6. Analyzing a text

In this section we describe the steps in the analysis of a text, including initiating the process, handing characters, the tokenizer, populating the chart, the timing of operations over terminals, the chart-parsing algorithms, and the various event-driven hooks for user code.

We will begin with what happens the moment that a call to Sparser is initiated. The initiating functions (§2.1) differ only in where they take the character stream from: either from a string that is typed in as the argument to the call (for analyze-text-from-string), or from a file (for analyze-text-from-file). In both cases, what they do is first setup the character buffers (§6.2)—thereby establishing the 'source' to be analyzed—and then make a call to the function analysis-core.

6.0.1 The Analysis Core

Every run that Sparser makes over some source text is mediated at its toplevel by the Sparser function `analysis-core`. It is called from one of the initiating functions and makes all of the calls to initialize Sparser's data structures and state variables, calls the parsing process, and executes any 'after analysis' actions. The function is quite simple: its full code is as follows:

```
(defun analysis-core ()
  (initialize-tokenizer-state)
  (initialize-chart-state)
  (per-article-initialization)
  (chart-based-analysis)
  (after-analysis-actions)
  :analysis-completed )
```

The first two initializations are fixed, required aspects of Sparser's operation. The third includes some required operations, but also includes a hook for user defined initializations as described in the next section (§6.1.2).

Sparser's analysis operations proper are initiated by the fourth call—`chart-based-analysis`. Only at that point do we begin to consume the source text, populate the chart, refer to the grammar to introduce edges, reach hooks and run user-written routines, etc. This function will not return until the analysis of the entire text is complete. The kind of analysis that is done is dependent on the parameter settings in force before the run begins as discussed in §6.7.

The value returned by the function that initiates Sparser (§2.1) will be the value that Analysis-core returns: the keyword symbol `:analysis-completed`.

6.0.2 After-Analysis actions

When chart-based-analysis has returned, which may be either because the source text has been completely processed or because some user routine elected to terminate the

process early (see §2.1.2), the function After-analysis-actions is called and looks for a set of forms to execute. Sparser has no standard after-analysis actions; any actions that are taken will have been specified by you as described here.

An action is specified as a Lisp form that will be passed to `eval`, i.e. a list giving the function to be called and any arguments that it takes. The arguments, if any, must be taken from the global environment since there is no local environment for these calls.

`*after-analysis-actions*  symbol (defvar)`

Points to the list of forms, s-expressions, that will be evaluated by After-analysis-actions in the order in which they are listed. The list is managed by the following two functions.

`define-after-analysis-action (s-exp)  function`

Adds the s-expression—typically a list—to the list of after-analysis actions. The s-expression must be suitable as an argument to `eval`. Any errors in the execution of the expression will not be trapped. The expression is added to the front of the list of actions, and will be executed before any of those already on the list. Returns the number of actions on the list after this one has been added.

`remove-after-analysis-action (s-exp)  function`

Removes the indicated s-expression from the list of after-analysis actions. The expression must be given in the identical form that was used to define it; the function `equal` will be used to test it against those on the list at the time.

`list-after-analysis-actions ()  function`

Prints a list of the s-expressions that will be evaluated as after-analysis actions. The order in which they appear is the order in which they will be executed.

6.1 Initializations

Two aspects of Sparser need to be initialized. The first are long term considerations that are usually determined once and only changed when experience shows that they should be adjusted. These include fixed data structures that need to have their sizes specified and the choice of various processing protocols. All of these items are given reasonable default values when they are defined in Sparser's code. The second aspect are the 'per-run' initializations, whereby Sparser's state variables and data structures are given the values that they must have at the start of a text's analysis.

We will talk about the first type in terms of initializing a "session". A session begins when one launches or loads Sparser and ends when one kills its process. A session will encompass any number of "runs", which are the individual calls to one of Sparser's initiating functions (§2.1) to analyse a particular text.

6.1.1 Initializing sessions

Sessions are initialized as part of the process of loading Sparser. The initialization directives are given in two files, and users are expected change these files as needed to provide the values that they they find appropriate to their applications of Sparser.

The first file has its effect at the beginning of the loading process, and establishes the sizes of Sparser's fixed data structures. This file is stored in the location below; see discussion in §2.2 for the interpretation of this filename.

```
…/Sparser/configuations/loader
```

This file consists of a set of parameter specifications. All the parameters are described here in this document, and the forms in the file include references to the appropriate section where one can get the information needed to understand their effect.

Initially all of the specifications use default values and are commented out since they are no different from the values embedded directly in the code. If you want to override a default, remove the blocking semi-colons and change the value to what you want. Note that in most cases these parameters can also be changed during the course of a session (sometimes requiring some instalation function to be run, as described in the section that defines them), and one can experiment with different values before making changes to this file and committing to them for future.

The second file has its effect at the end of the loading process. It is loaded after all of the regular files and is the standard place for customizing the values of standard tracing variables (§8). It is the point where the resources for the chart and edges are instantiated. It is also the place where switch settings are established (see just below) It is stored in:

```
…/Sparser/configurations/launch
```

In some versions of Lisp, such as MCL for the Macintosh, there is a provision to have a set of forms executed when a saved Lisp application is launched (§2.3). In such cases one will want to have this file loaded at launch time, as its name suggests. Doing this will allow one to modify these initializations between sessions.

6.1.1.1 Switch settings

Many of the processes and protocols described in this chapter can have different settings depending on the user's goals in deploying Sparser for a particular application. Often the protocols need to be used in particular combinations, and this is facilitated by the definition of "switch settings". These are functions that call the appropriate establishment routines or set the appropriate parameters all as a group so that the coordination is assured.

A standard switch setting will have been already provided for one's site. Modifications should be done by starting with a standard function and adding to it. Below is an example of this, starting from the setting Aal-edges-setting. (The name goes with a choice of parsing algorithm; see §6.7.2.)

```
(defun all-edges-setting/bbn ()
  (all-edges-setting)  ;; the base setting that is being modified
  (what-to-do-with-unknown-words :capitalize-&-digits)
  (setq *make-edges-for-unknown-words-from-their-properties* nil)
  (setq *switch-setting* 'all-edges-setting/bbn))
```

To view the current settings, use this function:

`switch-settings ( &optional stream )` function

Prints a sequence of statements to the indicated stream which give the current values of the parameterized procedures and protocols. The stream defaults to `*standard-output*`.

This is an example of the display one gets.

```
> (switch-settings)

Sparser switch settings: TOP-EDGES
            character translations: NO-CHANGES
 *break-on-meaningless-characters*: T
                     unknown words: CAPITALIZATION-&-DIGITS
             protocol for newlines: RETURN-NEWLINE-TOKENS/LINEFEED
     is a newline word whitespace?: T
                     Next-terminal: PASS-THROUGH-ALL-TOKENS
       capitalization FSA dispatch: NO-OP
               type of edge-vector: VECTOR
              Chart-level protocol: NEW-TOPLEVEL-PROTOCOL
                          Complete: CA/HA
      include property-based edges? T
               Length of the chart: 500
       Length of the edge resource: 500
```

6.1.2 Initializing runs

As indicated in the discussion of the Sparser function analysis-core (§6.0.1), at the start of each run of Sparser there are standard initializations of the state-driven processes: the tokenizer and the chart, and there is a facility with which the user can define their own standard initializations. We describe that facility here. Its design is the same as that of the 'after analysis' actions described in §6.0.2.

`*per-article-initializations*` symbol (defvar)

Points to the list of forms, s-expressions, that will be evaluated as part of the operations of the Sparser function per-article-initializations, see §6.0.1. The list is managed by the following two functions.

`define-per-run-init-form (s-exp)` function

Adds the s-expression—typically a list—to the list of per-article initializations. The s-expression must be suitable as an argument to `eval`. Any errors in the execution of the expression will not be trapped. The expression is added to the front of the list of forms, and will be executed before any of those already on the list. Returns the number of forms on the list after this one is added.

`remove-per-run-init-form (s-exp)` function

Removes the indicated s-expression from the list of per-article initializations. The expression must be given in the identical form that was used to define it; the function `equal` will be used to test it against those on the list at the time.

`list-per-run-init-forms ()` `function`

Prints a list of the s-expressions that will be evaluated as per-article initializations. The order in which they appear is the order in which they will be executed.

6.2 The Character buffer

Sparser's input is a sequence of characters, either a string or an ascii file that can be read as a character stream by the Lisp being used (§2.5). We will refer to this stream as the character 'source'. Characters are moved in batches from their source into a fixed array structure where they are scanned sequentially by the tokenizer. The tokenizer takes all of its input from this 'character buffer'. The design permits an eventual implementation by co-routines, and avoids the need to dynamically allocate storage ('cons') at the character-handling level.

Besides providing a uniform location for the character stream and thereby factoring out the management of the stream from downstream analysis operations, the character buffer is also a convenient basis for arranging displays of the results of an analysis, and it is with that application in mind that the information on the buffer is provided here.

6.2.1 State information

The character buffer consists of two arrays of element-type 'character'. Both arrays are of length `*length-of-character-input-buffer*`, which is set as part of the session initializations (see §6.1.1) and defaults to 1,000.

At any given moment, one of the two arrays is pointed to by the symbol `*character-buffer-in-use*`. This is the array from which the tokenizer is taking its characters. The other array is pointed to by `*the-next-character-buffer*`. The flag `*buffers-in-transition*` has the value `T` during the short time while the tokenizer notices that it has scanned all of the characters of the buffer in use and switches over to the other buffer.

The buffer switching process is transparent to token boundaries and can occur in the middle of a token without noticeable effect beyond the negligible time taken to fill the next buffer with characters from the source and swap the pointers.

The state of buffer facility is initialized at the beginning of each run. The two arrays are permanently pointed to by the symbols `*first-character-input-buffer*` and `*second-character-input-buffer*` and are used in that order.

Besides the two symbols that keep track of which array the characters are currently being taken from and which array will be used (or reused) next, the other parts of the character buffer's state include:

`*index-of-next-character*` `symbol (defvar)`

Its value is an integer designating the array cell containing the character that the tokenizer will look at next. It is initialized to -1 at the beginning of the run.

`*length-accumulated-from-prior-buffers*` `symbol (defvar)`

Its value is an integer that is some multiple of one-minus the length of the buffer arrays.

Adding the two numbers yields the position of the next character with respect to the source character stream from which Sparser's input is being taken.

The point in the character source that the buffer-filling process has reached at a given moment is determined differently depending on whether the source is a string or a file. If it is a string, the point is calculated from the accumulated length as just described. If it is a file, it is maintained by Lisp as part of the internal information of stream that is being kept open to the file. This file stream is pointed to by the symbol `*open-stream-of-source-characters*`. The designated file (see §2.5.1) is opened for input as the first action of a file-based run, defining the stream in the process. The stream is closed when the function `Terminate-chart-level-process` is executed.

When the source is a string, its characters are simply 'replaced' into the cells of the arrays. When the source is a file, the array cells are filled by a sequence of single-character accesses from the file, which remains open until all of its characters have been read.

Note that while the pointers (symbols) that keep track of the state of the buffer are initialized with each new run, the arrays themselves are not. The cells of the arrays will continue to contain whatever characters they happened to have been filled with during earlier runs until they are re-filled during the current run.

Assuming there are enough characters remaining in the source, the buffers are completely filled each time they switched to. The first buffer is filled at the start of the run, and the alternating 'other' buffer array is filled at the moment that the tokenizer has noticed that it has reached the end of the 'current' array. When the source is a string, its next pending substring up to the length of the array is replaced into the target array and any remaining unconsumed portion of the string noted for the next change-over. If the unconsumed portion of the string is smaller than the length of the array, then all of it is replaced into the array and the portion of the array beyond that left unchanged. Similarly when the source is a file, characters are taken from it until either the array has been filled or the end-of-file indicator is seen, and the unfilled cells of the array retain the characters they had in previous runs.

6.2.2 Special control characters

Three control characters in the ascii character set have been reserved for Sparser's exclusive use in managing the character buffer and signalling the boundaries of the source.

At the very beginning of each run, the character `#\^A` ('control-A'—ascii value 1) is introduced into cell zero of the first array (which is its first cell position given Lisp's

convention of zero-based indexing). This character is understood by the tokenizer to indicate the start of the text and the corresponding 'source-start' word is introduced into the chart (see §3.1.6) when the character is scanned.

Since the system is using the first cell this means that the first character of the source will be placed in cell 1 of the first array.

The character `#\^B` ('control-B'—ascii value 2) marks the end of the source. It is placed in the cell that follows the final character of the source. The tokenizer introduces the 'end-of-source' word into the chart when it scans this character and sets the `*source-exhausted*` flag.

The third reserved character is `#\^D` ('control-D'—ascii value 4). It is placed in the last cell of both buffer arrays at the time they are created, and is used to signal to the tokenizer that an array buffer has been completely scanned and it should switch to the other buffer as discussed just above.

These three characters should never be included in the source. If they do occur, perhaps because of line-noise encountered when the source is collected, then they will signal their usual functions just as though they had been deliberately introduced by Sparser. An accidental control-D will cause the buffers to be switched prematurely, skipping over the remaining characters in that array as a result. An accidental control-A or control-B will cause the corresponding words to be introduced into the chart, interrupting the contiguity of the analysis or causing it to terminate early, respectively. Note that these are not printing characters and would not be seen in the usual displays. Should one suspect that spurious characters are the source of some problem the sequence of source characters can be traced as described in §8.1.

6.2.3 Modifying characters' interpretations

The characters of the source are seen by the tokenizer as seven-bit ascii values encoded as Lisp character objects. The interpretation of these objects for the purpose of identifying words and other defined characters such as punctuation or whitespace is indirect. It is mediated by a table that the user can manipulate.

The motivation for using a table of interpretations rather than using the source characters directly is largely efficiency. It expedites Sparser's treatment of words as distinct individuals independent of their capitalization (§3.1.1), and facilitates the lookup of the words corresponding to punctuation and other single-character entities.

The table provides an interpretation for all 128 ascii characters. Its default interpretations are natural ones appropriate to how the characters are printed. The table of course includes the alphabetic characters, both upper and lower case, and the numerals. The non-printing control characters are included but are marked as 'meaningless' except for the three reserved for Sparser's use. The control characters that print as punctuation characters are interpreted as such (see the list in §3.1.5.1).

When the tokenizer examines a character, it uses its ascii value to index into the table and retrieve the character's entry. This entry provides the basis of how the character

will be interpreted, specifying its 'character type', its case, and the actual character to be used in assembling tokens and looking up the corresponding words. Should users need to use some alternative interpretation, the entry can be changed as described below.

There are three kinds of character-types defined, with the tokenizer behaving differently depending on which one is given in a source-character's entry: `:punctuation`, `:number`, and `:alphabetical`. As discussed in the section below on the definition of a token (§6.3.1), a sequence of characters interpreted as being either all numbers or all alphabetical characters will be grouped into a single token; characters interpreted as meaningful punctuation are tokenized as singletons; 'meaningless' punctuation characters in the source signal an error (§6.3.1.1). The punctuation for `#\space` (by default ascii 32) is treated specially.

`set-tokenizer-table-entry` function

`( &key ascii-number character-type case character )`

This function changes the entry of the source character with the indicated ascii-number (an integer in base 10), to have the indicated character-type, character, and if appropriate, case.

The character-type must be one of these three keywords: `:punctuation`, `:number`, or `:alphabetical`.

If the type is punctuation or number then the desired 'character' must be specified. This argument must be a Lisp character object, e.g. `#\space` or `#\A`. A data-check is made to insure that characters to be interpreted as numbers are indeed one of the ascii number characters zero through nine; and for punctuation that a 'word' defining that character as punctuation has already been defined (see §3.1.5.1 for how this is done). Alternatively, for punctuation the keyword `:meaningless` may be given (see §3.1.5.2) or the keyword `:space`.

If the type is alphabetical then not only must the character be specified but also whether it is to be interpreted as being in upper case (`#\A`) or lower case (`#\a`), using one of the keywords `:uppercase` or `:lowercase`.

For example, if for some reason one wanted the source character "A" to be interpreted as the digit "1", one would call the function like this:

```
(set-tokenizer-table-entry :ascii-number 65
                           :character-type :number
                           :character #\1 )
```

6.3 The Tokenizer

The lowest level of linguistic analysis is the scanning of the character buffer by the Tokenizer. The Tokenizer takes characters from the character buffers and aggregates them into meaningful groups ("tokens") corresponding to words, digits, punctuation, and whitespace. The Tokenizer is the manager of the character buffer, maintaining pointers into the buffer arrays and initiating their filling and recycling.

6.3.1 What is a token

The goal behind the design of Sparser's tokenizer is to provide a useful but conservative initial aggregation of the characters of a text—useful, in supporting the standard notions of words, numbers, and punctuation; conservative, in avoiding any presumptions about larger aggregations whose interpretation is likely to be domain specific. Such interpretations are the province of the grammar operating over edges in the chart. Thus for the following string, which in some tokenizers could be defined to be a 'money' token,

"*$43.2 million*"

the Tokenizer will see six tokens: "`$`", "`43`", ".", "`2`", " ", and "`million`". Should you want to have the string interpreted as an amount of money you do that by writing the appropriate set of grammar rules.

This design decision allows tokenizing to be done efficiently using just local criteria, while the more elaborate mechanisms available at the chart level allow context-sensitive criteria to be applied to distinguish among domain specific interpretations, as for example in deciding between whether "*F-15*" refers to the warplane or the function key.

Notice that the tokenizer does not throw away whitespace characters; rather assembles them into tokens and passes on the corresponding objects no differently than it does with regular words.

The way that characters aggregate into tokens is dictated by their type as just described in §6.2.4. Any contiguous sequence of characters of type `:alphabetical` or of type `:number` will constitute a token. Characters of the third type, `:punctuation`, may be reserved for Sparser's internal use (§6.2.2), may be 'meaningless', may be the usual punctuation characters (listed in §3.1.5.1), or may be a 'space'. Their treatment by the Tokenizer varies:

6.3.1.1 'meaningless' characters

Meaningless characters are those ascii characters that do not have a meaning in the text sources Sparser is to analyze and may, for example, be the result of line noise or other contaminants. A character is designated as meaningless by its entry in the tokenizer table (§6.2.4). What happens when a character interpreted as meaningless punctuation is scanned by the tokenizer depends on the value of this flag:

`*break-on-meaningless-characters*` `defparameter`

> If this flag is not `nil` (the default), a break loop will be entered whenever a meaningless character is scanned by the tokenizer. If you continue from the break, the analysis will continue and the character will be ignored. Setting the flag to `nil` avoids the break, but a message will still be printed to `*standard-output*` announcing that the character has been seen.

6.3.1.2 spaces

Contiguous sequences of spaces are treated as a single token. The resulting word may be one that is already defined or a new word may be defined at that moment. Only the usual space character `#\space` (ascii 32) is marked as space punctuation in the default

tokenizer table. Should one choose to define several characters as ‘spaces’, a sequence of them would be seen as all one word even if different characters were involved.

6.3.1.3 punctuation

All other punctuation characters include their corresponding word object as part of their definition in the tokenizer table. This word is returned as the token whenever a normal punctuation character is scanned. A contiguous sequence of punctuation characters, even of the same character, is treated as sequence of single-character tokens.

6.3.2 The data structure for ‘tokens’

A call to the tokenizer returns the `word` object (§3.1) that corresponds to the next token after the point in the character buffer where the tokenizer ended when the last call was made to it. There is no data type specific to the tokenizer alone.

All tokens are interned, i.e. all instances of the same character sequence will be mapped to the same word object.

As described in §6.6.2, the word returned by the tokenizer as the interpretation of a given sequence of alphabetic characters is always the lowercase ‘canonical’ version of the word. The capitalization pattern that that particular instance of the word had is available in the following symbol as soon as the accumulation of the characters of the token is complete:

`*capitalization-of-current-token*` `defvar`

The value of this symbol is defined whenever a call to the tokenizer returns. It will be one of the following keywords:

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

In addition, the tokenizer maintains information from which the position occupied by a token’s characters in the character buffer can be calculated.

`*index-of-next-character*` `defvar`

This index is initialized to minus one when an analysis begins. When the tokenizer wants to look at the next character it increments this index by one and accesses the corresponding cell of the current character buffer array (§6.2).

When it is accumulating alphabetic characters or digits, the tokenizer will have to look just beyond the last character of its token in order to determine that the token is finished; in these cases when the call to the tokenizer returns the index will pick out the first character of the next token.

If the token is a punctuation character other than space—by definition only of length one—this index will pick out the cell containing the punctuation character.

`*length-of-the-token*` `defvar`

The value of this symbol will be an integer, corresponding to the number of characters in the token when the tokenizer returns. Note that no token can ever have length zero.

The case of punctuation aside, the index of the first character in the token is obtained by subtracting the length of the token from the index of the next character once the tokenizer has returned, plus the length of any already scanned character buffers.

6.3.2.1 Known words

A word is 'known' if it has been defined by `define-word` (§3.1.2) or used on the right or left hand side of some phrase structure rule (§4.1.4). That process of explicit or implicit definition creates an object of type `word` whose identity is dictated by its sequence of alphabetic characters. When the tokenizer scans a sequence of characters that have been defined as a word it will return the identical object as created by the definition. This ensures that the objects used in the internal representation of the rules will be same as the ones passed into the chart by the tokenizer, and so can be compared directly by efficient tests (e.g. Lisp's `EQ` function).

The table used to intern words and ensure that the appropriate objects are returned for a given character sequence is *not case sensitive*. This means that while a word, say "President", may be defined in terms of marked case (i.e. with one or more of its letters capitalized), the tokenizer will not return the object for that case sensitive word. Instead, it will return the word object for its lowercase equivalent ("president")—the version that Sparser considers 'canonical'—and the case exhibited by this instance of the word will be recorded in the symbol `*capitalization-of-current-token*`, as described just above, and will be stored in the chart at the position where the word is placed. The representation of words provides pointers between lowercase words and their variants (and visa versa) that Sparser and the user can use to relate the two; see §6.6. For preterminal chart edges over words, the mapping to the appropriate non-canonical word is handled automatically (§6.6.2).

6.3.2.2 Digit sequences

A sequence of digits is treated in just the same way as a sequence of alphabetic characters. If the sequence has already been defined then it will be mapped to the corresponding word object. If it has not then it will be treated as an 'unknown word'.

6.3.2.3 Unknown words

A token consisting of a sequence of alphabetic characters or of digits that has not been defined as a word is an 'unknown word'. What the tokenizer returns for them is determined by the 'unknown-word policy' currently in effect; see §6.6.3. If the policy calls for constructing word objects for them—defining them implicitly when they are first seen, then subsequent instances of that same character sequence will return the same object.

As with known words, the object returned by the tokenizer will always be the lowercase canonical version (assuming that the policy calls for defining word objects). However, since they are unknown, they by definition do not lead to preterminal edges

when are scanned in the chart. Given this, no automatic mechanisms have been put in place for constructing word objects to represent non-lowercase words when an unknown word with marked case is seen by the tokenizer (though of course the capitalization information is always kept in `*capitalization-of-current-token*` and in the chart). It is up to the user to notice and construct the needed objects if they are important for her algorithms.

6.3.3 Calling the Tokenizer

As a process, the tokenizer is organized as a function that returns a single token (word object) each time it is called. Each successive call to the tokenizer will return the next token in the character sequence. The process is one way only: once a portion of the source has been tokenized it can not be retokenized—there is no provision for 'backing up'. Any user algorithms that need to move back and forth across a sequence of tokens should be implemented at the level of the chart where there is access to earlier words up to the point when the chart recycles.

`next-token ( )` `function`

Returns the word object corresponding to the next token in the input text after the token found in the previous call to next-token. The state information which keeps track of this sequencing is initialized with each user call to Sparser and updated automatically with every call to next-token.

The first word returned by the tokenizer as the analysis of the source is begun will be the word that is the value of the following symbol:

`*source-start*` `symbol`

The value of this symbol is the word object that has been designated to represent the beginning of a text.

Once after all of the source has been tokenized, the character that indicates the end-of-source will be reached in the character buffer, and a special word will be returned:

`*end-of-source*` `symbol`

The value of this symbol is the word object that has been designated to represent the token that the tokenizer returns as the next token the first time it is called after the characters of the text under analysis have been completely scanned.

Note that it is the responsibility of any user driver routine that makes calls to `next-token` to check for the 'end of source' word. There is no constraint on the tokenizer that would automatically keep it from looking beyond the end of the source: `next-token` can be called indefinitely. The words that the tokenizer would return after it has returned the end of source word will be arbitrary, reflecting whatever characters happen to be in the rest of the character buffer at the time. Barring hitting the breakpoint for meaningless punctuation, checks against `*end-of-source*` must be made or else a loop of calls to `next-token` will never terminate.

6.4 Populating the chart with terminals

One can use Sparser as just an engine for tokenizing and examining the words of a text without bothering to also define phrase structure rules or deploy Sparser's edge-level protocols and higher level hooks. One reason to do this might be to collect word-frequency information about a corpus, another could be to do word-level topic classification of articles.

The mechanism for establishing what facilities Sparser deploys at its chart level are given in §6.7, and we leave for that section the further discussion of how one sets up a 'words only' operation. In this section we describe the operations that are common to both that mode of operating and to all the higher order, edge-based facilities. We describe the notion of a 'terminal'—the class of word objects that can occupy positions in the chart— as well as the function that introduces words into the chart and the state variables that that function uses, and the "newline hook" which controls the interpretation of carriage return characters.

6.4.1 "Terminals"

A "terminal" will be defined as any word object that is entered into the chart at some position, i.e. a word that becomes the value of the `terminal` field of some position object (§5.2.1). These words—which normally include the word objects representing punctuation and digit sequences as well as what we normally think of as words—are the ones that will participate in the parsing process by triggering phrase structure rules and that will trigger word-level hooks such as completion (§7.1) or the start of polywords.

Terminals are not the same as the tokens that are returned by the tokenizer. This is because not every token that the tokenizer identifies will normally go into the chart. The tokenizer is designed merely to segment the character stream and make the mapping from character sequences to defined word objects. Since it is conservative in design and does not throw out any sequences, this means that it also returns the word objects that represent spaces, tabs, etc. This 'whitespace' (§3.1.6) is usually not intended to trigger rules except in special circumstances (such as when parsing tables or message headers), and therefore would not usually constitute terminals.

Sparser includes a layer in its processing, the function `next-terminal`, where tokens can be filtered, completely hiding some classes of words or specific words from consideration by later processes, in particular keeping them from being added to the chart. This function is the next step in Sparser's processing after the tokenizer. It is called by the routines that put terminals into the chart and maintain the chart's state variables that are discussed later. The body of this function, and hence its behavior, is set when Sparser is launched, though it can be changed at any time, including in the middle of an analysis.

`next-terminal ( )` `switched function`

> Each successive call to this function returns a word from the token stream of the source. Alternative definitions can be provided (see just below), and the routine is well situated to act as a local buffer for limited lookahead.

This function is part of the core of Sparser's procedure for populating the chart. It is the only routine that calls the tokenizer. It is the way that higher level protocols that use scan-next-position access the source text. The words that it returns, which can be a subset of those returned by the tokenizer, will be the only ones seen by later processes—in this sense it is a word filter. The protocol it uses to do its filtering is dictated by its current switch setting. The default setting is `:pass-through-all-tokens`.

You change this routine—'switching' it from one behavior to another—by executing the function below. There is one predefined alternative to the default: The setting `:no-whitespace` passes through only word objects that are not marked as whitespace. The setting `:pass-through-all-tokens` will make it identical in action to next-token You can also set the behavior to whatever you want by including your own function as an optional argument.

`establish-version-of-next-terminal-to-use` `function`

```
( keyword  &optional function-name )
```

When only the keyword argument is supplied, this sets the function Next-terminal to have one of two predefined values corresponding to the two keywords `:pass-through-all-tokens` and `:no-whitespace`. Alternatively the Next-terminal procedure can be set to an arbitrary function by using a different keyword and supplying the symbol that names the function whose body one wants to use. This function must return one word object with each call, and in the process to consume some potion of the source text by either making calls to Next-token retrieving from a local buffer, or some other means.

`*definition-of-next-terminal*` `symbol (defvar)`

Points to the keyword used in the last call to Establish-version-of-Next-terminal-to-use, and with that the current value of the Next-terminal function.

6.4.2 The standard procedure

The default setting for next-terminal is `:pass-through-all-tokens`, which, as its name suggests, makes the function effectively a no-op. The purpose of its layer in the procedure for populating the chart is to provide a hook for special operations over the output of the tokenizer—a hook that is not used in the standard procedure for populating the chart but could be utilized by your own code.

The standard procedure for handling the disposition of whitespace as contrasted with non-whitespace is to add both kinds of word objects to the chart, but in different places. This adjudication is handled at the next level of processing up from the next-terminal hook, the level that also maintains the hook for the special treatments of newline characters (§6.4.3) and with that the potential for recognizing paragraph boundaries, indented margins, and the like.

At this next level, which we will refer to as the *scan* process according to the name of its driver function scan-next-position (see below), we manage the state variables that

keep track of which positions in the chart have been populated and manage the recycling of position objects within the chart array (§5.5). Precisely which positions receive which word objects is dictated by the values of globally bound variables managed by a sequence of routines.

The disposition of whitespace and non-whitespace words is done by the function Add-terminal-to-chart, which is called from scan-next-position. A customized driver could call this function to populate the chart with terminals without engaging any of the Sparser's edge-driven parsing machinery.

`add-terminal-to-chart ( )` `function`

> This function is executed for side effects. With each call it adds one word object to the chart at the position indicated by *next-array-position-to-fill* as discussed below.

Add-terminal-to-chart checks for the flag `*source-exhausted*`. The last word that will be added to the chart is one representing the end-of-source (§3.1.5.3). After this word is added into the chart, this flag will be up (it is set by the Tokenizer when the end of source character is seen). The flag will then cause a Lisp break loop to be entered if another call to add-terminal-to-chart is made after that point. This situation usually indicates a run-away word-level routine (§7.2) that must be debugged.

When add-terminal-to-chart is called, it accesses the position object in the chart that is the value of this state variable:

`*next-array-position-to-fill*` `symbol (defvar)`

> The value of this symbol is an integer. It is initialized to zero and is increased by one each time next-terminal returns a non-whitespace word. Its value can be as high as one minus the length of the chart array (`*number-of-positions-in-the-chart*`); when it reaches that value it is reset to zero, which is the index of the first position in the chart array.

If add-terminal-to-chart's call to next-terminal returns a whitespace word, that word object becomes the value of the position's `preceding-whitespace` field. If its call returns a non-whitespace word then that word object becomes the value of the position's `terminal` field and `*number-of-positions-in-the-chart*` is incremented. We can illustrate this with the following example. Let us assume we are analyzing the following source string:

"July 4th"

This eight character source yields the sequence of six tokens (words) shown below, with the identity of the tokens indicated here by the symbols one could use to refer to them. Note the use of vertical bars to name Lisp symbols with lowercase print names, and the fact that the word for "July" is returned by the Tokenizer in lowercase.

1: word::source-start

2: word::|july|

3: word::one-space

4: word::|4|

5: word::|th|

6: word::end-of-source

With the first call to add-terminal-to-chart, the next chart array position to fill will be the one in cell zero of the array. Next-terminal will return the first token of the source—which is *always* the word bound to the symbol `*source-start*`—and that word object, because it is not marked as whitespace, will be put in the `terminal` field of the position object. This position is position number zero in terms of its 'array index', as well as being position zero in terms of its 'token index' (see §5.2). Since the word is not whitespace, the pointer `*next-array-position-to-fill*` will be incremented to one.

With the next call to add-terminal-to-chart, next-terminal will return the word "july". The next position is now position number one, and its `terminal` field will be set to that word object. The position's `capitalization` field is set to `:initial-letter-capitalized`, reflecting the capitalization of this instance of the word, which the Tokenizer returned in its fully lowercase, canonical form (see §3.1.1). (The capitalization of the source start word is `:punctuation`, which has gone into the capitalization field for position zero.)

The next call to add-terminal-to-chart gets the word for one space, the first instance of a whitespace word in this sequence. The next position is the position object with array index two (which in this case is the same as its token index since the chart has not wrapped; see §5.5). This whitespace word goes into position two's preceding-whitespace field. The state variable `*next-array-position-to-fill*` is *not* incremented in this case, because we have not yet seen a non-whitespace word.

The next call returns the word for the digit four. The position is still position two. Its terminal field is set to the digit word. Its capitalization field is set to the keyword `:digits`. Add-terminal-to-chart now increments the pointer to the next position, moving it to position three.

The next call to add-terminal-to-chart operates on position three. Next-terminal returns the word representing "th", which becomes position three's terminal. Its capitalization field is set to :lower-case; its preceding-whitespace field remains empty (`nil`) because the immediately preceeding word in the token stream ("4") was not marked as whitespace; and the pointer to the next position is incremented.

The next call—the last one that we can make without triggering a Lisp break—has Next-terminal returning the special word representing the end of the source; its capitalization is `:punctuation`. The word is entered into the chart as the terminal of position four, and `*next-array-position-to-fill*` is again incremented, even though we will never need to use the next position.

At this point, if we were to run the built-in display routine display-chart-terminals (§8.1), we would get the presentation below. Notice that it emphasizes how the chart is viewed by later levels of processing as having the words located *between* the positions, and that the whitespace is not shown, just as it is not visible to the parsing process.

```
0  source-start  1  "july"  2  "4"  3  "th"  4  end-of-source
```

With your own driver you can fill the chart with terminals by making a loop that calls Add-terminal-to-chart until the `*source-exhausted*` flag changed from the value `nil` to `t`. However you can also populate the chart's positions as a side-effect of the standard function for moving through the chart during the parsing process: scan-next-position.

`scan-next-position ( )` `function`

> Returns the position object designated by `*next-chart-position-to-scan*` and then increments that pointer. If the position does not have a terminal at the moment the call to this function is initiated—if it has not yet been filled—the function calls add-terminal-to-chart, which will set the position's `terminal` field to the next non-whitespace word returned by the Tokenizer. The position's `pos-assessed?` field is set to the keyword `:scanned`.

Sparser's parsing algorithms move through the source text in one left to right pass, calling scan-next-position each time they want to advance to the next word, and taking that word from the position object that scan-next-position returns. This is the word that 'follows' that position, since positions are contrued as falling between the terminals of the chart.

Scan-next-position is the appropriate function to be used by any user-written parsing routine whose operation is interleaved with Sparser's parsing algorithm. This is because it incorporates the routines that maintain the 'assessment protocol', as well as this state variable that tracks and governs the left to right scan:

`*next-chart-position-to-scan*` `symbol (defvar)`

> The value of this symbol is an integer. It is initialized to zero and increases by one at the end of each call to Scan-next-position up to a maximum of one minus the length of the chart array (`*number-of-positions-in-the-chart*`). When it reaches that value it is reset to zero, wrapping the chart array (§5.5).

6.4.3 The Newline hook

The word object that represents the newline character[1] is treated specially by add-terminal-to-chart. This special treatment can be modified by the user with a small amount of programming, and thereby provides a 'hook' at which to record and react to the orthographic information in a text that is communicated by line lengths, vertical whitespace, indentation, centering, margins, and so on.

When a newline is about to be added to the chart, i.e. at the point when the punctuation character `#\newline` has been seen by the tokenizer and the word for it returned by next-terminal to add-terminal-to-chart, a dispatch occurs to the following function:

---

[1] Common Lisp supplies the character `#\newline` to represent the division between lines. Under a given operating system this might actually be the linefeed character, the return character; but in either case the character that separates one line from the next will be consistently recognized as `#\newline`.

`newline-fsa ( position )` `switched function`

Called by add-terminal-to-chart whenever next-terminal returns the word representing the newline character. The position object passed in is the next one to be filled. The default behavior of this function is simply to return the newline word. The function *must* return either the newline word or some other word that is marked as whitespace, otherwise an error will be signaled.

The newline word is accessible as the value of the symbol `*newline*`. The default value for the newline-fsa function is used in conjunction with the following flag:

`*newline-is-a-word*` `symbol (defparameter)`

If this symbol is not `nil`, then when the newline-fsa function returns the newline word object it will be entered into the chart where it can be seen by phrase structure rules or other actions, rather than being treated as whitespace, as it is when the flag is down, i.e. has the value `nil`.

By allowing the newline word into the chart, one can reference it in rules just like any other punctuation character (§3.1.5.1). One can, for example, write rules that can notice when some word appears at the beginning of a line, which is a way to recognize the header keys of text types like email or online news services in a way that specifically distinguishes them from instances of the same word in running text.

One has the option of redefining the newline-fsa to one's own routine. There is no routine for doing this, rather one has to use the Lisp mechanisms such a routine would use, i.e. one must explicitly `setf` the `symbol-function` of the symbol `newline-fsa` to the symbol-function of the function one wishes to use. This function must take one argument, a position object, and it must return either the newline word (which is set to the symbol `*newline*`) or some other word that is marked as whitespace.

One can revert to the original default behavior of newline-fsa by executing the function `use-return-newline-tokens-fsa`, with no arguments.

6.5 Coordinating word-level actions

Much of what Sparser does applies just to the words in the chart (the terminals) rather than the edges they may lead to. These take place with each successive scan of a position in the chart and constitute the standard processing of the word that follows it. These actions include checking for the end of the source; looking for the initiation or polywords; looking for words that trigger word-level analyses including routines developed by the user; running the word 'completion' hook; and installating any edges that the words dictate given the current grammar.

Since this is the level at which most user-developed word-driven analysis routines are executed, it is important to understand not only what actions are taken at this level but their timing and coordination, which is the purpose of this section. The individual actions themselves are described in subsequent sections of this chapter.

All of these actions take place within a loop that starts with a call to scan-next-position. This call returns a position object that we will refer to here just as "the

position", and from the position we access the word object in its `terminal` field, referred to as "the word".

The very first action is to compare the word with the word that is bound to `*end-of-source*`. If they are the same word (if the two objects are EQ), then the function terminate-chart-level-process is called (§2.1.2) and the analysis process is completed.

If we have not reached the last word in the chart, then we next check for any kind of word-level analysis process triggered by this particular word. The most common process will be checking for instances of polywords (§3.2), but user-developed analysis processes (§7.2) are also checked for at this point.

The possibility of a polyword is looked for first. If the word initiates one or more polywords but none of them occur at this point in the text, then we go on to look for a user-defined process. If one of the polywords is present, then we do not go on to check for user routines. A polyword always takes precedence over other possible analyses.

There may be more than one user routine initiated by the word, and they are executed in order until one of them indicates that it has succeeded. Once one routine has succeeded no other routines are tried.

When word-triggered actions finish, there are two possibilities for what happens next: one for the case where one of the actions "succeeded" and another for when either there were no actions associated with the word or none of them succeeded.

In the case of no actions or no successful actions, the next thing that will occur is the "completion hook" (§7.1). The word is passed to the word-completion hook driver and any completion actions associated with that particular word are checked for and executed.

After completion, the grammar rules associated with the word are accessed and any "preterminal edges" that the word defines are instantiated and added into the chart (§6.4). These edges will start at the position we have just scanned and end at the position just following it.

If the word led to the introduction of one or more edges, those edges are now checked for the possibility that any of them trigger some analysis process (§7.2). All the edges are checked and all of the routines associated with them (if any) are checked. If several of them succeed, the edges they create are all entered into the chart.

That completes the sequence of actions that can take place in the analysis of a word at the point when it is scanned. The process will now loop, scanning the next position and repeating this sequence of checks and actions on the word that follows it.

Going back now to the case where a polyword or some other word-triggered action has succeeded, we must first appreciate what "success" means in this situation. The intent of any of this class of actions, polywords included, is to examine the words of the text, starting at this position, and to check for the occurrence of some pattern. In the case of a polyword this is a sequence of particular words, starting with the present one. Other patterns are typically for such things as numbers expressed as sequences of digits or for proper names, i.e. patterns that consist of an idiosyncratic and potentially highly variable sequence of terminals, especially when the resulting constituency—and the calculation of the referent—is better represented in terms of a flat, multi-daughter edge instead of the binary branching tree structures typical of phrase structure rules.

If one of these patterns is found, it will be represented in the chart by an edge (or edges) that the triggered routine has created. This edge will start at the scanned position and end at the position just following the last word in the identified pattern. To signal its success, the routine returns this ending position. If it has failed—if the targeted pattern is not present in the text at this point—then it returns `nil`.

The coordinating process that initiated the check for word-triggered actions checks the value returned. If it is `nil` then it proceeds with the further processing on the word at the current position as described above. If the returned value is some position object, then that same processing sequence occurs, but now with respect to the word at the position the routine has returned rather than the word originally scanned, i.e. this new word will now be passed on to the completion hook, its preterminal edges introduced, etc.

The import of this jump from the triggering word to the word after the returned position is that the rest of the normal action sequence on the triggering word and on the words at any of the intervening positions do not occur. These words are not passed to the completion hook; they do not introduce any edges of their own; and they are not themselves allowed to trigger any actions. It is as though they were not there from the point of view of actions at this level.

This protocol has been chosen carefully. Its purpose is to permit successful triggered actions to dictate what the lowest level of the chart looks like to later processes. these actions thus can pre-empt any alternative analyses that might have been made of the word sequences they define.

One can conceive of this protocol as moving to the chart level the analysis of idiosyncratic phrases that in other systems often takes place within the tokenizer. In Sparser, the tokenizer uses a conservative analysis protocol, leaving the identification of multi-token patterns to chart-level processes. This gives the grammar writer the flexibility to use phrase structure rules, transition nets, or arbitrary code for formulating their patterns, as their preferences and the kinds of information involved dictate.

6.6 Introducing edges over words

The introduction of edges for a word is an internal operation that is carefully synchronized with the other operations over positions and words (§6.5), and it is important to understand the issues involved. These are

(a) the relationship between how a word is mentioned in a rule and what edges are introduced for it;

(b) how a word's capitalization in a rule affects what happens when a particular instance of the word appears in the text; and

(c) what happens to 'unknown' words and what alternative policies there are for handling these new words that aren't included in the vocabulary established by the grammar.

6.6.1 Dependence on how a word appears in a rule

Words can appear in the righthand side of a rewrite rule (the statement of the rule's immediate constituents) in any of three ways. A given word can appear in any of three ways in an unlimited number of rules.

(1) As one of the words in a polyword. For example these single letters and periods or the parentheses and "c"—multiple tokens within a pair of quotation marks.

```
inc-term -> "G.m.b.H."

copyright-notice -> copyright "(c)"
```

(2) As one of several terms (labels) in the rule. For example the word "the"—a single token (word) within quotation marks.

```
NP -> "the" N-bar

to-company -> "to" company.
```

(3) As a so-called 'single-term' rule, where one token within quotation marks is the sole term on the righthand side of the rule

```
inc-term -> "Incorporated"

 month -> "December".
```

Polywords have their own special treatment, see §3.2.

Words that occur as one of several terms in a rule are called 'literals'. A word that appears as a literal will introduce only one edge when it is scanned, regardless of how many rules may refer to it. The label on this edge is the word itself. The edge's rule field will contain the keyword `:literal-in-a-rule`; its referent field will contain the word; and its form field will contain `nil`.

Single-term rules—where the righthand side consists of a single word—introduce edges just like rules with multiple terms. The label of the edge is the lefthand side label designated by the rule, and the other fields are also whatever the rule dictates. The only thing to point out is that a given word can be the righthand side of an any number of single-term rules, each one presumably corresponding to a different sense of the word (reflected in the rule's referent) or to an alternative subcategorization pattern[2] (reflected in the set of rules that mention the rule's lefthand-side). A edge will be introduced for each rule in which the word is the single term.

Given these protocols, it is entirely likely, even commonplace, that a word will introduce several edges. When this happens, the order in which these edges appear in their positions' edge vectors (§5.3) will be arbitrary and unpredictable—none of them will be designated the 'top-node'. Instead, the edge vectors' top-node field will contain the keyword `:multiple-initial-edges`.

---

[2] An example is the word "*retire*", which can be both a verb: "*He retired early*", and an adjective: "*the retired Admiral*".

6.6.2 Capitalized words

Words are capitalized for many reasons. They indicate proper names, the beginning of a sentence, headlines, book and movies titles, etc. Sometimes the fact that a given instance of a word is capitalized is significant in that it changes what the word means or how it can combine with its neighbors, at other times capitalization changes nothing that the grammar cares about and the word might as well have been given in lowercase.

In any event, the identity of a word is not changed by its orthographic presentation, even if some aspects of how it is processed do change. For this reason, Sparser's tokenizer separates the identity of words from their capitalization properties, as described at the data-structure level in §3.1.1. Here we describe the relationship between how a word is capitalized in a rule and what edges are introduced for it.

The simplest possibility is that a word *only* ever appears in rules in lowercase. This is taken as indicating that its capitalization does not matter and that it should be ignored when an instance of the word is seen by the tokenizer and its rules looked up and edges introduced. For example, if we have the rule, `company -> "the" company` and there are no rules that mention "the" in anything other than lowercase, then an edge for this literal use of "the" will be introduced even if what the tokenizer sees is "The", "THE", or even "tHE"—the capitalization of the instance is ignored.

If you do not want capitalization to matter, you should always use the lowercase version of a word when you refer to it in a rule. If you use anything else—capitalizing the word's first letter or using full-caps or some pattern of mixed capitalization—then the word will only introduce edges when that particular capitalization pattern is what the tokenizer sees. You could, for example, define one rule for when a title like "president" appears before a person's name and is normally capitalized, and another for when it appears in other positions and is invariably given in lowercase, e.g. `title -> "president", preposed-title -> "President".`

When you do use a word in a rule in such a 'marked' version of its capitalization, you should be careful to define all the variants you expect to need. This is because the protocol used for introducing edges over words requires an exact[3] match in the kind of capitalization once any capitalized versions of a word have been used. If, for example, you defined a rule using the capitalized form "Boeing" (say as part of defining the name of that aerospace company), and you did not also use the lowercase version "boeing" in some rule, then no edges would be introduced when an instance of "boeing" was seen in a text.

[3] Note from §5.3.2 that the tokenizer does not presently distinguish between any of the various mixed-case possibilities for a word, lumping them all together as a single kind. This means that if for some reason you defined a rule with the word "NeXT" and the text read "NExT", your rule would still complete even though the pattern isn't exactly what you specified.

6.6.3 Unknown words

No grammar is going to have predefined all the words that might occur in a real text from an unfiltered source such as tomorrow's edition of the Wall Street Journal. Sparser is designed to make it easy to cope with this fact in a disciplined way by setting the 'unknown word policy' to one of several possibilities. The protocol one selects will determine what happens when the Tokenizer encounters a word that has not already been defined in the grammar, and what happens to that word when the time comes to introduce edges.

The policy is set by passing one of three specific keywords to the following function. This is done as part of initializing a session, and can be redone in the midst of a session or even in the middle of a run.

`what-to-do-with-unknown-words ( keyword )` `function`

> Takes one of the three keywords described below and sets the unknown word policy accordingly. The policy is manifest by the value of the function Establish-unknown-word, which is set to a particular predefined function value associated with the keyword.

The policy currently in effect can be obtained from the value of the global symbol `*unknown-word-policy*`, whose value will be the keyword most recently passed to what-to-do-with-unknown-words.

`:ignore` `keyword for the unknown-word policy`

The simplest thing one can do is to 'ignore' all words that were not explicitly defined in the grammar. Under this policy, whenever the tokenizer identifies a alphabetic or a digit sequence token that does not already correspond to a word object (operationally, this is any non-punctuation token that does not pick out a symbol in the `word` package with that same character sequence), it will always return the same word object—one that was expressly created for the purpose of representing all unknown words, namely

`*the-unknown-word*` `symbol (defparameter)`

> This symbol is bound at system-creation time to a word object whose symbol field contains `word::unknown-word`, and whose pname field contains the null string. This word object is what the tokenizer returns whenever it encounters a undefined word and the unknown word policy is `:ignore`.

The effect of this policy is to map all unknown words into the same word object, which is entered into the chart and undergoes word-level actions just like any other. However, since this canonical representation for unknown words does not come with a substantive rule-set, it consequently will not introduce any edges or initiate any polywords. Users can, at their option, assign this word a completion action (for example to count how many of the words in a given text are outside their grammar), or have it trigger a user-defined word-level parsing routine. By design there is no way to reference this word in a rule.

It can be important, even necessary, to use this protocol when running continuously over a unlimited stream of texts (e.g. day after day of several online news sources). If you adopt one of the other protocols (below), new word objects will continually be being defined, one for each new word, and these objects take up space in your Lisp image. If you are running a Lisp system with a predefined upper limit to the size of its heap (as on a Macintosh without virtual memory), new words can accumulate to the point where you run out of space and your Lisp process crashes.

On the other hand, with suitable space allocations or when working with anticipated quantities of text per session, it can be more useful to have new, normal word objects constructed for any unknown words as they are first seen, and to handle them in the grammar heuristically according to their orthographic or morphological properties. Once such a new word object has been defined automatically by the tokenizer in this way, it will, of course, be returned for all subsequent instances of the same (case-sensitive) character sequence. To this end two other unknown-word policies have been defined, differing only in whether or not the morphological properties of the word are noted.

`:capitalization-&-digits` keyword for the unknown-word policy

Under this policy and the next, word objects are automatically created for any unknown words that are encountered by the tokenizer. The effect is almost the same as if the previously unseen string of characters had been passed to define-word (§3.1.2): A word object is created with that string as its `pname`, along with a corresponding special symbol in the `word` package.

At any latter moment in the current session, if that the same string of characters is seen—regardless of the capitalization it may have in that instance—the newly created word object will be returned by the tokenizer. The object will happen to record the capitalization pattern of the first instance, but it is defined under the assumption that its capitalization pattern does not matter and that any capitalization variant of the original should be treated as another instance of the same word.

Note that no permanent, off-line record of this new word will be made: its data structures will not be saved beyond the duration of the current session; for that one would need to explicitly define the words in a file that was loaded as part of one's own grammar.

When a digit sequence is seen (e.g. "194"), not only will a word object be defined, but its value as a positive integer will be calculated and saved with the word object in its `plist` field with the tag `:numerical-value`.

`:capitalization-digits-&-morphology` keyword for unknown-word policy

This unknown word policy is identical in behavior to the last one, with the addition that for new alphabetic character sequences (as is done for any word defined normally by define-word), a simple morphological affix-recognizing routine is run, and a keyword is entered in the word object's `morphology` field if the last characters of the word are "s", "ed", or "ing"; see §3.1.4.

6.6.4 Digits

If the unknown word policy is set so as to accept otherwise undefined sequences of digits, and if the flag below is not `nil`, then whenever such a digit sequence is scanned an edge will be introduced over it with the category "digit-sequence".

`*make-edges-for-unknown-words-from-their-properties*` `flag`

A flag checked as part of the process of introducing edges over terminals. Its default value is `t`.

Note that this is a default edge. If a digit sequence is explicitly defined in one's grammar, say the sequence "1993", then the edge (if any) introduced by that known word will be added to the chart and the default "digit-sequence" edge will not be.

6.7 Chart-driven processing

Sparser's standard mode of operation is to perform its analysis in one pass, scanning each word of the source in sequence from the beginning of the source to its end. Once a word has been scanned and entered into the chart at the next position in sequence (§6.4.2), and after all of the word level actions triggered by that word have been taken (§7.2), all of the subsequent processing, up to the moment when the next word of the text is scanned, takes place at what we can call the "chart level".

At the level of the chart a user can adopt any of several alternative parsing protocols. In the current version of Sparser these are 'just terminals', 'all edges', and 'top edges only'.

Which protocol is deployed in a given run is adjudicated by the function chart-based-analysis just after the initializations are done at the beginning of a run. When this function begins, it accesses a global symbol: `*kind-of-chart-processing-to-do*`, and dispatches from it to one of the three chart-level protocols described below.

Since the choice of protocol is parameterized like this, one can freely change it between runs just by changing the value of the global by using the function given just below. Note, however, that the change will not automatically turn on/off any rules that are intended only for one protocol rather than another.

The protocol can also be changed at any time *during* the parsing process. This can be useful when the article has very different kinds of section within it; for instance you might want to skip over the header portion of an online news service article or use a different kind of operations. Making the change during a run takes two steps, and would be initiated at one of Sparser's hooks (§7). The steps are (1) calling the function below with the symbol that names the protocol you want to change to, just as you would if you were making the change between runs; and (2) executing a Lisp `throw` to the indicated tag.

`establish-kind-of-chart-processing-to-do (keyword)` `switched function`

The keyword argument must be one of the predefined alternatives as given below. The change will take effect with the next run of the parser or immediately upon a throw from within the parsing process to the tag below.

`change-kind-of-chart-processing` `throw tag`

Throwing to this tag will return the process to the level at which it looks up the chart processing protocol presently in effect and starts the execution of the protocol's state machine. Nothing else in the process's state is changed—the process picks up precisely where it left off.

At this level, i.e. inside the function chart-based-analysis, is also the throw tag for terminating the parsing process before it would normally finish, as described in §2.5.2.

6.7.1 Just terminals

One can run Sparser without doing any parsing—that is to say without any of the word-level or edge-based operations—just scanning the terminals of the text one chart position after another, dispatching each time to a designated function. This protocol is useful for doing such things as recording word frequency or testing what is being returned by the tokenizer and next-terminal. This protocol is deployed when the following keyword has been passed to Establish-kind-of-chart-processing-to-do:

`:just-do-terminals` `keyword naming a chart-level protocol`

Under this protocol, Sparser enters a loop where it makes calls to scan-next-position, filling successive positions of the chart, until the word returned is the one indicating the end of the source.

With each iteration, after filling the next position of the chart, a call is made to a shell function that the user should specialize. This specialization is done by executing a function that sets a shell routine to the body of a designated user function along with setting a global symbol to a keyword that names the choice of specialization one has made.

`establish-version-of-look-at-terminal` `function`
`(keyword function)`

This function `setf`'s the `symbol-function` of the shell function look-at-terminal to have the `symbol-function` value that the symbol that the 'function' argument—indicated by a symbol—has at that moment. The 'keyword' argument becomes the value of the symbol `*definition-of-look-at-terminal*`.

The function one uses must take one argument, which will be the word object just returned by Scan-next-position. If the function needs to reference other items in the chart, it will access them via the state variables that are maintained to manage the process of populating the chart with terminals as described in §6.4.

6.7.2 The 'All-Edges' parsing protocol

A version of the standard bottom-up, chart-based parsing algorithm is available by passing the following keyword to establish-kind-of-chart-processing-to-do:

```
:all-edges                          keyword naming a chart-level protocol
```

This protocol will find every edge that the grammar sanctions for a given text. Because it constructs *every* possible edge, the chart can end up including edges that some other protocols would not have constructed (e.g. Earley's algorithm or Sparser's top-edges only protocol), and which the user will typically want to ignore.

These unnecessary edges may reflect redundant analyses, where there are several edges with the same label and same set of constituents for a given span of text. These edges will reflect alternative orderings of how the daughter constituents were composed and are often called *spurious edges* because they reflect uninteresting differences in the organization of the whole constituent tree. There can also be *stranded edges* that are formed over local spans of the text but which will not make sense for the text as a whole, and so will not be linked into any larger edges. These are the standard potential problems of bottom-up parsing protocols, and Sparser's version is no different.

Given this property of the protocol, then for any non-trivial grammar there will be several ostensibly valid analyses for any text, and it is the user's responsibility to select from the edges in the chart the ones she wants to use for later (user) processes. The usual criteria is to find the successive topmost edges that span the greatest amount of the text and to then use the trees of edges that descend from these.

Under the all-edges protocol, every edge that is formed is a correct, local analysis of the text it spans, given the current grammar. Edges are checked for and formed continuously as the terminals of the text are entered into the chart; the algorithm does not, for example, first populate the entire chart with terminals and only then go back and start introducing edges in successive layers. Instead it is an incremental process: At the point in the algorithm where it is about to scan the next position, N, it will have constructed all of the edges that will ever span any pair of positions between 0 and N.

The edge-forming process is interleaved with the process of carrying out word-level actions (§6.5). Each increment of checking for possible compositions and constructing those edges that the grammar allows takes place after all of the word-level actions have been taken on the word just scanned. This is after the edges that immediately dominate the word have been introduced (§6.6) and after any edge-driven user-routines have been executed. If no edge(s) were introduced over the just-scanned word, then no checking for larger edges will occur during that increment.

When an edge of any length is created, two things happen after it is entered into the chart. (1) It is entered into a queue, and (2) a check is made for grammar rules that have the label of the new edge as the sole term on their righthand sides. If any such single-term rules are found, additional edges corresponding to each of those rules are created.

According to the all edges protocol, each edge that is created must be checked for possible composition with all of its neighbor edges adjacent to it to its left and to its right. Whenever there is a rule in the grammar that combines the edge with one of those on its left (right), then the corresponding edge is created and entered into the chart. The checks

are made in the order in which the edges were created by taking them from the queue in FIFO order ("first in, first out").

During an increment of edge checking, each new edge that is created will itself also be checked, which may in turn lead to the introduction of further edges and thereby extend the duration of the increment until all the newly introduced edges have been checked and no additional edges added.

Note that each increment is initiated by the introduction of one or more edges over what is at that moment the rightmost word in the chart. Those edges will be checked against neighbor edges to their left, but any neighbors they may come to have to their right do not yet exist, and in fact cannot exist until the scan proceeds further. This is not a problem. If we have two adjacent edges all that matters is that the pair be checked, not which edge prompts the check. At the time an edge is introduced, all of the edges formed by rules that end at the position where it begins (its left neighbors) will already be in place. If the new edge ends at some position to the left of the position just scanned (a position with a lower token index) then some, but probably not all of its right neighbors will be in place and can be checked immediately.

The all-edges parsing protocol introduces every edge that results from successful checks to the left. When checking to the right, however, it first determines whether any potential new edge— viewed as the combination of two specific edges, not two specific labels—has already been introduced into the chart, which could have occurred when the right edge of the pair looked to its left. Thus any new edge that would be introduced by a given pair of edges is guaranteed to be added to the chart only once, since the only time at which it would be considered for introduction is when a check is being made by one of its daughters, and only one of those checks will go through without a redundancy check first being made.

6.8 The 'top-edges only' protocol

The 'top-edges only' parsing protocol is used in conjunction with Sparser's phrasal segment bracketing rules and other advanced, grammar-specific protocols. An overview of this suite of algorithms can be found in the paper by David McDonald in the 1992 ACL "Applications of Natural Language Processing" meeting, in Trento Italy, available from the ACL. It's central feature, as compared with the all-edge algorithm is that when a new edge is introduced only the 'topmost' of its neighbors—the one most recently introduced—will ever be checked. As a result, the top-edges protocol will only ever introduce a fraction of those edges that might be introduced by the all-edges protocol.

To avoid missing some desired edge, the top-edge algoritm is only used in conjunction with a suite of companion algorithms in a carefully synchronized protocol.

…

6.8.1 Segments

Inserting segment brackets

Scan with brackets

Parse within segments

6.8.2 Forest Level

Continue

Parse over treetops

Quiescence

6.8.3 Heuristics

CA