5. The Chart

This description of Sparser's chart data structure is provided so that you can prepare code—to be run from Sparser's hooks or after an analysis has finished—that can walk through the chart and examine or display the words and edges it contains. Details not pertinent to these uses have been omitted, as they are likely to change between releases.

The notion of a 'chart' and its use in bottom-up parsing algorithms was developed by Martin Kay in the 1960s. It is closely related to the structures used in the CKY bottom-up algorithm and to the well-formed-substring table used in some top-down algorithms. In its essence, a chart consists of a set of indexed 'positions' that are connected by some number of 'edges' as dictated by the grammar and the analysis process. Positions identify the locations of the words in the input text (the terminals); edges correspond to the phrases (terminal and non-terminal nodes) that span contiguous sequences of words as the result of the successful application of phrase structure rules.

Sparser's chart differs in the details of its design from Kay's original design and from many of the myriad versions of charts developed since then. Its design is motivated in large part by the need to analyze texts of arbitrary length, e.g. tens of thousands of words, not just single sentences, which leads to the need to *recycle* the objects in the chart as described in §5.5. Some of the differences include having a fixed set of position indexes rather than the dynamically modifiable 'vertexes' of Kay's original. As usual, there is a position index between every word in the text, but no additional indexes can be defined in the course of an analysis. Because of this predictable stability, Sparser can use position objects to encode a great deal of information beyond simple indexes (see §5.2).

Like positions, the edges in Sparser's chart (§5.4) are first class objects (Lisp structures) that will be recycled in the course of analyzing a long text. From an edge one can reconstruct the tree of daughter edges below it, see the rule that was responsible for its creation, and track a user-supplied 'referent', or denotation, for the configuration of edges and terminal words that the edge dominates.

To make edges and positions easier to recycle and to systematize the relationships among the succession of edges that are linked to a given position, we have reified the connection between edges and positions in an object that we call an 'edge vector' (§5.3). Besides providing the recycleable data structure that implements the link between edges and the positions at which they begin and end, edge vectors record information about the number and derivation order of edges at a given position, and expedite the operation of the parsing algorithm by precomputing the most frequently needed information.

5.1 The data structure for the chart

As a data structure, the chart is a Lisp array of structures of type position. It is normally only created once at the beginning of a session, and the data in the positions is automatically re-initialized at the beginning of each run of the analysis routine (§6.1.2).

As it is used, the chart is effectively, though not literally, a circular array, making it possible to think of it as supplying an unbounded number of positions and with that the ability to analyze texts of arbitrary lengths. We will talk about the chart as containing

some number of terminals or "tokens" corresponding to the number of words in the text or the number scanned so far at some given point in the incremental left to right parsing process. Various access functions, described below in §5.2.2, maintain a virtual threading of the positions in the array such that the position stored in the last cell of the array is followed immediately by the position stored in the first cell. Like any Lisp array, the first position in the array has index zero.

The chart array is available via this symbol:

`*the-chart*` `symbol (defvar)`

> The value of this symbol is a one dimensional Common Lisp 'general array', with each cell filled by a position object. The array may be freely examined, but it should not be modified by the user, as Sparser's initialization routines keep pointers into the array to show where they should do their cleanup.

The length of the array, and with than the maximum number of words that an edge can span, is determined by this parameter.

`*number-of-positions-in-the-chart*` `symbol (defvar)`

> The value of this symbol is a positive integer. It is set to a default value as part of the configuration established when Sparser is loaded, typically 500; see §6.1.1.

The length of the chart is fixed at the time it is created. Should it be necessary to change the length, a new array must be created. To do that, change the value of the 'number of positions' parameter to the desired value, and set the symbol `*the-chart*` to the value of the function `make-a-chart` called with no arguments.

It is not possible to maintain more than one chart at a time, since part of using Sparser's chart is maintaining a set of state variables that are used by the parsing routines, and only one set of these variables is available.

5.1.1 Extracting information from the chart

Sparser is executed for side-effects rather than for a return value; in particular it does not return the 'root node' of the syntactic analysis that it makes (as sentence-based parsers for question answering systems can do). This is because Sparser is designed for the analysis of actual texts of unlimited length (tens of thousands of words) that are written by people on unrestricted topics. In such situations the notion of a single return value does not make sense—what, for example, would be the syntactic root node of a multi-paragraph news article? Indeed, given the syntactic complexity and diversity of content of real texts and the weaknesses of today's grammars it is often impossible even to identify full sentences.

The principal side effect of Sparser's operation is populating the chart with the words of the text and edges formed over them by the application of the current grammar. These may be accessed via their chart positions or displayed by the built-in routines described in §8.1.

The objects in the chart retain their values until they are initialized at the beginning of the next call to analyze a text (§6.1.2). Users are free to write their own routines to examine the chart and extract from it by employing the fields and routines described in this section.

When working with long texts, however, it is important to appreciate the implications of how the chart recycles its structures (see §5.5 for details). A given position or edge object may have its field values recycled several times depending on the size of the resource sets and the length of the text. Consequently it can be important to extract any needed information stored in positions or edges shortly after it becomes available, since it could be lost if one waited until the entire analysis was finished. To facilitate this, a set of hooks into the analysis process are provided (see §7) at which user-defined actions can be triggered and the desired information extracted and stored in the user's own data structures. This stream of triggered actions is Sparser's other important side effect.

5.2 Positions

In Sparser, position objects have been given additional representational duties beyond their standard role as vertices indicating where edges begin and end. Positions provide a record the words in the text and the whitespace between them. They maintain the information about a word's capitalization in each instance of its occurrence, providing a representation of the word as a particular 'token', to supplement the information stored permanently with the word object when it is thought of as a 'type'.

Positions support a relative index that allows a word to be accessed by its location in the chart counted in the number of words from the beginning of the text, and an absolute index with respect to their location within the fixed chart array.

Positions keep pointers to the location in the character buffers (§6.2) where words start so that one can manipulate the actual characters that comprised them.

Positions also maintain information about the state of analysis of the word they record. This information is used to control the word-level processing of Sparser's analysis routines, and should be maintained by any user routines operating at that level to ensure that their actions are synchronized with those of the regular processes. See §6.4 and §6.5 for particulars.

Positions are conceptualized as being located *between* the words of the text. An edge thus starts 'before' one word and ends 'after' another (or the same) word further on in the text. However words do not have an independent status in the data structures provided by Sparser's chart; instead they pointed to from the position objects: Each position points to ('stores in one of its fields') the word object that comes after that position in the succession of words in the text. This is reflected in the printing conventions for positions, described just below.

As a data structure, a position object is a Common Lisp 'structure' of type `position`. Positions have nine fields as indicated by the access functions listed below. A position object prints as, e.g.

```
#<position1 1251 "in">
```

The number appended to the symbol "position" gives the permanent location of the position object in the chart array (1). The number set off by itself (1251) gives the object's token index: This position holds the 1,251st word of the text, in this case the word "in". This word falls between positions 1,251 and 1,252. The position just before this one will be stored in array cell zero of the chart and have the token index 1,250. Position 1,249 will be the position object stored in the last cell of the chart array, whose array index will be the number `*number-of-positions-in-the-chart*` minus one. (In this example the chart array was defined to hold two hundred and fifty position objects.)

5.2.1 The fields in a position object

The nine fields of a position object are accessed by the functions listed below. The field values for a position are established at the moment that that position is reached as the parsing process scans successive words (§6.4.2). If the text is long enough to recycle the chart to reach the same position object again, the initial values will be replaced with the values for the new word; this can happen any number of times depending on the length of the text and of the chart array.

The position objects are *not* re-initialized at the start of a run. They each retain the field values that they most recently held until the are again reached by the scanning process. This means that you must be careful in examining positions between runs and use the state variables given in §5.6 to determine which position objects are germane to the run being examined and how the cells of the chart array correspond to the token positions of the parse.

The access functions below all take one argument, a structure of type `position`, indicated here as "p".

`pos-array-index (p)` — `structure access function`

Returns an integer, which is the fixed index to the cell that stores the position object in the chart array, and acts as a backpointer to that cell. This index is used to determine what positions lie before or after the position in the chart, rather maintaining direct pointers to these positions in the manner of a linked list. The number ranges from zero to the assigned length of the chart minus one.

`pos-character-index (p)` — `structure access function`

Returns an integer, which is the location of the first character of this position's terminal in the current input string or file as mirrored by the character buffers (§6.2). It is interpreted as an offset from the beginning of that string or of the file seen as a character stream. The first character of the input will have character index number one.

`pos-token-index (p)` — `structure access function`

Returns a number, which is the index of the position with respect to the terminals (tokens) of the text. It is the customary index for parsing and is the one reported in traces. The number starts at zero and increases indefinitely.

`pos-ends-here (p)` `structure access function`

Returns an edge-vector object. This object records all the edges that end just before the word stored at this position.

`pos-starts-here (p)` `structure access function`

Returns an edge-vector object. This object records all the edges that start just before the word stored at this position.

`pos-terminal (p)` `structure access function`

Returns a word object. This word is the one located 'between' this position and the chart position just after it.

`pos-preceding-whitespace (p)` `structure access function`

If there was whitespace between the terminal at this position and the terminal at the previous position, this function returns the word object that represents that whitespace, for example the object `#<word one-space>`. If there was no whitespace between them it returns `nil`—for example if the word at this position is the number "10" in the string "$10". If it happened that there was more than one word marked as whitespace between the two terminals (e.g. a newline followed by indentation spaces), then only the final one appears here.

`pos-capitalization (p)` `structure access function`

Returns a keyword that represents the case pattern of the word at this position. The word itself (the value of `pos-terminal`) will be in its canonical form. This field holds the capitalization information for this instance of the word.

The possible values of this field are: `:lower-case`, `:initial-letter-capitalized`, `:all-caps`, `:single-capitalized-letter`, `:mixed-case`, `:digits`, `:spaces`, or `:punctuation`.

`pos-assessed? (p)` `structure access function`

Returns one of a number of designated keyword symbols or `nil`. The set of possible keywords depends on the mode in which Sparser is being run as discussed in §6.7. The keywords and the stages they indicate are described in §6.5.1.

5.2.2 Operations over positions

A number of functions are provided to access positions by their array or token indexes. These functions are useful because they incorporate the machinery needed map between token indexes, which view the chart as an arbitrarily long sequence of positions,

and array indexes, which determine which position object represents that token as the array is recycled in the course of analyzing a long text.

`position# (number)` function

Returns the position stored in the corresponding cell of the chart array. The number must be an integer between zero and one minus `*number-of-positions-in-the-chart*`. For short texts this array index is identical to the token index, which makes this a convenient function to use to look at particular instances of words and the information kept about them on position objects.

`chart-position (number)` function

Returns the position object corresponding to the position with that number in the chart seen as a token counting from the beginning of the text—the 'token index'. It takes into account the possibility that the chart array has been recycled and that token indexes and array indexes no longer have a simple correspondence. Signals an error if the desired position is no longer available, i.e. if the array has been recycled and the position object that originally represented that token position in the chart has been recycled to represent a later token position.

`still-in-the-chart (number)` function

A predicate that returns `t` if the position with that token index is still represented in the chart array or `nil` if the position object originally representing it has been recycled.

`chart-position-before (position)` function

Takes a position object as input and returns the position object whose token index is one less.

`chart-position-after (position)` function

Takes a position object as input and returns the position object whose token index is one higher.

`word-after (p)` function

Returns the word that follows the position. This is the word that the position stores.

`word-before (p)` function

Returns the word associated with the chart position before this position.

`position-precedes (earlier later)` function

A predicate that returns t if the first argument position appears earlier in the chart than the second, otherwise nil.

`words-between (earlier later)` `function`

Returns a list in order of the words between its two arguments, starting with the word after the first argument position and ending with the word before the second.

5.3 Edge-vectors

The record of what edges start or end at a given position is stored in the two "edge-vector" objects that each position has, one each in its "starts-here" and "ends-here" fields as indicated above. The packaging provided by edge vectors makes for a more efficient implementation of the parsing algorithms by giving immediate access to the most relevant information as a single unit .

Besides listing the edges that start or end at a given position, an edge vector keeps track of the number of edges at the position, which edge is on the "top" (see below), and, for some versions of Sparser, what "boundary" occurs at that position.

Edge vectors are implemented as Lisp structures of type `edge-vector`. Each position object in the chart is assigned two edge vector objects at the time the chart is initialized. One is designated to hold the edges that end at the position, i.e. any edge that starts at some earlier position and ends at this one, and the other to hold the edges that start at this position and end at some other position later in the chart. Any edge vector will thus be either a "starting" or "ending" vector accordingly, which is reflected in its print form, e.g.

```
#<edges starting at 1251>
```

where the number indicates the token index of the position the edge vector belongs to. (Note that this number will change as the chart array is recycled in the course of analyzing long texts. The edge vector objects are uniquely associated with individual position objects, which in turn are ultimately individuated by their location in the cells of the chart array. As the chart recycles, a given position object will come to represent a succession a chart terminals, each at a new token index. Consequently if one were to store a particular edge vector object and then to examine its print form at different times, the form would change to reflect the token index that its position object has at the time it is printed.)

Edge vectors are created as part of creating their positions in the chart. Like positions they are re-initialized incrementally as the scan operation of the parsing process moves through the text.

While all edge vectors have the same six fields as described with their access functions below, there are two different mechanisms that they can use for storing edges. Which mechanism is used is a matter of the parameter settings in force when a session is initialized; the settings are established in conjunction with the choice of parsing algorithm (§6.7). Which mechanism is operating can be determined from the value of this symbol:

`*edge-vector-type*` `symbol (defparameter)`

This symbol is set by a switch-setting (§6.1.2) and will have one of two keywords as its value, either `:kons-list`, or `:vector`.

The 'kcons list' mechanism allows an arbitrarily large number of edges to be stored in an edge vector, while remaining storage-efficient through the use of a recycled resource of Lisp cons cells. The alternative mechanism, 'vector', uses a fixed length array to store edges, with the advantage of faster access and initialization at the cost of having a strict limit on the number of edges each position can have. The edges appear in a different order in the two cases, which must be taken into account when iterating over them as described below.

5.3.1 The fields of an edge-vector

`ev-edge-vector (edge-vector)` `structure access function`

The object returned by this function depends on the value of `*edge-vector-type*`. If the value is `:kons-list`, then the object is a list. Each of the items on the list is an edge that starts or ends at the edge vector's position as noted in the 'direction' field. The length of the list is given in the edge vector's 'number-of-edges' field. The edges are ordered on the list from most recent (the 'top-node') to the earliest, according to when the edges were created.

If the value is `:vector`, then the object is a general array of length `*maximum-number-of-edges-in-an-edge-vector*`. Edges occupy cells in the array in the order that they were created starting at cell zero for the earliest edge. Cells are then populated in order up to one minus the number of edges. The rest of the cells will contain `nil`.

`ev-top-node (edge-vector)` `structure access function`

Returns the edge most recently added to the start/end of the position or `nil` if there are no edges.

`ev-number-of-edges (edge-vector)` `structure access function`

Returns an integer corresponding to the number of edges that start/end at that position or zero if there are no such edges.

`ev-boundary (edge-vector)` `structure access function`

If the version of Sparser in use supports 'segment boundaries', this field will contain the bracket object, if any, that applies to the start/end of this position.

`ev-position (edge-vector)` `structure access function`

Returns the position this vector is associated with. This field serves as the backpointer for the vector.

`ev-direction (edge-vector)` `structure access function`

Returns one of the two keywords `:|starting at|` or `:|ending at|`. This field is effectively a backpointer indicating the use to which the vector is being

put. Note however that edge vectors are manipulated by the parser according to the field of the position object that contains them, either 'ends-here' or 'starts-here', and not because of the value in this field.

5.4 Edges

Along with the terminal words of the text, the edges introduced by the parsing process are the most important structures in the chart. Each edge represents the identification of a pattern of immediate constituents defined by some phrase structure rule. Precisely what edges will be formed is a consequence of both what rules have been defined and the parsing protocol presently in use (§6.7).

In Sparser we use the term "edge" for these structures because it is traditional when working with a chart parser. The terms "constituent", "non-terminal node", "parse node" and others mean essentially the same thing. Using "edge" emphasizes the fact that the constituent represented by the edge spans a sequence of words of the text between two vertices ("positions") in the chart.

Unlike the parse nodes in some kinds of parsing protocols, such as recursive-descent parsers or active-edge chart algorithms, *all* of the edges in Sparser's chart represent constituents (completed rules) that have actually been found in the text, not just hypothesized. (The analog in recursive-descent parsing to a chart of such edges is the well-formed substring table.) The one exception to this is the edges for dotted rules as discussed in §4.1.5.

Edges are introduced into the chart as the result of the successful completion of some phrase structure rule. Edges are also introduced over individual words that appear on the righthand side of some rule or rules even though these edges do not reflect phrases or non-terminals; such edges are referred to as "literals". These edges are included for the convenience of the parsing algorithms: they allow all parsing operations to be done in terms of edge objects rather requiring a more complicated implementation in terms of both edge and word objects.

Edges are implemented as Lisp structures of type `edge`. Like positions, all edge objects are predefined at the start of a session and stored in a recycleable resource array. (Unlike positions, however, the order of the edges within this array is uninteresting, with the array just serving as a convenient way to implement the resource.) As seen by the parsing algorithms, the number of edges is unbounded, while in fact there is fixed number of edge objects that are recycled as need be.

The print form of an edge object depends upon whether it is currently installed within the chart. If it is not, it will be marked as inactive, e.g.

```
#<edge42 inactive, resource 42>
```

The number, 42, indicates which cell of the edge resource array the edge is stored in. If the edge object is currently being used, i.e. it has been installed in chart to span some number of adjacent immediate constituents as the result of the completion of some rule, its print form will reflect this, e.g.

```
#<edge42  102 date 106>
```

The number appended to the symbol "edge" indicates the cell that contains the edge in the resource array. The other two numbers give the token indexes of the positions where the edge starts and ends respectively. The term between them is the label on the edge, corresponding to the lefthand side of the rule that led to the edge. Categories are indicated by symbols as in this example; if the label is a word or a polyword it will appear within quotation marks.

One typically accesses edges by way of the positions (edge vectors) that anchor them into the chart. However any edge can be accessed directly from the resource array using the index of its cell in the resource as given in its print form:

`edge# (integer)` `function`

Returns the edges object in the indicated cell of the edge resource array. The number must be between zero and one minus the present length of the array or an array access error will occur.

5.4.1 The fields on an edge

Edge objects have nine fields, accessed by the functions below. Since edges, like positions, will recycle during the analysis of a long text, the field values that an edge has reflect its most recent deployment in the chart, and no record is kept of its earlier uses.

`edge-category (edge)` `structure access function`

Returns the edge's label, which can be either a category, a word or a polyword. This label corresponds to the lefthand-side of the phrase structure rule (cfr) whose completion led to the introduction of this edge into the chart.

`edge-referent (edge)` `structure access function`

Returns a user-defined object or `nil`; see §4.2.

`edge-starts-at (edge)` `structure access function`

Returns an edge vector. This will be the same edge vector object as is stored in the "starts-here" field of the position where the edge starts.

`edge-ends-at (edge)` `structure access function`

Returns an edge-vector. This will be the same edge vector as in the "ends-here" field of the position where the edge ends.

`edge-rule (edge)` `structure access function`

Returns a cfr object. This is the rule whose completion led to the introduction of this edge into the chart.

`edge-left-daughter (edge)` `structure access function`

Returns an edge, a word or a polyword. If the rule was binary or n-ary, this object will be the left (earlier) constituent of the pair of adjacent 'daughter' constituents that were composed to form this edge. If it was a unary rule, with a single label on its righthand side, the single constituent that the edge spans will be recorded in this field.

`edge-right-daughter (edge)` `structure access function`

Returns an edge, a word, a polyword, a keyword, or `nil`. The keyword `:single-term` indicates that the edge was formed by a unary rule. The keyword `:literal-in-a-rule` goes with the case where the "left-daughter" field contains a word, and that word was one of several labels mentioned on the righthand side of a binary or n-ary rule. Any value other than a keyword is the constituent that is the right constituent of the pair composed by the rule.

`edge-used-in (edge)` `structure access function`

Returns an edge or `nil`. If an edge, it is the parent of this edge, i.e. this edge is one of the parent's immediate constituents and will appear in the "left-daughter" or "right-daughter" field of the parent. If `nil`, then this edge has no parent dominating it in the chart. Such edges are referred to as 'treetops'. An edge can have only one parent.

`edge-position-in-resource-array (edge)` `structure access function`

Returns an integer between zero and one minus `*length-of-edge-resource*`. This number is the index of this edge object in the edge resource array and indicates the cell in which it is stored.

In addition to these access functions, there are functions that compute useful combinations of edge and position information:

`pos-edge-starts-at (edge)` `function`

Returns the position that the edge starts at. Equivalent to composing Ev-position and edge-starts-at.

`pos-edge-ends-at (edge)` `function`

Returns the position that the edge ends at. Equivalent to composing Ev-position and edge-ends-at.

5.4.2 Constructing an edge

The user has the option to write arbitrary Lisp code to augment Sparser's operations at any of the designated 'hooks' (§7). If the constituents identified by that code are to be integrated into the rest of Sparser's operations, they must be encoded in an edge object just like the constituents identified by phrase structure rules. This process involves getting an edge object from the edge resource, filling the relevant fields, and calling the functions that will integrate the edge into the chart and Sparser's indexes. All of these

operations are required, and they must be done with some care since the data structures they affect are pivotal for the parser's correct operation. To facilitate this care, the user is provided with a single, keyword-driven function for constructing edges. Note that this routine makes binary edges; if n-ary edges are needed they must be assembled as a tree of binary edges (see §4.1.5).

`Make-chart-edge` `function`

Returns the next available edge from the preallocated resource of edges, having entered it into the chart and filled its relevant fields according to the values supplied by the keyword arguments. N.b. all of these arguments are evaluated.

`:left-edge, :right-edge` `keyword arguments`

These arguments should evaluate to edge objects. If both are supplied, they will determine the starting and ending positions of the edge, overriding the position arguments. The new edge will start at the position where the left edge starts and end at the position where the right edge ends. The "left-daughter" and "right-daughter" fields of the new edge will be set to these edges.

`:starting-position, :ending-position` `keyword arguments`

These arguments should evaluate to position objects. They are redundant if the edge arguments are supplied, but are required if the edges are not supplied. They will dictate where the new edge is placed in the chart.

`:category` `keyword argument`

This argument should evaluate to a category or word or polyword object. The argument is required. The "category" field of the edge will be set to this value.

`:rule-name` `keyword argument`

The value of this argument is unconstrained. The "rule" field of the edge will be set to it. Its purpose is to describe to a person inspecting the chart the reason why this edge was created.

`:referent` `keyword argument`

The value of this argument is unconstrained. It is not required. If it is supplied, it will be put into the "referent" field of the edge.

5.5 Chart objects as reusable resources

Sparser's design emphasizes speed and the ability to analyze texts of unlimited size while using a fixed, relatively small amount of storage. To this end, Sparser's primary representational devices, its chart and edges, are implemented as reusable resources. The needed storage is allocated once, when Sparser is launched, and then continually reused.

The chart is preallocated a fixed number of positions according to the value of a globally bound symbol as indicated below. A typical value is 500. When the text is longer than this value, there will come a moment in the parsing process when the chart will 'wrap', and it will reuse positions that it had used at the beginning of the analysis, in the process destroying any record of what had been in those positions originally.

The same is true of edges. All the edge objects available to Sparser are preallocated and stored in an array just as in a standard Lisp system all the data cells are pre-allocated in a heap that is automatically maintained by the process of garbage collection. When more than the preallocated number of edge objects have been used, the edge resource wraps and proceeds to write over the original values in the fields of those edge objects with new values as each later edge is added to the chart.

The only difficulty that this use of resources can pose is if the grammar needs access to a position or edge that is too far back in the analysis and has been recycled because its resource has wrapped around it. How large the resources have to be to avoid this problem is an empirical issue and dictated by the grammar being used. Experience has shown that a chart array of only 250 positions will not cause any problems since this length is usually enough to analyze whole paragraphs without wrapping. The appropriate number of edges will usually be a multiple of the number of positions, based on how many analyses a given number of words is likely to average: A deterministic grammar with no lexical ambiguity will need only slightly more edges than the number of positions; a highly ambiguous grammar can need the square of that number or more if the lexical ambiguity is high.

Besides the need to keep analyses within the active region of the resources, there is also a practical difficulty if you want to peruse the chart or its edges after an analysis has completed and you need to examine the entire chart. One way around this is simply to allocate very large resources, another is to do the examinations incrementally during the analysis by placing code at one of Sparser's hooks.

Management of the resources involves a set of globally bound symbols that act as state variables. Some of these are listed elsewhere in this documentation as well, we include them again here. We will begin with what is needed to set the size of the resources.

The chart array of positions and edge resource array are both created when Sparser is launched, setting the lengths of the two arrays to the values that these symbols have at the time (see §6.1.1).

`*number-of-positions-in-the-chart*` `symbol (defparameter)`

Value is an integer. It establishes the length of the chart array, and with that how long a text can be before the chart array will wrap and destroy the record of the earliest positions (at one position per word object, which will include punctuation and any other non-whitespace tokens). The default value is 500.

`*length-of-edge-resource*` `symbol (defparameter)`

Value is an integer. Indicates how many edges can be completed before the edge resource will wrap and destroy the record of the earliest edges. The default value is 500.

The arrays themselves are kept in these symbols:

`*the-chart*` `symbol (defparameter)`

`*all-edges*` `symbol (defparameter)`

To change the size of the arrays you must construct new arrays. The appropriate Sparser functions should be used rather than Lisp's own array constructing routines so that the initial values of the objects, their ancillary objects such as edge vectors, and their interconnections are all prepared correctly.

`make-the-chart ()` `function`

Returns an array of position-objects of length *number-of-positions-in-the-chart*, and sets the symbol *the-chart* to it. The position objects that fill the cells of the array are constructed at the same time, and have their `array-index` fields set to the integer index of the cell they are put in.

`make-the-edge-resource ()` `function`

Returns an array of edge objects of length `*length-of-edge-resource*`, and sets the symbol `*all-edges*` to it. The edge objects that fill the cells of the array are constructed at the same time, and have their `position-in-resource-array` fields set to the integer index of the cell the are put in.

These variables track the state of the chart as a resource of positions and should be checked whenever you are examining positions with your own routines.

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

The value of this symbol is an integer corresponding to the index of the cell in the chart (seen as an array of positions) that will be filled by the word returned by the Next-terminal routine (§6.4.2) the next time that it is called. Its value cycles between 0 and `*number-of-positions-in-the-chart*`.

`*position-array-is-wrapped*` `symbol (defvar)`

A flag that is set to `T` at the moment that the scan through the source text has reached as many words as there are positions allocated to the chart. For instance if the number of positions in the chart has been set to 500, then when the Tokenizer returns the 501st word this flag will be set. It remains up for the rest of the run.

`*first-chart-position*` `symbol (defvar)`

The value of this symbol is an integer corresponding to the index in the chart array of the earliest position (the position object with the lowest token index) that still corresponds to a word that is 'in the chart'.

When the analysis begins, this symbol has the value 0. Once the position array (chart) wraps, its value is 2 larger than the current value of `*next-array-position-to-fill*`, recycling between 0 and `*number-of-positions-in-the-chart*`. Note that this means that once the chart has started to recycle its positions, there will be two 'empty' positions between its effective start and end positions. These two positions are in their initial state with `Nil` in every field except their array indexes.

The following variables track the state of the edge resource and should be checked whenever you are examining edges independently of their positions in the chart or when there is the possibility that the resource has wrapped and begun to recycle edges and you are not confident that the region of the chart you are examining has only valid edges.

`*edge-resource-is-wrapped*` `symbol (defvar)`

This symbol functions as a flag whose value will be `t` or `nil`. The flag is initialized to `nil`. It goes up (is set to `t`) at the first moment that one additional edge is allocated beyond the number stored in the edge resource array.

`*index-of-furthest-edge-ever-allocated*` `symbol (defvar)`

Edges are allocated in the order of the cells that store them within the edge resource array, starting at zero. Until the edge resource has wrapped, this index is a count of the number of edges that have been allocated in the course of the current run. It starts at one with the allocation of the first edge and increases until it reaches the length of the array, where it stays regardless of how many more edges are allocated.

This symbol is designed for iterating over the allocated edges. It is used at the beginning of a run to re-initialize just the edges used in the last run by making it the limit in a `Do-times` loop. In such a loop it accesses the edges in the order that they were allocated, which can be convenient for edge-driven applications of all kinds; though note that once the resource has wrapped, the strict relationship of allocation order to array order will be lost.

`*position-of-next-available-edge-in-resource*` `symbol (defvar)`

As its name suggests, this state variable gives the index (integer value) of the cell in the edge resource array that contains the next edge to be allocated. It is initialized to zero and increases by one with each allocation of an edge, up to a maximum of one minus the length of the array (`*length-of-edge-resource*`). When that limit is reached, it is reset to zero and resumes counting upwards again.

At the beginning of a run, all of the edges that were used in the last run are re-initialized: the values of all of their fields (except for their position in the resource array)

are set to `nil`. Once the resource has wrapped, edges are re-initialized at the moment they are allocated.