4. Phrase Structure Rules

The actions Sparser takes in the course of an analysis are dictated primarily by the grammar of context-free and limited context-sensitive phrase structure rules currently being used. (These are also known in the parsing literatures as 'rewrite rules' or 'productions'). The notation of phrase structure rules is described in §4.1.2, their run-time form in §4.1.3, and the routines for deleting rules in §4.5. We will cast the discussion here in terms of context-free rules; the points where context-sensitive rules differ are described in §4.3.

Phrase structure rules define patterns of labeled adjacent immediate constituents in the usual manner. The labels are either words (§3.1), polywords (§3.2), or non-terminal atomic 'categories' (§3.3). There is deliberately no provision for features or other kinds of structured categories in Sparser; these have not proven necessary and have a dramatic impact on runtime efficiency.

One can choose to include data to be the 'referent' of the constituent that a rule forms; see §4.2. The intended use for this data is to act as the denotations of the text's constituents in the model that provides the text's meaning. Typically this model is comprised of objects in the application program for which Sparser is performing the analysis.

As manipulated by Sparser, rules, words and categories are all objects implemented as Lisp 'structures'. The user can define rules either directly in terms of these objects, which is convenient when the rules are written by a program, say as part of defining a class of objects in the user's application; or the user can define rules by using an s-expression notation based on symbols and strings and the corresponding objects will be retrieved or constructed as the rule definition is executed.

Rule are uniquely defined by the labels on their left and right hand sides. Thus if you happen to evaluate the same rule definition twice you will get the same object both times.

In this section we will describe Sparser's rule objects: how they are defined, their internal representation, how they are managed, and so on. This will take us to a discussion of the more unusual aspects of Sparser's treatment of phrase structure rules, including the mechanisms used for rules with more than two terms on their righthand sides ("n-ary rules"), its notion of context-sensitive rules, and the use of referents.

4.1 Context-free phrase structure rules

In this section we will describe the 'cfr' object ('context free rule'). Besides being the mechanism for representing context free phrase structure rewrite rules, this object is also the basis for the other kinds of phrase structure rules discussed later. These are all implemented using cfr objects, but are treated differently by Sparser's operations because of special encodings.

4.1.2 Terminology

Phrase structure rules and the constituents formed from them during a parse are defined by patterns of words and non-terminal categories. When the distinction does not matter we will refer to both kinds of objects as 'labels'.

It is convenient to talk about the labels in phrase structure rules being on the 'lefthand' or 'righthand' side of the arrow used in the standard written notation for rewrite rules. In the example below, the category "month" is the rule's lefthand side and the word "December" is its righthand side. Terms in quotation marks indicate words ("December"). Terms without quotation marks indicate categories (month).

```
month -> "December"
```

Context free rules have a single label on their lefthand side. It becomes the label on the edges formed using the rule—it indicates the "parent" of the immediate constituents with the labels on the righthand side of the arrow. In Sparser, the lefthand side of a context free rule may be a word or a polyword as well as a category; having words as edge labels is a useful way to handle abbreviations and similar phenomena, since you can write rules such as

```
"Incorporated" -> "Inc" "."
```

and have the abbreviation interpreted transparently as though the full form of the word had been used.

Context free rules can have any number of labels on their righthand sides. For efficient implementation, however, rules with more than two righthand side terms are transparently expanded internally into a set of 'binary' rules with just two labels on their righthand sides. (see §4.1.5). New categories for the additional rules are created automatically as needed following a dotted-rule notation.

When the chart contains a sequence of adjacent constituents whose labels match the identity and order of the labels on the righthand side of some rule, the rule is said to be 'complete', and a new constituent is formed with those consistents as its daughters. This constituent (phrase) is called an 'edge' following the terminological conventions of chart-based parsing.

4.1.3 "CFR" Objects

CFR objects are the Sparser's representation of all types of phrase structure rules. They are implemented as Lisp defstructs of type `cfr`, and are given in a rewrite rule notation when printed, e.g.,

```
#<psr14  month -> "December">
```

The first part of this expression, the string `"psr"` followed by a number, is the name of the symbol that has been bound to this rule to make it convenient to access. These symbols are in the `*phrase-structure-rule-package*`, which has the name `rule` unless it has been changed to avoid name clash with a package in the user's system. The symbols are declared 'special' and are internal to the package (§2.2). The example rule could thus be accessed by this symbol:

```
rule::psr14
```

Rule numbering starts at one and indicates the order in which rules were defined. The term to the left of the arrow is the rule's lefthand side label; the terms to the right are its righthand side labels—its immediate constituents.

As a structure, each cfr object is comprised of eight fields that can be accessed via the following functions:

`cfr-category (cfr)` `access-function`

Returns either a category, a word, or a polyword. This object will be the label on any edges formed using this rule. It is the label on the lefthand-side when the rule is written out in "arrow" notation.

`cfr-rhs (cfr)` `access-function`

Returns a list of categories and/or words or polywords, or a single polyword. This list will have at least one element. The sequence of the terms in the list defines the pattern of ordered adjacent constituents that will trigger the completion of this rule by the parser. These are the righthand side when the rule is written out in arrow notation.

`cfr-referent (cfr)` `access-function`

Returns an object established by the user at the time the rule was defined or an expression that will be used to find or construct the referent whenever the rule is completed by the parser. See §4.2 for a description of the valid expressions.

`cfr-form (cfr)` `access-function`

This field is not valid is all versions of Sparser. It would contain a category object or `nil` as described in §4.7.

`cfr-relation (cfr)` `access-function`

This field should contain a category object or `nil`. It can be used as a user-defined description of the rule, or will be defined automatically by the rule-definition facilities distributed with some versions of Sparser.

`cfr-completion` `access-function`

This field is used to encode information that signals that the cfr object is to be interpreted as a context sensitive rule (§4.3) or a form rule (§4.7).

`cfr-plist (cfr)` `access-function`

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

`cfr-symbol (cfr)` `access-function`

Returns a symbol in the `*phrase-structure-rule*` package (§2.2). The value of this symbol is the cfr object. These symbols are gensyms based on the prefix `PSR` ("Phrase Structure Rule"). The symbols are 'special' are internal to the package and so have to be referenced with two colons, e.g. `rule::psr14`.

4.1.4 Defining context-free rules

Rules can be defined either with a macro, ‘def-cfr’, where the arguments are given as unevaluated symbols and strings that are interpreted as the form is evaluated, or with a function, ‘define-cfr’, where all the arguments will be evaluated and must return objects of the appropriate types.

Def-cfr is intended to be the more commonly used routine, since it is less elaborate to write and handles the definition or retrieval of its argument objects automatically. This rule definition form is usually given as a toplevel form in a files of rules that are all loaded as a group. The function version, Define-cfr, is provided for use within other functions when one is already manipulating the needed objects.

```
def-cfr   ( <label>   ( <one or more labels> )                    macro
              &key :referent <special list or label>
                   :form <category label>
                   :relation <category label>  )
```

Returns a cfr object. If that combination of left and righthand side labels has been defined as a cfr before, then the same object will be returned. If they are a new combination, then a new object will be constructed.

Either a symbol or a string may be used where the argument list above indicates just “label”. Symbols will be interpreted as categories, strings as words. “Category labels” are to be named by symbols.

The first argument is a single label; it will be the lefthand side of the rule. The second argument is a list of one or more labels; they will be the righthand side of the rule. What expressions can be used with the optional `:referent` keyword argument is described in section §4.2 below. The `:form` argument is available only in certain versions (§4.7). The `:relation` argument is controlled only in certain versions; in other versions it can be used for whatever purpose the user wishes.

Typical usage:

```
>  (def-cfr  month  ("December"))
>  #<psr37  MONTH -> "December">

>  (def-cfr  day-of-the-month  (month number))
>  #<psr38  DAY-OF-THE-MONTH ->  MONTH NUMBER>

>  (def-cfr  date  (day-of-the-month "," year)
>  #<psr39  DATE ->  DAY-OF-THE-MONTH COMMA YEAR>

>  (def-cfr  holiday  ("July 4th")
>  #<psr40  HOLIDAY ->  "July 4 th">
```

```
define-cfr  ( <label> ( <one or more labels> )                  function
                &key :referent <expression to be evaluated>)
                     :form <category label>
                     :relation <category label> )
```

Returns a cfr object. If that combination of left and righthand side labels has been defined as a cfr before, then the same object will be returned. If they are a new combination then a new object will be constructed.

This is the function version of the rule definition routine. Its argument pattern is identical to the macro version, except that its arguments are evaluated by Lisp before they are passed to the function. The user must take care that the argument expressions will have the desired values at the time the form is executed.

Possible usage:

```
> (define-cfr  category::month  `(,word::December)
    :referent  (user:month-named 'December))
#<psr37  MONTH -> "December"
```

Note that, as indicated by having the identical rule symbol—psr37—the function version of the rule defining the month "December" (define-cfr) yields the identical cfr object as the macro version (def-cfr). The only difference is that if the function version were used as done just above, the object would include a referent; while if the other version had been used it would not. (A reference can be included with the Def-cfr version of the rule-creating form, as described in §4.2.)

4.1.5 More than two terms on a righthand side

We will call any phrase structure rule with more than two terms (labels) on its righthand side an *n-ary rule*, in contrast with 'binary' rules containing two terms and 'unary' rules containing one. The run-time representations of n-ary rules is different from the other two kinds, and we explain the differences in this section.

One does not define n-ary rules any differently than binary or unary rules. The special arrangements for them are set up transparently when the rule-defining machinery notices that there are more than two terms on its righthand side of the rule being defined.

This special handling is required because for reasons of efficiency Sparser's rule-manipulating routines work only with binary or unary rules. Thus in order for an n-ary rule to be used, it must first be reformulated as a set of binary rules.

To illustrate this, imagine we want to define the following rule:

```
(1)  date  ->  number  "/"  number "/"  number
```

This rule has five terms on its righthand side, and as a result cannot be recognized directly by the parsing algorithm since the algorithm only considers pairs of adjacent constituents rather than a five-tuple like this. It must be reformulated in terms of binary

rules, which is done by writing a set of rules that combine the terms of original rule in successive pairs starting, by convention, from the left.

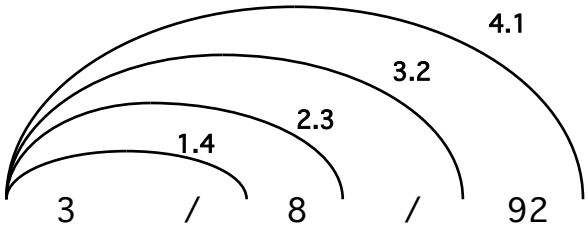
In this case, the definition of rule 1 leads to the automatic definition of four new rules, shown below. These rules correspond to the intermediate states in the acceptance of a n-ary rule in Earley's algorithm (which is also at its heart a binary algorithm), and we refer to these rules as *dotted-rules* following Earley's convention of naming the categories of the intermediate completions by placing a 'dot' between the already scanned terms to the left and those to the right which remained to be scanned. At run-time it is these four dotted-rules that take part in the parse, not the original n-ary rule.

```
(psr1/1.4)  number_/  ->  number  "/"

(psr1/2.3)  number_/_number -> number_/  number

(psr1/3.2)  number_/_number_/ -> number_/_number  "/"

(psr1/4.1)  number_/_number_/_number -> number_/_number_/  number
```

The categories that are formed for dotted rules (the labels on their lefthand sides) are named by concatenating the names of the terms to the left of the dot in the corresponding Earley state with underbar characters ("_"). These artificial names are used to minimize the possibility that they would clash with a category name that a grammar writer might choose naturally.

Rule symbols are constructed for dotted rules as indicated in parenthesis for the example rules, i.e. `psr1/1.4`. Recall that the rule symbol for the original n-ary rule consists of the characters "PSR" concatenated to a number, where the number indicates how many rules have been defined in the session so far. The rule symbol for a dotted rule starts with the symbol for the original rule (e.g. "psr1") and adds two numbers separated by a dot. The numbers reflect where the rule sits in the set: the number on the left of the dot indicates how many dotted rules in the set have already been completed when it completes, and the number to the right indicates how many constituents remain to be scanned at that point.

To see how the reformulation of a n-ary rule in terms of a set of dotted-rules operates in practice, consider the diagram below. It illustrates how the dotted rules collectively produce a left recursive tree of binary rules where their corresponding n-ary rule would apply, taking in one additional constituent to the right with each successive edge. (The edges are labeled here with the number of the rules that formed them as given above.)



Dotted-rules are represented in Sparser as standard cfr objects and participate in the parsing just like other rules. The only differences are in the information stored on their property lists. This information distinguishes the intermediate rules, in this case 1/1.4,

1/2.3, and 1/3.2, from the final rule, 1/4.1. The thre intermediate rules have the tag `:dotted-rule`. The rule whose span is identical with that of the original n-ary rule has the tag `:rolled-out-from`. The n-ary rule itself (the rule whose definition initiated the construction of the dotted rules) has the tag `:n-ary`.

`:dotted-rule` `tag in plist of a cfr`

The value of this property is a three-element list. The first element of the list is the original n-ary rule that the dotted-rule is an intermediary of. The second is a two-element list that restates the righthand side of the dotted rule. The third is a list of the terms of the original rule that remain to be scanned if that rule is to be completed.

`:rolled-out-from` `tag in plist of a cfr`

The value of this property is the n-ary rule that this dotted-rule mimics in its span of constituents and in the values of its form and referent fields. A rule with this property is the outermost of the set of binary, dotted rules formed as a result of the definition of the n-ary rule.

`:n-ary` `tag in plist of a cfr`

The value of this property is a two-element list. The first element lists the categories that make up the righthand-side of the rule. The second lists the dotted rules from from the rule with this property, listed in order from the rule with the largest span to the smallest.

While a n-ary rule does not itself take part in a parse, arrangements are made so that any side-effects that the completion of the rule could have had also take place when its maximal dotted-rule completes (e.g. 1/4.1 above). This is done by copying the relevant fields of the n-ary rule onto that dotted rule, in particular its 'referent' field.

Note that any user routine that examines the chart during or after a parse and looks for an edge that resulted from the completion of an n-ary rule will not find that rule per se. Instead, if the pattern of constituents that rule defines occur,ed the chart will contain an edge formed by its maximal dotted rule, the one marked with the property `:rolled-out-from`.

Also note that if only part of the constituent pattern of an n-ary rule completes, and there is no other regular rule that covers that same span, then the chart will contain some topmost edges ('treetops') that do not represent the completion of any of the rules in the user's grammar. On some occasions it can be valuable to see such evidence of the "debris" of the partial application of grammar rules; however it is generally the case that edges formed from dotted-rules are not syntactically or semantically significant, and consequently any user code that examines the chart should carefully note distinctions in rule types.

4.2 "Referents" in phrase structure rules

As the word is generally understood, a 'parser' analyzes how a text is organized into a hierarchy of phrases, and prepares a 'structural description' representing what it has determined. For Sparser, this description is its chart, which can be freely examined after the parse has finished. (An even narrower conception of a parser is simply as a 'recognizer' that determines whether the text is a legal sentence in the grammar. One could interpret Sparser's behavior in this way, but since Sparser assembles a chart as a required element of its processing nothing would be gained.)

This standard notion of parsing omits the companion process of determining a text's meaning, which in most language understanding systems is done as a separate process of 'semantic interpretation' that takes place after the structural analysis. This need not be the case, however, and for Sparser the two operations can be interleaved by associating a rule of semantic interpretation with every parsing rule, as described in this section.

Since Sparser is typically used with a 'semantic grammar'—one where the labels on edges are symbols (category objects) for semantic notions like date or person rather than symbols for syntactic notions such as noun phrase or verb—then the description provided by the chart can be as close to a description of the meaning in terms of categorized individuals and events as one wishes.

However, Sparser is also designed to be used with an application that maintains a data base of specific individuals (representations of particular people, particular dates, etc.). Here one wants the act of parsing a text to go beyond a description and to establish a correspondence between the phrases in the text and objects or structures in the application's data base.

To make this possible, Sparser includes a facility for including a *referent* with each phrase structure rule. This sense of referent is the traditional one from model-theoretic semantics, where an expression in a text, say the phrase "*the day after tomorrow*", is said to 'denote' some object in a model, such as the user's application's representation of a particular day in some calendar.

Referents can be objects that already exist when a rule is defined, or they can specify instructions for finding or constructing objects at parse-time in terms of the referents of the constituent edges that the rule combines.

4.2.1 Referent fields

All phrase structure rules (cfr objects) include a field named `referent`. This field is populated when the rule is defined, as dictated by an expression given as the defining form's `:referent` keyword argument as defined below. This expression is interpreted according to a syntax about to be described, and the result becomes the value of the rule's referent field.

Edges (non-terminal nodes in the chart) also have a `referent` field. This field constitutes the links between the phrase that the edge defines and the object in the user's model that the phrase is taken to denote, as dictated by the phrase structure rule that constructed the edge. The value of the field is determined by interpreting the referent

field of the rule at the time the rule completes. The field will either already point to the desired object, or will provide instructions for finding or constructing it.

As the discussion of these various elements can be confusing, we will adopt the following terminology. The term *referent field* will be used here only to refer to the field in a rule. It will mean the object or expression that results from the define-time processing of the *referent expression* that is given as the value of the *:referent keyword argument* in a rule's definition. The object that fills the referent field of an edge resulting from the rule will be referred to as the edge's *referent*. Note that the referent expression is the only thing that a user supplies; the referent field in rule and the referent of an edge are all produced by operations within Sparser.

Referents are optional. If the `:referent` argument is omitted from a rule's definition its referent field will contain `nil` and the referents of any edges made by the rule will also be `nil`.

The syntax for the referent expressions will be described in terms of the following cases, which we will take up in turn: (a) supplying the referent directly as part of a rule's definition, as opposed to (b) providing instructions for the referent's calculation, which in turn divides into cases according to whether the righthand side of the rule being defined has one, two, or more than two terms. At the end of this section we will also discuss the global variables that are maintained when the referent of an edge is being computed; these can be referred to in any user-defined function for referent calculation.

4.2.2 Supplying referents directly with the rule

Recall that phrase structure rules can be defined with either of two forms: Define-cfr, which is a function and evaluates all of its arguments, or Def-cfr, which is a macro that evaluates none of its arguments. The expression given with the `:referent` keyword argument is processed by the same routine in both cases, but for Define-cfr this routine gets the value returned after Lisp evaluates the expression you supply, and for Def-cfr it is given the expression unchanged from what you supplied.

This 'define time' processing routine treats referent expressions that are lists as instructions to be interpreted when a rule is applied (see next section), and accepts all other data types unchanged. Thus if you supply a list, the rule's referent field will consist of a variation of that list with some of its contents changed to a canonical run-time form; if you supply anything other than a list, it will go into the referent field unchanged and will become the referent of edges made with that rule.

To make this concrete, let us suppose that you have already loaded your application into the Lisp and so can access its objects at the time you are defining phrase structure rules for Sparser. In such a case you can make the referent expression be a form that returns the object you want.

For purposes of this example, suppose that the objects your application uses to represent the months of the year can be accessed with the function `user:month-named` that returns a month object given the symbol that acts as the month's name. Using the rule definition function Define-cfr, we can then associate particular month objects from the application with the rules that recognize particular months in the text as shown here.

```
(define-cfr  category::month  `(,word::December)
  :referent  (user:month-named 'December))
```

The access function is evaluated by Lisp at the time the rule form is evaluated, and its value—your object that represents the month of December—is put in the referent field of the rule. When the word "December" is scanned in the course of a parse the edge that is formed based on this rule will have your December-object as its referent.

Note the explicit package indicator on the access function and on the labels in this example. It is a reminder that one must be careful about such details when mixing Sparser and user code like this. The function symbol Define-cfr, like all other Sparser functions listed in this documentation, is only defined in `sparser` package. The category named 'month' and the word spelled "December" are here accessed via the value of the package-specific symbols that were constructed for them when they were defined in earlier expressions. If the category or word was not already defined at the time this rule form was evaluated then these symbols would be unbound and a Lisp error would occur.

The very same rule can be created using the macro Def-cfr as shown below. This form may have the advantage of a less cluttered presentation. It also will automatically create the objects for the category and word if they had not been predefined; one must be careful, however, to avoid inadvertent misspellings, which will create different objects than those intended and lead to confusion when the expected parses fail to occur.

```
(def-cfr  month  ( "December" )
  :referent  (:eval (user:month-named 'December)))
```

If a word is intended to have more than one meaning, then it should be given multiple definitions, each with its own referent.

4.2.3 Instructions for calculating referents

The direct association of a rule with a specific entity is very effective for most common nouns and other words that have what we can think of as having an 'atomic' meaning (e.g. "*two*"). Most of the rules in a grammar, however, will define patterns of composition rather than refer to already existing objects. This will entail building up denotations recursively on a foundation of existing objects or object classes through the composition of attributes, relations, and occasional re-categorizations.

To this end, a set of *referent instructions* are provided by which to define operations in referent expressions that will carry out such actions in the course of composing edges. These instructions are deployed in the referent field of a phrase structure rule as part of its definition, just as direct references to user objects are. The available instructions and the constraints on their use are described below.

A rule can have one instruction or several (or none, in which case the referent of the edge will be `nil`). All of the instructions are given together as a single list. Each instruction will consist of a keyword naming the operation followed by its arguments. Note that since keywords are used to delimit instructions, keywords cannot be used as any of the arguments. A phrase structure rule can have as many referent instructions as the user wishes. They will be executed sequentially according the order in which they appear.

The first instruction is distinguished, however, in that it determines what object is to be returned as the referent of the edge (i.e. it will determine what goes into the edge's `:referent` field). Other instructions may dictate side effects on the referent or carry out other sorts of actions on any other object accessible when an edge is completed (see below); but the other instructions cannot change what is returned as the referent. (Note that this is a difference from the default in Lisp: usually the last form in a sequence determines the return value of the whole, here it is the first.)

The arguments to an instruction can be the referents of any of the constituents given in the righthand-side of the rule (or some other global items listed below). They are referred to symbolically in a referent expression by using any of the fixed set of names, listed below, that indicate the desired constituent by its position in the rule. In a binary rule, for example, one can refer to either the `left-edge` or the `right-edge`. (Note that these symbols are in the `sparser` package.) In an n-ary rule one can refer to the `first` constituent, the `second` constituent, `third`, `fourth`, etc. up to a built-in limit of ten.

The 'function' instruction takes as one of its arguments a user function, which should be given as the symbol for the function's name (which may be in any package, though if the symbol is not in the `sparser` package its package prefix will be required). In the list below constituent arguments are given as `<daughter>` and function arguments as `<fn>`.

`:daughter <daughter>` `referent instruction`

The single required argument to this instruction is the referent of the indicated daughter (immediate constituent) of the edge being formed by the rule. That object is returned as the value of the instruction. Note that since returning the referent of one of the constituents is all that this instruction does, it can only sensibly be used as an initial instruction.

`:function <function> [<daughter>* | <symbol>*]` `referent instruction`

This instruction is the equivalent of applying the indicated function to the indicated arguments. The function may take any number of arguments, including no arguments. A warning will be given if the function (indicated by a symbol, possibly including a package designator) is not defined at the time the rule-form is evaluated.

The arguments to the function may be either (the referents of) daughter constituents to the edge indicated by one of the designated names or may be arbitrary symbols. The symbols are not evaluated at rule-definition time, but they are evaluated when the instruction is evaluated at the time the rule is completed. Consequently they can only sensibly be either global variables in one's application. As usual, if a symbol does not have a value (i.e. is unbound) a Lisp error will occur.

`:eval <lisp form>` `referent instruction`

Unlike the other two referent instructions, this instruction is applied at the time the phrase structure rule is defined rather than when the rule is completed and

an edge formed. The single argument must be a list. It will be evaluated in the environment that holds at the time the rule in which it appears is defined. The Lisp function 'eval' will applied to the list and the resulting value returned as the value of the instruction. Lisp errors in the execution of the form are not trapped.

`left-edge, left, left-daughter` names for instruction argument

These three symbols are alternative names picking out the same constituent in a binary rule, namely the first of the two constituents seen from left to right (lower numbered positions in the chart to higher). They are logically equivalent to the symbol 'first', although that symbol is not valid in a binary rule.

`right-edge, right, right-daughter` names for instruction argument

These three symbols are alternative names picking out the same constituent in a binary rule, namely the second of the two constituents seen from left to right (lower numbered positions in the chart to higher). They are logically equivalent to the symbol 'second', although that symbol is not valid in a binary rule.

`first, second, third` names for an instruction argument
`fourth, fifth, sixth,`
`seventh, eighth, ninth, tenth`

These symbols are to be used as arguments in a referent instruction for a 'n-ary' rule with three or more terms. They pick out their respective constituent edges (or rather the referents of those constituents) numerically starting from the left end of an edge's immediate constituents and proceeding to the right. Should an edge happen to have more than ten constituents (there are no constraints on the number of terms on the righthand side of an n-ary rule) there is no way provided in this version of Sparser for addressing them short of using a 'function' instruction and the global variables bound to the constituents to walk the chart to reach them. On the other hand there is little linguistic motivation for a context free rule of even half that size, so this limitation is not a difficulty in practice.

Note that all edges in fact only ever have only one or two immediate constituents (see discussion in §4.1.5). The capacity to define these virtual constituents by unwinding the left-recursive tree of edges that the completion of an n-ary rule produces is embedded within the mechanism that supports these symbols during rule calculation.

`daughter` name for an instruction argument

This symbol is to be used with unary (single constituent) rules. It picks out the sole constituent of such edges.

Usage: Assume we have a recursive rule that takes a phrase labeled "company" and extends it by spanning the determiner "the" just to its left. If we decide that the addition of the determiner doesn't change the denotation already established for the company, we

will just want to copy the referent on the daughter company edge up to the new edge, which we can do with this formulation of the referent expression:

```
(def-cfr  company  ("the" company)
   :referent  (:daughter  right-edge))
```

The symbol "right-edge" is used because the edge which has the object that we want to pass up to the newly formed edge (labeled "company") is the second of the two edges, the one on the right.

As another example, assume you have a procedure for defining a geographically-specific subsidiary of a company when you see evidence for one in a text, e.g. as in "Mitsubishi Corp. (Hong Kong)". This would call for using both pieces of information, the company and the location, and passing them as arguments to a function that will create the subsidiary, say, user:geographic-subsidiary. The result returned by that function will become the new referent of the spanning edge, which gets the label "company", the same as the label of its left edge.

```
(def-cfr  company  (company country-in-parenthesis)
   :referent  (:function  user:geographical-subsidiary
                          left-edge right-edge))
```

4.2.4 Global variables for referent calculation

There are occasions when the operation one wishes to perform in calculating a referent requires accessing the actual edge objects for the daughter constituents or certain other objects that are naturally accessible within the context of a referent calculation. To make this possible, a number of global variables are maintained, listed below, that are dynamically bound at the start of the reference calculating operation. These symbols are defined in the `sparser` package. Note that there is no provision in this version of Sparser for direct access to the edges of the constituents in an n-ary rule; should this be necessary, the requisite edges can be accessed by working down through the two daughter edges following the conventions for how n-ary rules are reformulated into 'dotted' binary rules, see §4.1.5.

`*referent*` `symbol (defvar)`

After the first referent instruction in a list of instructions has been executed, its value—the referent of the edge—is bound to this symbol.

`*left-edge-into-reference*` `symbol (defvar)`

Points to the left (first) edge of the two edges that are the actual immediate constituents of the edge.

`*right-edge-into-reference*` `symbol (defvar)`

Points to the right (second) edge of the two edges that are the actual immediate constituents of the edge.

`*single-daughter-edge*` `symbol (defvar)`

When the rule is unary, i.e. has only one term on its righthand side, this symbol points to that one immediate constituent edge.

`*parent-edge-getting-reference*` `symbol (defvar)`

Points to the newly completed edge whose referent is being calculated.

`*rule-being-interpreted*` `symbol (defvar)`

Points to the phrase structure rule (cfr object) that is responsible for the new edge. Its referent field is supplying the instructions to calculate this edge's referent.

4.3 Context-sensitive rules

Sparser provides a facility for a limited kind of context sensitive phrase structure rule, built on top of the facility for parsing context-free rules. Generally speaking, a context-sensitive rewrite rule (Chomsky Type One) can have any number of terms on its lefthand side, most of which provide a "context" which stipulates in what neighborhood of adjacent constituents one of those terms may be rewritten by the terms on the rule's righthand side, preserving the 'context' terms to its right and left.

Using Sparser we can define a restricted subset of such rules using the machinery provide by the `cfr` object and the basic parsing protocol that notices adjacent edges. The restriction is that the term being rewritten can take its context either from its right or from its left but not both, and that the context can only be a single constituent. This is a consequence of the fact that context-sensitive rules are built on the machinery of binary context-free rules, and has not yet been extended to n-ary rules.

One can write context-sensitive rules of the form

(1) `A -> B / C ___` or (2) `A -> B / ___ C`

which mean that label A will be rewritten as label B whenever label A is adjacent to label C on its left (case one) or adjacent on its right (case two). In terms of edges in the chart, that is to say that the parser will be looking for two adjacent edges with labels A and C (the labels being either categories, words, or polywords). When it finds such a pair, a new edge, with label B, will be introduced over edge A. With a context-free rule on the other hand, the pair of constituents A and C would have both been spanned by the new edge; with a context sensitive rule only the A edge is covered, leaving the top edges of the chart as B, C or as C, B depending on whether the context was specified as being to the right or to the left.

As a concrete example, imagine that you have a conservative set of rules for recognizing proper names, which, when faced with a phrase where there is not enough evidence from the words in the phrase itself to determine the category of object that it names, will just give the phrase the label "name" and then wait for the context in which the phrase appears to be established and to determine the phrase's category from that. Thus we might have the text "*Miffler Swope retired as vice president of Ajax Corp*.", and

we would want to have the name "*Miffler Swope*" relabeled as a person when it appears before the word "*retired*". We can do that with the following rule, written informally as

```
name -> person / ___ "retired"
```

Here the parser will look for a pair of adjacent constituents with the labels "name" and "retired". When it finds them it creates a new edge labeled "person" over the same span as that of the "name" constituent. (Note that the direction of the arrow has a different interpretation in these context sensitive rules than it does for Sparser's context-free rules. In a context free rule, the righthand side constituent(s) is covered by a new edge with the label given by the lefthand side of the rule; for a context sensitive rule it is the lefthand side constituent that is respaned by the right.)

4.3.1 Defining context-sensitive rules

As with context-free rules, there are two defining forms, one a function and the other a macro. The macro is most convenient when the rules are being defined from a text file and the labels are given as symbols or strings. The function is more convenient when the definition is embedded in a function where you are already manipulating the needed objects. The provision for a referent is the same as for context-free rules, except that it will be establishing the referent for the new edge (e.g. the B edge that respans A) and not for the pair as a whole.

```
def-csr ( daughter  parent                                          macro
          &key :left-context    label
               :right-context   label
               :referent <special list or label>
               :form <category label>
               :relation <category label> )
```

Returns a cfr object that is specially interpreted by the parser as a context-sensitive rule as described above. One of the two keyword arguments :left-context or :right-context is required. The referent, form, and relation keyword arguments are interpreted as in Def-cfr, see §4.1.4. None of the arguments are evaluated.

The arguments indicated as "daughter" and "parent" take labels as values, i.e. symbols or strings, denoting categories, or words or polywords, respectively. If objects corresponding to the spellings of those arguments do not exist at the time the form is evaluated they will be created. When the rule completes, the edge labeled with the "daughter" label will be covered by a new edge that will be given the "parent" label.

The left (or right) context arguments indicate which label is to appear on the adjacent context-defining edge, and specify whether this edge is to appear adjacent to the left or to the right of the edge with the daughter label, depending on which keyword is used. When the rule completes nothing will happen to this edge.

```
Define-csr ( daughter  parent                                       function
             &key :left-context    label
```

```
:right-context   label
:referent <special list or label>
:form <category label>
:relation <category label> )
```

Returns a cfr object that is specially interpreted by the parser as a context-sensitive rule as described above. The arguments are interpreted as in Def-csr, except that since this routine is a function they are all evaluated. The categories or words supplied to the function must already be defined as they will not be defined automatically.

As a structure, a cfr object that specifies a context-sensitive phrase structure rule has a different print form than one for a context-free rule. The form mimics the way the rule would be written in standard literature as illustrated at the beginning of this section. The example given there would look like this:

```
#<PSR6  name -> person / ___ "retired">
```

The form defining that rule looks like this, assuming of course that the categories ‘name’ and ‘person’ have been defined.

```
(def-csr name person
    :right-context "retired")
```

4.4 Keeping track of rules

When a rule is defined, several indexes are created to keep track of it. The rule object is bound to a symbol for easy access interactively, and the object is put on a globally available list. The object may also be cross-indexed to the words/categories that make it up via their rule-sets. Deleting the rule removes all these indices.

The symbol associated with a rule is displayed as the first part of the rule’s print form, e.g. `#<psr37 MONTH -> “December”>`. The symbol is internal to its package, consequently it must be referred to using two colons, e.g. `rule::psr37`. Other means of accessing rules are given below. These apply to phrase structure rules of all kinds because they are based on the cfr object type that they all employ.

`*context-free-rules-defined*` symbol (defvar)

This symbol is bound to a list of all the cfr objects currently defined. The list is ordered from the most recently defined rule to the earliest.

`find-cfr  (<label> <list-of-labels>)` function

Returns the cfr object whose lefthand side is the first argument and whose righthand side is the second argument, or `nil` if there is no such rule currently defined. The labels should be expressed as strings for words and polywords and as symbols for categories; note that the arguments are evaluated.

`display-all-cfrs (&optional stream)` function

Displays all the currently defined context-free rules on successive lines of the indicated stream in order from the most recently defined to the earliest, along with their rule numbers. When called with no arguments, the display is to the listener window (i.e. `*standard-output*`).

4.5 Deleting rules

To delete a cfr is to remove it from the set of indices that establish its righthand side as a pattern of constituents that the parser can recognize. It is simultaneously removed from all the other indices that point to the cfr object, allowing it to be garbage collected.

You will want to delete a rule if you no longer want it to be part of the set of patterns Sparser considers in analyzing a text—perhaps you have changed your mind about how a given analysis should be done. You also need deletion if you have to make any changes to the lefthand side of a rule, i.e. keeping the righthand side pattern in the analysis, but changing the choice of lefthand side, even just changing how it is spelled. If what you want to do is change the referent of a rule then all you have to do is re-evaluate the modified definition; the same cfr object will be returned as was before, but now with the new value for its referent field.

`delete/cfr (cfr)` function

Returns the cfr object, having first removed it from all the indices that allow it to function within the parser's grammar. If you do not keep a pointer to the returned object it will be garbage collected. Re-evaluating the definition form corresponding to this rule will result in the construction of a new cfr object.

`delete/cfr# (number)` function

A sometimes more convenient form for deleting cfrs. Uses the number to lookup the cfr that is bound to the rule symbol (gensym) with that number and then calls Delete/cfr on that object. The number of a rule is included in its structure print form, and as part of the listing provided by display-all-cfrs.

`delete-cfr (<label> <list-of-labels>)` macro

The arguments to this version of the delete function mimic the pattern of find-cfr or def-cfr. The labels should be expressed as strings for words and polywords and as symbols for categories; note that the arguments are not evaluated. The rule with the first label as its lefthand side and the list of labels (or of one label) as its righthand side is identified and deleted by passing it to delete/cfr.

When one deletes a rule, the labels that comprise it are not affected. They remain defined, and act normally in any of the other rules that mention them. This is true even if they are used in no other rule. By contrast, if one were to delete some word or category that a rule mentions and then redefine that label, there would be a new object representing

the label—a different object from the one still used in the rule—with the effect that the rule would no longer work as intended.

4.6 Rule-sets

In Sparser, the information about how phrases may form and be combined is based on the labels used in rules or applied to hooks. All labels—words, polywords and categories—have essentially the same possibilities for participating in rules, and this is reflected by having the connection between them and the rules be mediated by the same data structure, the "rule-set". Words are linked to their rule sets by the access function `word-rule-set`; categories by `cat-rule-set`, and polywords by `pw-rules`.

There are several other fields in a rule set than the ones described here. These contain information that pertains to extensions to Sparser presently under development. Similarly there there are some details of the fields that are not described, since they are liable to change and are not otherwise needed by the facilities described in this documentation.

`rule-set` `defstruct`

Rule-set objects are the parser's representation of how a label participates in rules. The printed form of a rule-set is, e.g.,

```
#<rule-set for #<category MONTH>>
```

which simply states what label the rule set is associated with.

`rs-fsa (rule-set)` `access-function`

Returns `nil` if the label triggers no word-level routines (§7.2). Otherwise it returns a list that will contain one or more symbols and/or a cfr object. The symbols name functions. The cfr, if present, corresponds to one or more polywords that a word label may initiate.

`rs-completion-actions (rule-set)` `access-function`

The contents of this field are processed by the 'completion action' facilities (§7.1)

`rs-backpointer (rule-set)` `access-function`

Returns the label that this rule-set is associated with. Forms the basis of the print form.

Rule-sets are constructed automatically for labels in rules. The rule definition facilities looks up the labels, and if there is no rule-set object already linked to the label they build one. When you define an action routine (§7.2), you may also need to explicitly create and link in a rule set if the word that you have designated as initiating the routine does not already have one. This is straightforward (assuming you have worked with Lisp structures before). You simply use the Lisp-supplied 'make' form for the rule set structure to create it (`make-rule-set`), and `setf` its backpointer and `fsa` fields

appropriately. You then `setf` the corresponding field on the label to link the new rule-set to the label. There is an example in §7.2.1.