters in the string are all of the same type or not. Acts as a dispatch function for the two cases, so the usual conventions and side-effects for each case apply.

3.2.1 The appearance of polywords in the chart

To have their effect in the parsing process, polywords are implemented in terms of phrase structure rules ('cfr objects', see §4.1.3). They differ from regular rules in how they are triggered and in the procedure for checking whether they complete. The fact that they by definition consist exclusively of words allows a particularly efficient process to be used.

When a polyword is found, that span of words will be covered with an edge whose label ('edge category') is the polyword. The daughter fields of the edge will contain `nil`, as will the form and referent fields. (See §5.4 for discussion of the fields in an edge object.)

The phrase structure rule object created for the polyword is a normal cfr object. Its righthand side will be a list of one element, the polyword. Its lefthand side (its 'category') will also be the polyword. Its property list will contain one additional tag–value pair: the keyword `:polyword` and the polyword. If the polyword consists of more than two words then the rule will be 'n-ary' (§4.1.5).

If the polyword succeeds, then only one edge will be formed. If, on the other hand only some prefix of the polyword appears in the text, then several edges will be formed: one over each of the words and one over each sub-prefix. This follows from the fact that polywords are implemented as regular phrase structure rules and will share much of their behavior, particularly that of n-ary rules. The edge over each word individually will be the usual 'literal-in-a-rule' type edge, and the edges over each prefix will be 'dotted-intermediary' rules. See the section of n-ary rules (§4.1.5) for an explanation.

The 'trigger' that initiates the checks for a polyword is the first word of the polyword's sequence of words. That word will have some cfr object in the "fsa" field of its `rule-set`. A given word can start any number of polywords, and so which cfr will be in the field is not predictable.

3.2.2 The fields in a polyword

Polyword objects are defstructs of type `polyword`. As structures, polywords are quite similar to words (§3.1.4). Both have fields for the symbol that points to them, the plist that records miscellaneous properties, the pname that gives the string that defines them (this string has canonical whitespacing between the words, i.e. one space), and the rules field that links them to Sparser's run-time operations (i.e. phrase structure rules and hooks).

A polyword has one field that words do not. This is the field "fsa" (for 'finite state automaton'). It contains the cfr object created to implement the polyword as discussed just above.

`pw-symbol (polyword)` `access-function`

Returns the symbol that was constructed and bound to this polyword at the time it was defined. It is in the `word` package. The symbol-name of the symbol is the same string as the pname of the polyword, which means that most references to this symbol will have to be enclosed in vertical bars so that Lisp references the intended case sensitive string, e.g. `word::|July 4 th|`.

`pw-pname (polyword)` `access-function`

Returns a string. This capitalization-sensitive string establishes the identify of the polyword to the definition routine, and is the basis of the phrase structure rules that recognize it in parsing

`pw-rules (polyword)` `access-function`

Returns one of these values: either a 'rule-set' object, the keyword symbol `:whitespace` (§3.1.6), or else `nil` when the word has no rules associated with it. See §4.6 for a description of rule sets.

`pw-plist (polyword)` `access-function`

Returns a list of paired objects that is intended to be employed as a 'property list' in the usual sense.

`pw-fsa (polyword)` `access-function`

Returns the cfr object (phrase structure rule, see §4.1) that was constructed to recognize this sequence of words when the polyword was defined.

3.2.3 Managing polywords

The polyword corresponding to a given multi-word string can be accessed as follows:

`polyword-named (string)` `function`

Returns the polyword object whose word sequence is the same as the sequence of words in the input string. If no such polyword has been defined then Nil is returned.

One can also access a polyword by the symbol that was assigned to it when it was defined. The symbols that allow convenient access to words or polywords are both in the same package: this package is the value of the symbol `*word-package*`, which will use the symbol `WORD` unless it has been changed to avoid a name clash with the user's system (see §2.4). Using the normal Lisp convention for accessing package-specific symbols, one could thus access our example polyword with the following symbol. Note the use of vertical bars to insure that the lookup is case-sensitive, and the presence of the space between the digit "4" and the word "th".

```
word::|July 4 th|
```

A list of all the polywords that have been defined is kept on the following symbol. The list is ordered from the most recently defined polyword to the earliest.

`*polywords-defined*` `defparameter`

To delete a polyword you use one of the following functions:

`delete-polyword (string)` `function`

The polyword corresponding to the input multi-word string is looked up. That object is then unlinked from the indexes that make it known to Sparser, and the symbol bound to it is made 'unbound'. Returns the deleted polyword object.

`delete/polyword (polyword)` `function`

This is the version to use when the polyword object is already available. It is otherwise the same.

When a polyword is deleted the words that comprise it remain defined.

3.3 Categories

Sparser uses objects of type "category" to represent the labels on most non-terminal edges. Non-terminals can also be labeled with words or polywords; we will use the term *label* when we want to refer to any of these three types of objects equally.

A category object is uniquely defined by the case-insensitive spelling of the symbol used to refer to it. This symbol is interned in the `*category-package*`, which will have the name `category` unless it has been renamed to avoid clashing with a user package (see §2.4). Categories may be accessed with the following function, or by referring directly to the symbol using its package, e.g. `category::month`.

`category-named (symbol)` `function`

Returns the category object corresponding to the input symbol, or `nil` if no such category has been defined.

Categories, like words and polywords, are defined automatically when they are used in a phrase structure rule. One can define a category on its own by using the macro below; but as a non-terminal label, no category will ever appear in an analysis unless there are rules that define it as the left-hand side of some phrase structure rule.

`define-category (symbol)` `macro`

Returns a category object. If the symbol was defined as a category earlier, the same object will be returned. The symbol is interpreted without concern for its case—the default for Lisp symbols. Note that this routine is a macro—its symbol argument should not be quoted, e.g.

```
(define-category holiday)
```

As a side effect of the definition, an internal symbol is created in the `*category-package*` and bound to the category object. The pname of the

symbol is the same as that of the input symbol. The symbol is declared 'special'.

Note that when the model is loaded, then `define-category` has a much larger set of arguments and creates a different type of object. See §9.1.

3.3.1 The components of the category object

Categories are implemented as defstructs of type `category`. Their components can be accessed by the following access functions, each corresponding to a field in the structure in the usual fashion. As Lisp objects, categories print in sharp sign notation as, e.g. `#<category MONTH>`, where the term in uppercase is the symbol used to define the category and stored in its "symbol" field. It appears in uppercase because that is Lisp's default for printing symbols, and reflects the fact that categories are case insensitive.

`cat-rule-set (category)` — `access-function`

Returns a 'rule-set' object, or `nil` if the category has no rules associated with it. See §4.6 for a description of rule sets.

`cat-plist (category)` — `access-function`

Returns a list of paired objects (or `nil`). It is intended to be employed as a 'property list' in the usual sense.

`cat-symbol (category)` — `access-function`

Returns the symbol that was constructed and bound to this category at the time it was defined. It is in the `category` package. The symbol-name of the category is the same as that of the symbol used to define the category.

3.3.2 Managing Categories

As with the other types of labels, a category object can be accessed from its symbol. All defined categories are also stored in a list. The list is ordered by the sequence in which the category was defined; new categories appear at the beginning of the list.

`*categories-defined*` — `symbol (defvar)`

Category objects are deleted using one of these functions.

`delete/category (category)` — `function`

The object for the category is unlinked from the indexes that make it known to Sparser. The object is returned. A new object will be constructed the next time a call to `Define-category` with the symbol for this category is made.

Note: deleting a category does not remove that instance of the category object from any other structures that might reference it. In particular, this means that any phrase structure rules or completion routines (§7.1) that were defined

with respect to the, now-deleted, category object will still refer to it. This can lead to a confusing situation if care is not taken.

`delete-category (symbol)` `function`

The category with this symbol is looked up and passed to `delete/category`.

3.4 Generic Operations

There is a set of functions that operate over any type of label. These are used internally in creating rules, since words, polywords, and categories can be used in the same places within rules, and the same sorts of things are done to them. If you make extensions that operate over these types of objects you may also find them useful.

3.4.1 Property lists

All labels have a property list:, usually referred to as a "plist" — a simple list of alternating tags and values. You can retrieve or set an object's plist as a whole, get or set the value of individual properties, remove properties, or change their value.

`plist-of (obj)` `function`

`set-plist-of (obj plist)` `function`

`get-tag-for (tag object)` `function`

`remove-property-from (obj tag)` `function`

`change-plist-value (obj property new-value)` `function`

`push-onto-plist (obj value tag)` `function`