Libmarpa

Version 9.0.2
23 June 2022

Jeffrey Kegler

This manual (23 June 2022) is for Libmarpa 9.0.2.

Published 23 June 2022 by Jeffrey Kegler

Table of Contents

1 No warranty

The Libmarpa license takes precedence over the statements in this document. In particular, the license states that Libmarpa is free software and has no warranty. No statement in this document should be construed as providing any kind of warranty.

2 About this document

2.1 How to read this document

This is essentially a reference document, but its early chapters lay out concepts essential to the others. Readers will usually want to read the chapters up and including Chapter 10 [Introduction to the method descriptions], page 19, in order. Otherwise, they should follow their interests.

2.2 Prerequisites

This document is very far from self-contained. It assumes the following:

- The reader knows the C programming language at least well enough to understand function prototypes and return values.
- The reader has read the documents for one of Libmarpa's upper layers. As of this writing, the only such layer is `Marpa::R2` or `Marpa::R3`, in Perl.
- The reader knows some parsing theory (Section 2.3 [Parsing theory], page 2).

2.3 Parsing theory

This document assumes some acquaintance with parsing theory. The reader's level of knowledge is probably adequate if he can answer the following questions, either immediately or after a little reflection.

- What is a BNF rule?
- What is a Marpa sequence rule?
- As a reminder, Marpa's sequence rules are implemented as left recursions. What does that mean?
- Take a Marpa sequence rule at random. What does it look like when rewritten in BNF?
- What does the sequence look like when rewritten in BNF as a right-recursion?

2.4 Terminology and notation

In this document,

- A *boolean* value, or *boolean*, is an integer which is 0 or 1.
- *iff* abbreviates "if and only if".
- *application* means an "application" of Libmarpa. In this document, a Libmarpa application is not necessarily an application program. For our purposes, an "application" might be another library which uses Libmarpa.
- `max(x,y)` is the maximum of `x` and `y`, where `x` and `y` are two numbers.
- *Libmarpa method*, or just *method* means a C function or a function-like macro of the Libmarpa library.
- *user* means a "user" of the Libmarpa library. A user of the library is also a programmer, so that in this documents, "user" and "programmer" are essentially synonyms.

- *We* (and “us” and ”our”) refer to the authors. As of this writing, there is a primary author, but the plural is traditional, and our “we” is intended to include the reader and everyone we are joining on the millenia-old voyage of discovery into mathematics and language.

2.4.1 Application and diagnostic behavior

An *application behavior* is a behavior on which it is intended that the design of applications will be based. Most of the behaviors specified in this document are application behaviors. We sometimes say that “applications may expect” a certain behavior to emphasize that that behavior is an application behavior.

After an irrecoverable failure, the behavior of a Libmarpa application is undefined, so that there are no behaviors which can be relied on for normal application processing, and therefore, there are no application behaviors. In this circumstance, some of the application behaviors become diagnostic behaviors. A *diagnostic behavior* is a behavior which it is suggested that the programmer may attempt in the face of an irrecoverable failure, for testing, diagnostics and debugging. They are hoped for, rather than expected, and intended to allow the programmer to deal with irrecoverable failures as smoothly as possible. (See Chapter 9 [Failure], page 14.)

In this document, a behavior is a diagnostic behavior only if that is specifically indicated. Applications should not be designed to rely on diagnostics behaviors. We sometimes say that “diagnostics may attempt” a certain behavior to emphasize that that behavior is a diagnostic behavior.

3 About Libmarpa

Libmarpa implements the Marpa parsing algorithm. Marpa is named after the legendary 11th century Tibetan translator, Marpa Lotsawa. In creating Marpa, I depended heavily on previous work by Jay Earley, Joop Leo, John Aycock and Nigel Horspool.

Libmarpa implements the entire Marpa algorithm. This library does the necessary grammar preprocessing, recognizes the input, and produces parse trees. It also supports the ordering, iteration and evaluation of the parse trees.

Libmarpa is very low-level. For example, it has no strings. Rules, symbols, and token values are all represented by integers. This, of course, will not suffice for many applications. Users will very often want names for the symbols, non-integer values for tokens, or both. Typically, applications will use arrays to translate Libmarpa's integer ID's to strings or other values as required.

Libmarpa also does **not** implement most of the semantics. Libmarpa does have an evaluator (called a "valuator"), but it does **not** manipulate the stack directly. Instead, Libmarpa, based on its traversal of the parse tree, passes optimized step by step stack manipulation instructions to the upper layer. These instructions indicate the token or rule involved, and the proper location for the true token value or the result of the rule evaluation. For rule evaluations, the instructions include the stack location of the arguments.

Marpa requires most semantics to be implemented in the application. This allows the application total flexibility. It also puts the application is in a much better position to prevent errors, to catch errors at runtime or, failing all else, to successfully debug the logic.

4 Architecture

4.1 Major objects

The classes of Libmarpa's object system fall into two types: major and numbered. These are the Libmarpa's major classes, in sequence.

- Configuration: A configuration object is a thread-safe way to hold configuration variables, as well as the return code from failed attempts to create grammar objects.
- Grammar: A grammar object contains rules and symbols, with their properties.
- Recognizer: A recognizer object reads input.
- Bocage: A bocage object is a collection of parse trees, as found by a recognizer. Bocages are similar to parse forests.
- Ordering: An ordering object is an ordering of the trees in a bocage.
- Tree: A tree object is a bocage iterator.
- Value: A value object is a tree iterator. Iteration of tree using a value object produces "steps". These "steps" are instructions to the application on how to evaluate the semantics, and how to manipulate the stack.

The major objects have one letter abbreviations, which are used frequently. These are, in the standard sequence,

- Configuration: C
- Grammar: G
- Recognizer: R
- Bocage: B
- Ordering: O
- Tree: T
- Value: V

4.2 Time objects

All of Libmarpa's major classes, except the configuration class, are "time" classes. Except for objects in the grammar class, all time objects are created from another time object. Each time object is created from a time object of the class before it in the sequence. A recognizer cannot be created without a precomputed grammar; a bocage cannot be created without a recognizer; and so on.

When one time object is used to create a second time object, the first time object is the *parent object* and the second time object is the *child object*. For example, when a bocage is created from a recognizer, the recognizer is the parent object, and the bocage is the child object.

Grammars have no parent object. Every other time object has exactly one parent object. Value objects have no child objects. All other time objects can have any number of children, from zero up to a number determined by memory or some other machine-determined limit.

Every time object has a *base grammar*. A grammar object is its own base grammar. The base grammar of a recognizer is the grammar that it was created with. The base grammar

of any other time object is the base grammar of its parent object. For example, the base grammar of a bocage is the base grammar of the recognizer that it was created with.

4.3 Reference counting

Every object in a "time" class has its own, distinct, lifetime, which is controlled by the object's reference count. Reference counting follows the usual practice. Contexts which take a share of the "ownership" of an object increase the reference count by 1. When a context relinquishes its share of the ownership of an object, it decreases the reference count by 1.

Each class of time object has a "ref" and an "unref" method, to be used by those contexts which need to explicitly increment and decrement the reference count. For example, the "ref" method for the grammar class is `marpa_g_ref()` and the "unref" method for the grammar class is `marpa_g_unref()`.

Time objects do not have explicit destructors. When the reference count of a time object reaches 0, that time object is destroyed.

Much of the necessary reference counting is performed automatically. The context calling the constructor of a time object does not need to explicitly increase the reference count, because Libmarpa time objects are always created with a reference count of 1.

Child objects "own" their parents, and when a child object is successfully created, the reference count of its parent object is automatically incremented to reflect this. When a child object is destroyed, it automatically decrements the reference count of its parent.

In a typical application, a calling context needs only to remember to "unref" each time object that it creates, once it is finished with that time object. All other reference decrements and increments are taken care of automatically. The typical application never needs to explicitly call one of the "ref" methods.

More complex applications may find it convenient to have one or more contexts share ownership of objects created in another context. These more complex situations are the only cases in which the "ref" methods will be needed.

4.4 Numbered objects

In addition to its major, "time" objects, Libmarpa also has numbered objects. Numbered objects do not have lifetimes of their own. Every numbered object belongs to a time object, and is destroyed with it. Rules and symbols are numbered objects. Tokens values are another class of numbered objects.

5 Input

5.1 Earlemes

5.1.1 The traditional input model

In traditional Earley parsers, the concept of location is very simple. Locations are numbered from 0 to *n*, where *n* is the length of the input. Every location has an Earley set, and vice versa. Location 0 is the start location. Every location after the start location has exactly one input token associated with it.

Some applications do not fit this traditional input model — natural language processing requires ambiguous tokens, for example. Libmarpa allows a wide variety of alternative input models.

In Libmarpa a location is called a *earleme*. The number of an Earley set is the *ID of the Earley set*, or its *ordinal*. In the traditional model, the ordinal of an Earley set and its earleme are always exactly the same, but in Libmarpa's advanced input models the ordinal of an Earley set can be different from its location (earleme).

The important earleme values are the latest earleme. the current earleme, and the furthest earleme. Latest, current and furthest earleme, when they have determinate values, obey a lexical order in this sense: The latest earleme is always at or before the current earleme, and the current earleme is always at or before the furthest earleme.

5.1.2 The latest earleme

The *latest Earley set* is the Earley set completed most recently. This is initially the Earley set at location 0. The latest Earley set is always the Earley set with the highest ordinal, and the Earley set with the highest earleme location. The *latest earleme* is the earleme of the latest Earley set. If there is an Earley set at the current earleme, it is the latest Earley set and the latest earleme is equal to the current earleme. There is never an Earley set after the current earleme, and therefore the latest Earley set is never after the current earleme. The `marpa_r_start input()` and `marpa_r_earleme_complete()` methods are only ones that change the latest earleme. See [marpa_r_start_input], page 34, and [marpa_r_earleme_complete], page 36.

The latest earleme is different from the current earleme if and only if there is no Earley set at the current earleme. A different end of parsing can be specified, but by default, parsing is of the input in the range from earleme 0 to the latest earleme.

5.1.3 The current earleme

The *current earleme* is the earleme that Libmarpa is currently working on. More specifically, it is the one at which new tokens will **start**. Since tokens are never zero length, a new token will always end after the current earleme. `marpa_r_start_input()` initializes the current earleme to 0, and every call to `marpa_r_earleme_complete()` advances the current earleme by 1. The `marpa_r_start input()` and `marpa_r_earleme_complete()` methods are only ones that change the current earleme. See [marpa_r_start_input], page 34, and [marpa_r_earleme_complete], page 36.

5.1.4 The furthest earleme

Loosely speaking, the *furthest earleme* is the furthest earleme reached by the parse. More precisely, it is the highest numbered earleme at which a token ends and is 0 if there are no tokens. The furthest earleme is 0 when a recognizer is created. With every call to `marpa_r_alternative()`, the end of the token it adds is calculated. A token ends at the earleme location *current*+*length*, where *current* is the current earleme, and *length* is the length of the newly added token. If `old_f` is the furthest earleme before a call to `marpa_r_alternative()`, the furthest earleme after the call is `max(old_f, current+length)`. The `marpa_r_new()` and `marpa_r_alternative()` methods are only ones that change the furthest earleme. See [marpa_r_new], page 34, and [marpa_r_alternative], page 35.

In the basic input models, where every token has length 1, calling `marpa_r_earleme_complete()` after each `marpa_r_alternative()` call is sufficient to process all inputs, and the furthest earleme's value can be typically be ignored. In alternative input models, where tokens have lengths greater than 1, calling `marpa_r_earleme_complete()` once after the last token is read may not be enough to ensure that all tokens have been processed. To ensure that all tokens have been processed, an application must advance the current earleme by calling `marpa_r_earleme_complete()`, until the current earleme is equal to the furthest earleme.

5.2 The basic models of input

For the purposes of presentation, we (somewhat arbitrarily) divide Libmarpa's input models into two groups: basic and advanced. In the *basic input models of input*, every token is exactly one earleme long. This implies that, in a basic model of input,

- every token is the same length,
- the ordinal of an Earley set will always be the same as its earleme location, and
- the latest earleme and the current earleme are always equal.

In the *advanced models of input*, tokens may have a length other than 1. Most applications use the basic input models. The details of the advanced models of input are presented in a later chapter. See Chapter 23 [Advanced input models], page 77.

5.2.1 The standard model of input

In the standard model of input, there is exactly one successful `marpa_r_alternative()` call immediately previous to every `marpa_r_earleme_complete()` call. A `marpa_r_alternative()` call is *immediately previous* to a `marpa_r_earleme_complete()` call iff that `marpa_r_earleme_complete()` call is the first `marpa_r_earleme_complete()` call after the `marpa_r_alternative()` call.

Recall that, since the standard model is a basic model, the token length in every successful call to `marpa_r_alternative()` will be one. For an input of length n, there will be exactly n `marpa_r_earleme_complete()` calls, and all but the last call to `marpa_r_earleme_complete()` must be successful.

In the standard model, after a successful call to `marpa_r_alternative()`, if c is the value of the current earleme before the call,

- the current earleme will remain unchanged and therefore will be c; and
- the furthest earleme be c+1.

In the standard model, a call to `marpa_r_earleme_complete()` follows a successful call of `marpa_r_alternative()`, so that the value of the furthest earleme before the call to `marpa_r_earleme_complete()` will be `c+1`, where *c* is the value of the current earleme. After a successful call to `marpa_r_earleme_complete()`,

- the current earleme will be advanced to `c+1`; and
- the furthest earleme will be *c+1*, and therefore equal to the current earleme.

Recall that, in the basic models of input, the latest earleme is always equal to the current earleme.

5.2.2 Ambiguous input

We can loosen the standard model to allow more than one successful call to `marpa_r_alternative()` immediately previous to each call to `marpa_r_earleme_complete()`. This change will mean that multiple tokens become possible at each earleme — in other words, that the input becomes ambiguous. We continue to require that there be at least one successful call to `marpa_r_alternative()` before each call to `marpa_r_earleme_complete()`. And we recall that, since this is a basic input model, all tokens must have a length of 1.

In the ambiguous input model, the behavior of the current, latest and furthest earlemes are exactly as described for the standard model. See Section 5.2.1 [The standard model of input], page 8.

5.3 Terminals

A terminal symbol is a symbol which may appear in the input. Traditionally, all LHS symbols, as well as the start symbol, must be non-terminals. This is Marpa's behavior, by default.

Marpa allows the user to eliminate the distinction between terminals and non-terminals. In this, it differs from traditional parsers. Libmarpa can arrange for a terminal to appear on the LHS of one or more rules, or for a terminal to be the start symbol. However, since terminals can never be zero length, it is a logical contradiction for a nulling symbol to also be a terminal and Marpa does not allow it.

Token values are `int`'s. Libmarpa does nothing with token values except accept them from the application and return them during parse evaluation.

6 Exhaustion

A parse is *exhausted* when it cannot accept any further input. A parse is *active* iff it is not exhausted. For a parse to be exhausted, the furthest earleme and the current earleme must be equal. However, the converse is not always the case: if more tokens can be read at the current earleme, then it is possible for the furthest earleme and the current earleme to be equal in an active parse.

Parse exhaustion always has a location. That is, if a parse is exhausted it is exhausted at some earleme location `X`. If a parse is exhausted at location `X`, then

- There may be valid parses at `X`.
- The parse was active at all locations earlier than `X`.
- There may be valid parses at locations before `X`.
- There will be no valid parses at locations after `X`.
- No tokens can start at location `X`.
- No tokens can end at a location after `X`.
- No tokens can start at any location after `X`.
- No tokens will be accepted by an exhausted parser. It is an irrecoverable hard failure to call `marpa_r_alternative()` after a parser has become exhausted.
- No Earley sets will be at any location after `X`.
- No earlemes are completed by, and no Earley sets are created by, an exhausted parser. It is an irrecoverable hard failure to call `marpa_r_earleme_complete()` after a parser has become exhausted.

Users sometimes assume that parse exhaustion means parse failure. But other users sometimes assume that parse exhaustion means parse success. For many grammars, there are strong associations between parse exhaustion and parse success, but the strong association can go either way, Both exhaustion-loving and exhaustion-hating grammars are very common in practical application.

In an *exhaustion-hating* application, parse exhaustion typically means parse failure. C programs, Perl scripts and most programming languages are exhaustion-hating applications. If a C program is well-formed, it is always possible to read more input. The same is true of a Perl program that does not have a `__DATA__` section.

In an *exhaustion-loving* application parse exhaustion means parse success. A toy example of an exhaustion-loving application is the language consisting of balanced parentheses. When the parentheses come into perfect balance the parse is exhausted, because any further input would unbalance the brackets. And the parse succeeds when the parentheses come into perfect balance. Exhaustion means success. Any language which balances start and end indicators will tend to be exhaustion-loving. HTML and XML, with their start and end tags, can be seen as exhaustion-loving languages.

One common form of exhaustion-loving parsing occurs in lexers which look for longest matches. Exhaustion will indicate that the longest match has been found.

It is possible for a language to be exhaustion-loving at some points and exhaustion-hating at others. We mentioned Perl's `__DATA__` as a complication in a basically exhaustion-hating language.

`marpa_r_earleme_complete()` and `marpa_r_start_input` are the only methods that may encounter parse exhaustion. See [marpa_r_earleme_complete], page 36, and [marpa_r_start_input], page 34. When the `marpa_r_start_input` or `marpa_r_earleme_complete()` methods exhaust the parse, they generate a `MARPA_EVENT_EXHAUSTED` event. Applications can also query parse exhaustion status directly with the `marpa_r_is_exhausted()` method. See [marpa_r_is_exhausted], page 39.

7 Semantics

Libmarpa handling of semantics is unusual. Most semantics are left up to the application, but Libmarpa guides them. Specifically, the application is expected to maintain the evaluation stack. Libmarpa's valuator provides instructions on how to handle the stack. Libmarpa's stack handling instructions are called "steps". For example, a Libmarpa step might tell the application that the value of a token needs to go into a certain stack position. Or a Libmarpa step might tell the application that a rule is to be evaluated. For rule evalution, Libmarpa will tell the application where the operands are to be found, and where the result must go.

The detailed discussion of Libmarpa's handling of semantics is in the reference chapters of this document, under the appropriate methods and classes. The most extensive discussion of the semantics is in the section that deals with the methods of the value time class (Chapter 19 [Value methods], page 49).

8 Threads

Libmarpa is thread-safe, given circumstances as described below. The Libmarpa methods are not reentrant.

Libmarpa is C89-compliant. It uses no global data, and calls only the routines that are defined in the C89 standard and that can be made thread-safe. In most modern implementations, the default C89 implementation is thread-safe to the extent possible. But the C89 standard does not require thread-safety, and even most modern environments allow the user to turn thread safety off. To be thread-safe, Libmarpa must be compiled and linked in an environment that provides thread-safety.

While Libmarpa can be used safely across multiple threads, a Libmarpa grammar cannot be. Further, a Libmarpa time object can only be used safely in the same thread as its base grammar. This is because all time objects with the same base grammar share data from that base grammar.

To work around this limitation, the same grammar definition can be used to a create a new Libmarpa grammar time object in each thread. If there is sufficient interest, future versions of Libmarpa could allow thread-safe cloning of grammars and other time objects.

9 Failure

As a reminder, no language in this chapter (or, for that matter, in this document) should be read as providing, or suggesting the existence of, a warranty. See [license], page 2. Also, see Chapter 1 [No warranty], page 1.

9.1 Libmarpa's approach to failure

Libmarpa is a C language library, and inherits the traditional C language approach to avoiding and handling user programming errors. This approach will strike readers unfamiliar with this tradition as putting an appallingly large portion of the burden of avoiding application programmer error on the application programmer themself.

But in the early 1970's, when the C language first stabilized, the alternative, and the consensus choice for its target applications was assembly language. In that context, C was radical in its willingness to incur a price in efficiency in order to protect the programmer from themself. C was considered to take a excessively "hand holding" approach which very much flew in the face of consensus.

The decades have made a large difference in the trade-offs, and the consensus about the degree to which even a low-level language should protect the user has changed. It seems inevitable that C will be replaced as the low-level language of choice, by a language which places fewer burdens on the programmer, and more on the machine. The question seems to be not whether C will be dethroned as the "go to" language for low-level progamming, but when, and by which alternative.

Modern hardware makes many simple checks essentially cost-free, and Libmarpa's efforts to protect the application programmer go well beyond what would have been considered best practice in the past. But it remains a C language library. But, on the whole, the Libmarpa application programmer must be prepared to exercise the high degree of carefulness traditionally required by its C language environment. Libmarpa places the burden of avoiding irrecoverable failures, and of handling recoverable failures, largely on the application programmer.

9.2 User non-conformity to specified behavior

This document specifies many behaviors for Libmarpa application programs to follow, such as the nature of the arguments to each method. The C language environment specifies many more behaviors, such as proper memory management. When a non-conformity to specified behavior is unintentional and problematic, it is frequently called a "bug". Even the most carefully programmed Libmarpa application may sometimes contain a "bug". In addition, some specified behaviors are explicitly stated as characterizing a primary branch of the processing, rather than made mandatory for all successful processing. Non-conformity to non-mandatory behaviors can be efficiently recoverable, and is often intentional.

This chapter describes how non-conformity to specified behavior by a Libmarpa application is handled by Libmarpa. Non-conformity to specified behavior by a Libmarpa application is also called, for the purposes of this document, a *Libmarpa application programming failure*. In contexts where no ambiguity arises, *Libmarpa application programming failure* will usually be abbreviated to *failure*.

Libmarpa application programming success in a context is defined as the absence of unrecovered failure in that context. When no ambiguity arises, *Libmarpa application programming success* is almost always abbreviated to *success*. For example, the success of an application means the application ran without any irrecoverable failures, and that it recovered from all the recoverable failures that were detected.

9.3 Classifying failure

A Libmarpa application programming failure, unless specified otherwise, is an irrecoverable failure. Once an irrecoverable failure has occurred, the further behavior of the program is undefined. Nonetheless, we specify, and Libmarpa attempts, diagnostics behaviors (see Section 2.4.1 [Application and diagnostic behavior], page 3) in an effort to handle irrecoverable failures as smoothly as possible.

A Libmarpa application programming failure is recoverable if and only if it is specified as such.

A failure is called a *hard failure* is it has an error code associated with it. A recoverable failure is called a *soft failure* if it has no associated error code. (For more on error codes, see Section 9.11 [Error codes], page 18.)

All failures fall into one of five types. In order of severity, these are

- **memory allocation failures**,
- **undetected failures**,
- **irrecoverable hard failures**,
- **partially recoverable hard failures**, and
- **fully recoverable hard failures**, and
- **soft failures**.

9.4 Memory allocation failure

Failure to allocate memory is the most irrecoverable of irrecoverable errors. Even effective error handling assumes the ability to allocate memory, so that the practice has been, in the event of a memory allocation failure, to take Draconian action. On *memory allocation failure*, as with all irrecoverable failures, Libmarpa's behavior in undefined, but Libmarpa attempts to terminate the current program abnormally by calling `abort()`.

Memory allocation failure is the only case in which Libmarpa terminates the program. In all other cases, Libmarpa leaves the decision to terminate the program, whether normally or abnormally, up to the application programmer.

Memory allocation failure does not have an error code. As a pedantic matter, memory allocation failure is neither a hard or a soft failure.

9.5 Undetected failure

An *undetected failure* is a failure that the Libmarpa library does not detect. Many failures are impossible or impractical for a C library to detect. Two examples of failure that the Libmarpa methods do not detect are writes outside the bounds of allocated memory, and use of memory after it has been freed. C is not strongly typed, and arguments of Libmarpa

routines undergo only a few simple tests, tests which are inadequate to detect many of the potential problems.

By undetected failure we emphasize that we mean failures undetected **by the Libmarpa methods**. In the examples just given, there exist tools that can help the programmer detect memory errors and other tools exist to check the sanity of method arguments.

This document points out some of the potentially undetected problems, when doing so seems more helpful than tedious. But any attempt to list all the undetected problems would be too large and unwieldy to be useful.

Undetected failure is always irrecoverable. An undetected failure is neither a hard or a soft failure.

9.6 Irrecoverable hard failure

An *irrecoverable hard failure* is an irrecoverable Libmarpa application programming failure that has an error code associated with it. Libmarpa attempts to behave as predictably as possible in the face of a hard failure, but once an irrecoverable failure occurs, the behavior of a Libmarpa application is undefined.

In the event of an irrecoverable failure, there are no application behaviors. The diagnostic behavior for a hard failure is as described for the method which detects the hard failure. At a minimum, this diagnostic behavior will be returning from the method which detects the hard failure with the return value specified for hard failure, and setting the error code as specified for hard failure.

9.7 Partially recoverable hard failure

A *partially recoverable hard failure* is a recoverable Libmarpa application programming failure

- that has an error code associated with it; and
- after which some, but not all, of the application behaviors remain available to the programmer.

For every partially recoverable hard failure, this document specifies the application behaviors that remain available after it occurs. The most common kind of partially recoverable hard failure is a library-recoverable hard failure. For an example of partially recoverable hard failure, see Section 9.8 [Library-recoverable hard failure], page 16.

9.8 Library-recoverable hard failure

A *library-recoverable hard failure* is a type of partially recoverable hard failure. Loosely described, it is a hard failure which allows the programmer to continue to use many of the Libmarpa methods in the library, but which disallows certain methods on some objects.

To state the restrictions of application behaviors more precisely, let the "failure grammar" be the base grammar of the method which detected the library-recoverable hard failure. After a library-recoverable hard failure, the following behaviors are no longer applcation behaviors:

- Libmarpa mutator and constructor method calls where the base grammar is the failure grammar.

Recall that any use of a behavior which is not an application behavior is an irrecoverable failure.

The application behaviors remaining after a library-recoverable hard failure are the following:

- All Libmarpa accessor method calls, even those whose base grammar is the failure grammar.
- All Libmarpa destructor method calls, even those whose base grammar is the failure grammar. An application will often want to destroy all Libmarpa objects whose base grammar is the failure grammar, in order to clear memory of unusable objects.
- All Libmarpa mutator and constructor method calls, except those whose base grammar is the failure grammar.
- All Libmarpa static method calls.
- All use of non-Libmarpa interfaces, including other libraries and the C language environment.

An example of a library-recoverable hard failure is the `MARPA_ERR_COUNTED_NULLABLE` error in the `marpa_g_precompute` method. See [marpa_g_precompute], page 32.

9.9 Fully recoverable hard failure

A *fully recoverable hard failure* is a recoverable Libmarpa application programming failure

- that has an error code associated with it; and
- after which all of the application behaviors remain available to the programmer.

One example of a fully recoverable hard failure is the error code `MARPA_ERR_UNEXPECTED_TOKEN_ID`. The "Ruby Slippers" parsing technique (see [Ruby Slippers], page 36), which has seen extensive usage, is based on Libmarpa's ability to recover from a `MARPA_ERR_UNEXPECTED_TOKEN_ID` error fully and efficiently,

9.10 Soft failure

An *soft failure* is an recoverable Libmarpa application programming failure that has no error code associated with it. Hard errors are assigned error codes in order to tell them apart. Error codes are not necessary or useful for soft errors, because there is at most one type of soft failure per Libmarpa method.

Soft failures are so called, because they are the least severe kind of failure. The most severe failures are "bugs" — unintended, and a symptom of a problem. Soft failures, on the other hand, are a frequent occurrence in normal, successful, processing. In the phrase "soft failure", the word "failure" is used in the same sense that its cognate "fail" is used when we say that a loop terminates when it "fails" its loop condition. That "failure" is of a condition necessary to continue on a main branch of processing, and a signal to proceed on another branch.

It is expected that Libmarpa applications will be designed such that successful execution is based on the handling specified for soft failures. In fact, a non-trival Libmarpa application can hardly be designed except on that basis.

9.11 Error codes

As stated, every hard failure has an associated error code. Full descriptions of the error codes that are returned by the external methods are given in their own section (Section 21.3 [External error codes], page 64).

How the error code is accessed depends on the method which detects the hard failure associated with that error code. Methods for time objects always set the error code in the base grammar, from which it may be accessed using the error methods described below (Section 21.1 [Error methods], page 64). If a method has no base grammar, the way in which the error code for the hard failures that it detects can be accessed will be stated in the description of that method.

Since the error of a time object is set in the base grammar, it follows that every object with the same base grammar has the same error code. Objects with different base grammars may have different error codes.

While error codes are properties of a base grammar, irrecoverability is application-wide. That is, whenever any irrecoverable failure occurs, the entire application is irrecoverable. Once an application becomes irrecoverable, those Libmarpa objects with error codes for recoverable errors are still subject to the general irrecoverability.

10 Introduction to the method descriptions

The following chapters describe Libmarpa's methods in detail.

10.1 About the overviews

The method descriptions are grouped into chapters and sections. Each such group of methods descriptions begins, optionally, with an overview. These overviews, again optionally, end with a "cheat sheet". The "cheat sheets" name the most important Libmarpa methods in that chapter or section, in the order in which they are typically used, and very briefly describe their purpose.

The overviews sometimes speak of an "archetypal" application. The *archetypal Libmarpa application* implements a complete logic flow, starting with the creation of a grammar, and proceeding all the way to the return of the final result from a value object. In the archetypal Libmarpa application, the grammar, input and semantics are all small but non-trivial.

10.2 Naming conventions

Methods in Libmarpa follow a strict naming convention. All methods have a name beginning with `marpa_`, if they are part of the external interface. If an external method is not a static method, its name is prefixed with one of `marpa_c_`, `marpa_g_`, `marpa_r_`, `marpa_b_`, `marpa_o_`, `marpa_t_` or `marpa_v_`, where the single letter between underscores is one of the Libmarpa major class abbreviations. The letter indicates which class the method belongs to.

Methods that are exported, but that are part of the internal interface, begin with `_marpa_`. Methods that are part of the internal interface (often called "internal methods") are subject to change and are intended for use only by Libmarpa's developers.

Libmarpa reserves the `marpa_` and `_marpa_` prefixes for itself, with all their capitalization variants. All Libmarpa names visible outside the package will begin with a capitalization variant of one of these two prefixes.

10.3 Return values

Some general conventions for return values are worth mentioning:

- For methods that return an integer, a return value of -1 usually indicates soft failure.
- For methods that return an integer, a return value of -2 usually indicates hard failure.
- For methods that return an integer, a return value greater of zero or more usually indicates success.
- If a method returns an pointer value, `NULL` usually indicates failure. Any other result usually indicates success.

The Libmarpa programmer should not overly rely on the general conventions for return values. In particular, -2 may sometimes be ambiguous — both a valid return value for success, and a potential indication of hard failure. In this case, the programmer must distinguish the two return statuses based on the error code, and a programmer who is relying too heavily on the general conventions will fall into a trap. For a the description of the return values of `marpa_g_rule_rank_set()`, see Section 13.7 [Rank methods], page 30.

10.4 How to read the method descriptions

The method descriptions are written on the assumption that the reader has the following in mind while reading them:

- Each method description begins with the signature of its "topic method".
- In the method description, the phrase `"this method"` always refers to the topic method.
- Whenever `"this method"` is the subject of a sentence in the method description, it may be elided, so that, for example, `"This method returns 42"` becomes `"Returns 42"`.
- If the return type of a method is not `void`, the last paragraph of its method description is a "return value summary". The return value summary starts with the label "**Return Value**".
- Every method returns in exactly one of three statuses: success, hard failure, or soft failure.
- A return status of hard failure indicates that the method detected a hard failure.
- A method may have several kinds of hard failure, including several kinds of irrecoverable hard failure and several kinds of recoverable hard failure. On return, these can be distinguished by their error codes.
- If a method call hard fails, its error code is that associated with the hard failure. Unless stated otherwise in the return value summary, the error code is set in the base grammar of the method call, and may be accessed with the methods described below. See Section 21.1 [Error methods], page 64.
- If a method allows a recoverable hard failure, this is explicitly stated in its return value summary, along with the associated error code. The method description with state the circumstances under which the recoverable hard failure occurs, and what the application must do to recover.
- A return status of soft failure indicates that the method detected a soft failure.
- Every method has at most one kind of soft failure.
- If a method allows a soft failure, this is explicitly stated in its return value summary, and the method description will state the circumstances under which the soft failure occurs, and what the application must do to recover.
- If a method call soft fails, the value of the error code is indeterminate.
- If a method call succeeds, the value of the error code is indeterminate.
- A return status of success indicates that the method did not detect any failures.
- If both a hard failure and a soft failure occur, the return status will be hard failure.
- If both a recoverable hard failure and an irrecoverable hard failure occur, the error code will be for an irrecoverable hard failure.
- The behaviors specified for success and soft failure are application behaviors.
- The behaviors specified for hard failures are diagnostic behaviors if an irrecoverable failure occurred, and application behaviors otherwise.

11 Static methods

`Marpa_Error_Code marpa_check_version` (*int* `required_major`, *int* `required_minor`, *int* `required_micro`) [Function]

[Accessor] Checks that the Marpa library in use is compatible with the given version. Generally, the application programmer will pass in the constants `MARPA_MAJOR_VERSION`, `MARPA_MINOR_VERSION`, and `MARPA_MICRO_VERSION` as the three arguments, to check that their application was compiled with headers the match the version of Libmarpa that they are using.

If *required_major.required_minor.required_micro* is an exact match with 9.0.2, the method succeeds. Otherwise the return status is an irrecoverable hard failure.

Return value: On success, `MARPA_ERR_NONE`. On hard failure, the error code.

`Marpa_Error_Code marpa_version` (*int* version*) [Function]

[Accessor] Writes the version number in *version*. It is an undetected irrecoverable hard failure if *version* does not have room for three `int`'s.

Return value: Always succeeds. The return value is indeterminate.

12 Configuration methods

The configuration object is intended for future extensions. These may allow the application to override Libmarpa's memory allocation and fatal error handling without resorting to global variables, and therefore in a thread-safe way. Currently, the only function of the `Marpa_Config` class is to give `marpa_g_new()` a place to put its error code.

`Marpa_Config` is Libmarpa's only "major" class which is not a time class. There is no constructor or destructor, although `Marpa_Config` objects **do** need to be initialized before use. Aside from its own accessor, `Marpa_Config` objects are only used by `marpa_g_new()` and no reference to their location is not kept in any of Libmarpa's time objects. The intent is to that it be convenient to have them in memory that might be deallocated soon after `marpa_g_new()` returns. For example, they could be put on the stack.

`int marpa_c_init ( Marpa_Config* config)` [Function]

[Mutator] Initialize the *config* information to "safe" default values. An irrecoverable error will result if an uninitialized configuration is used to create a grammar.

Return value: Always succeeds. The return value is indeterminate.

`Marpa_Error_Code marpa_c_error ( Marpa_Config* config, const char** p_error_string )` [Function]

[Accessor] Error codes are usually kept in the base grammar, which leaves `marpa_g_new()` no place to put its error code on failure. Objects of the `Marpa_Config` class provide such a place. *p_error_string* is reserved for use by the internals. Applications should set it to `NULL`.

Return value: The error code in *config*. Always succeeds, so that `marpa_c_error()` never requires an error code for itself.

13 Grammar methods

13.1 Overview

An archetypal application has a grammar. To create a grammar, use the `marpa_g_new()` method. When a grammar is no longer in use, its memory can be freed using the `marpa_g_unref()` method.

To be precomputed, a grammar must have one or more symbols. To create symbols, use the `marpa_g_symbol_new()` method.

To be precomputed, a grammar must have one or more rules. To create rules, use the `marpa_g_rule_new()` and `marpa_g_sequence_new()` methods.

For non-trivial parsing, one or more of the symbols must be terminals. To mark a symbol as a terminal, use the `marpa_g_symbol_is_terminal_set()` method.

To be precomputed, a grammar must have exactly one start symbol. To mark a symbol as the start symbol, use the `marpa_g_start_symbol_set()` method.

Before parsing with a grammar, it must be precomputed. To precompute a grammar, use the `marpa_g_precompute()` method.

13.2 Creating a new grammar

`Marpa_Grammar marpa_g_new ( Marpa_Config* configuration )` [Function]

[Constructor] Creates a new grammar time object. The returned grammar object is not yet precomputed, and will have no symbols and rules. Its reference count will be 1.

Unless the application calls `marpa_c_error()` Libmarpa will not reference the location pointed to by the *configuration* argument after `marpa_g_new()` returns. (See [marpa_c_error], page 22.) The *configuration* argument may be `NULL`, but if it is, there will be no way to determine the error code on failure.

Return value: On success, the grammar object. On hard failure, `NULL`. Also on hard failure, if the *configuration* argument is not `NULL`, the error code is set in *configuration*. The error code may be accessed using `marpa_c_error()`.

`int marpa_g_force_valued ( Marpa_Grammar g )` [Function]

[Mutator] It is recommended that this call be made immediately after the grammar constructor. It turns off a deprecated feature.

The `marpa_g_force_valued()` forces all the symbols in a grammar to be "valued". The opposite of a valued symbol is one about whose value you do not care. This distinction has been made in the past in hope of gaining efficiencies at evaluation time. Current thinking is that the gains do not repay the extra complexity.

Return value: On success, a non-negative integer, whose value is otherwise indeterminate. On failure, -2.

13.3 Tracking the reference count of the grammar

`Marpa_Grammar marpa_g_ref` (*Marpa_Grammar g*) [Function]

[Mutator] Increases the reference count of *g* by 1. Not needed by most applications.

Return value: On success, *g*. On hard failure, `NULL`.

`void marpa_g_unref` (*Marpa_Grammar g*) [Function]

[Destructor] Decreases the reference count by 1, destroying *g* once the reference count reaches zero.

13.4 Symbol methods

`Marpa_Symbol_ID marpa_g_start_symbol` (*Marpa_Grammar g*) [Function]

[Accessor] When successful, returns the ID of the start symbol. Soft fails, if there is no start symbol. The start symbol is set by the `marpa_g_start_symbol_set()` call.

Return value: On success, the ID of the start symbol, which is always a non-negative number. On soft failure, −1. On hard failure, −2.

`Marpa_Symbol_ID marpa_g_start_symbol_set` (*Marpa_Grammar g*, *Marpa_Symbol_ID* `sym_id`) [Function]

[Mutator] When successful, sets the start symbol of grammar *g* to symbol *sym_id*. Soft fails if *sym_id* is well-formed (a non-negative integer), but a symbol with that ID does not exist.

Return value: On success, *sym_id*, which will always be a non-negative number. On soft failure, −1. On hard failure, −2.

`int marpa_g_highest_symbol_id` (*Marpa_Grammar g*) [Function]

[Accessor] **Return value**: On success, the numerically largest symbol ID of *g*. On hard failure, −2.

`int marpa_g_symbol_is_accessible` (*Marpa_Grammar g*, *Marpa_Symbol_ID* `sym_id`) [Function]

[Accessor] A symbol is *accessible* if it can be reached from the start symbol. Soft fails if *sym_id* is well-formed (a non-negative integer), but a symbol with that ID does not exist. A common hard failure is calling this method with a grammar that is not precomputed.

Return value: On success, 1 if symbol *sym_id* is accessible, 0 if not. On soft failure, −1. On hard failure, −2.

`int marpa_g_symbol_is_nullable` (*Marpa_Grammar g*, *Marpa_Symbol_ID sym_id*) [Function]

[Accessor] A symbol is *nullable* if it sometimes produces the empty string. A **nulling** symbol is always a **nullable** symbol, but not all **nullable** symbols are **nulling** symbols. Soft fails if *sym_id* is well-formed (a non-negative integer), but a symbol with that ID does not exist. A common hard failure is calling this method with a grammar that is not precomputed.

Return value: On success, 1 if symbol *sym_id* is nullable, 0 if not. On soft failure, −1. On hard failure, −2.

`int marpa_g_symbol_is_nulling` (*Marpa_Grammar* `g`, *Marpa_Symbol_ID* `sym_id`) [Function]

[Accessor] A symbol is *nulling* if it always produces the empty string. Soft fails if *sym_id* is well-formed (a non-negative integer), but a symbol with that ID does not exist. A common hard failure is calling this method with a grammar that is not precomputed.

Return value: On success, 1 if symbol *sym_id* is nulling, 0 if not. On soft failure, −1. On hard failure, −2.

`int marpa_g_symbol_is_productive` (*Marpa_Grammar* `g`, *Marpa_Symbol_ID* `sym_id`) [Function]

[Accessor] A symbol is *productive* if it can produce a string of terminals. All nullable symbols are considered productive. Soft fails if *sym_id* is well-formed (a non-negative integer), but a symbol with that ID does not exist. A common hard failure is calling this method with a grammar that is not precomputed.

Return value: On success, 1 if symbol *sym_id* is productive, 0 if not. On soft failure, −1. On hard failure, −2.

`int marpa_g_symbol_is_start` (*Marpa_Grammar* `g`, *Marpa_Symbol_ID* `sym_id`) [Function]

[Accessor] On success, if *sym_id* is the start symbol, returns 1. On success, if *sym_id* is not the start symbol, returns 0. On success, if no start symbol has been set, returns 0. is the start symbol.

Soft fails if *sym_id* is well-formed (a non-negative integer), but a symbol with that ID does not exist.

Return value: On success, 1 or 0. On soft failure, −1. On hard failure, −2.

`int marpa_g_symbol_is_terminal` (*Marpa_Grammar* `g`, *Marpa_Symbol_ID* `sym_id`) [Function]

[Accessor] On succcess, returns the "terminal status" of a *sym_id*. The terminal status is 1 if *sym_id* is a terminal, 0 otherwise. To be used as an input symbol in the `marpa_r_alternative()` method, a symbol must be a terminal.

By default, a symbol is a terminal if and only if it does not appear on the LHS of any rule. The terminal status can be set explicitly with the `marpa_g_symbol_is_terminal_set()` method. See [marpa_g_symbol_is_terminal_set], page 25.

Soft fails if *sym_id* is well-formed (a non-negative integer), but a symbol with that ID does not exist.

Return value: On success, 1 or 0. On soft failure, −1. On hard failure, −2.

`int marpa_g_symbol_is_terminal_set` (*Marpa_Grammar* `g`, *Marpa_Symbol_ID* `sym_id`, *int* `value`) [Function]

[Mutator] Sets the "terminal status" of a symbol. This function flags symbol *sym_id* as a terminal if *value* is 1, or flags it as a non-terminal if *value* is 0. To be used as an input symbol in the `marpa_r_alternative()` method, a symbol must be a terminal. On success, this method returns *value*.

Once set to a value with this method, the terminal status of a symbol is "locked" at that value. A subsequent call to this method that attempts to change the terminal

status of *sym_id* to a value different from its current one will hard fail with error code `MARPA_ERR_TERMINAL_IS_LOCKED`. Other hard failures include when *value* is not 0 or 1; and when the grammar *g* is precomputed.

By default, a symbol is a terminal if and only if it does not appear on the LHS of any rule. An attempt to flag a nulling symbol as a terminal will cause a failure, but this is not necessarily detected before precomputation.

Return value: On success, *value*, which will be 1 or 0. On soft failure, −1. On hard failure, −2.

`Marpa_Symbol_ID marpa_g_symbol_new` (*Marpa_Grammar g*) [Function]

[Mutator] When successful, creates a new symbol in grammar *g*.

Return value: On success, the ID of the new symbol; which will be a non-negative integer. On hard failure, −2.

13.5 Rule methods

`int marpa_g_highest_rule_id` (*Marpa_Grammar g*) [Function]

[Accessor] **Return value**: On success, the numerically largest rule ID of *g*. On hard failure, −2.

`int marpa_g_rule_is_accessible` (*Marpa_Grammar g*, *Marpa_Rule_ID* `rule_id`) [Function]

[Accessor] A rule is *accessible* if it can be reached from the start symbol. A rule is accessible if and only if its LHS symbol is accessible. The start rule is always an accessible rule.

Soft fails if *rule_id* is well-formed (a non-negative integer), but a rule with that ID does not exist. A common hard failure is calling this method with a grammar that is not precomputed.

Return value: On success 1 or 0: 1 if rule with ID *rule_id* is accessible, 0 if not. On soft failure, −1. On hard failure, −2.

`int marpa_g_rule_is_nullable` (*Marpa_Grammar g*, *Marpa_Rule_ID* `ruleid`) [Function]

[Accessor] A rule is *nullable* if it sometimes produces the empty string. A **nulling** rule is always a **nullable** rule, but not all **nullable** rules are **nulling** rules.

Soft fails if *rule_id* is well-formed (a non-negative integer), but a rule with that ID does not exist. A common hard failure is calling this method with a grammar that is not precomputed.

Return value: On success 1 or 0: 1 if the rule with ID *rule_id* is nullable, 0 if not. On soft failure, −1. On hard failure, −2.

`int marpa_g_rule_is_nulling` (*Marpa_Grammar g*, *Marpa_Rule_ID* `ruleid`) [Function]

[Accessor] A rule is *nulling* if it always produces the empty string.

Soft fails if *rule_id* is well-formed (a non-negative integer), but a rule with that ID does not exist. A common hard failure is calling this method with a grammar that is not precomputed.

Return value: On success 1 or 0: 1 if the rule with ID *rule_id* is nulling, 0 if not. On soft failure, −1. On hard failure, −2.

`int marpa_g_rule_is_loop` (*Marpa_Grammar g*, *Marpa_Rule_ID* `rule_id`) [Function]

[Accessor] A rule is a loop rule if it non-trivially produces the string of length one which consists only of its LHS symbol. Such a derivation takes the parse back to where it started, hence the term "loop". "Non-trivially" means the zero-step derivation does not count — the derivation must have at least one step.

The presence of a loop rule makes a grammar infinitely ambiguous, and applications will typically want to treat them as fatal errors. But nothing forces an application to do this, and Marpa will successfully parse and evaluate grammars with loop rules.

Soft fails if *rule_id* is well-formed (a non-negative integer), but a rule with that ID does not exist. A common hard failure is calling this method with a grammar that is not precomputed.

Return value: On success 1 or 0: 1 if the rule with ID *rule_id* is a loop rule, 0 if not. On soft failure, −1. On hard failure, −2.

`int marpa_g_rule_is_productive` (*Marpa_Grammar g*, *Marpa_Rule_ID* `rule_id`) [Function]

[Accessor] A rule is *productive* if it can produce a string of terminals. A rule is productive if and only if all the symbols on its RHS are productive. The empty string counts as a string of terminals, so that a nullable rule is always a productive rule. For that same reason, an empty rule is considered productive.

Soft fails if *rule_id* is well-formed (a non-negative integer), but a rule with that ID does not exist. A common hard failure is calling this method with a grammar that is not precomputed.

Return value: On success 1 or 0: 1 if the rule with ID *rule_id* is productive, 0 if not. On soft failure, −1. On hard failure, −2.

`int marpa_g_rule_length` (*Marpa_Grammar g*, *Marpa_Rule_ID* `rule_id`) [Function]

[Accessor] The length of a rule is the number of symbols on its RHS.

Soft fails if *rule_id* is well-formed (a non-negative integer), but a rule with that ID does not exist.

Return value: On success, the length of the rule with ID *rule_id*. On soft failure, −1. On hard failure, −2.

`Marpa_Symbol_ID marpa_g_rule_lhs` (*Marpa_Grammar g*, *Marpa_Rule_ID* `rule_id`) [Function]

[Accessor] Soft fails if *rule_id* is well-formed (a non-negative integer), but a rule with that ID does not exist.

Return value: On success, the ID of the LHS symbol of the rule with ID *rule_id*. On soft failure, −1. On hard failure, −2.

`Marpa_Rule_ID marpa_g_rule_new` (*Marpa_Grammar* `g`, *Marpa_Symbol_ID* `lhs_id`, *Marpa_Symbol_ID* *`rhs_ids`, *int* `length`) [Function]

[Mutator] On success, creates a new external BNF rule in grammar *g*. The ID of the new rule will be a non-negative integer, which will be unique to that rule. In addition to BNF rules, Marpa also allows sequence rules, which are created by the `marpa_g_sequence_new()` method. See [marpa_g_sequence_new], page 29.

Sequence rules and BNF rules are both rules: They share the same series of rule IDs, and are accessed and manipulated by the same methods, with the only differences being as noted in the descriptions of those methods.

The LHS symbol is *lhs_id*, and there are *length* symbols on the RHS. The RHS symbols are in an array pointed to by *rhs_ids*.

Possible hard failures, with their error codes, include:

- `MARPA_ERR_SEQUENCE_LHS_NOT_UNIQUE`: The LHS symbol is the same as that of a sequence rule.
- `MARPA_ERR_DUPLICATE_RULE`: The new rule would duplicate another BNF rule. Another BNF rule is considered the duplicate of the new one, if its LHS symbol is the same as symbol *lhs_id*, if its length is the same as *length*, and if its RHS symbols match one for one those in the array of symbols *rhs_ids*.

Return value: On success, the ID of the new external rule. On hard failure, −2.

`Marpa_Symbol_ID marpa_g_rule_rhs` (*Marpa_Grammar* `g`, *Marpa_Rule_ID* `rule_id`, *int* `ix`) [Function]

[Accessor] When successful, returns the ID of the symbol at index *ix* in the RHS of the rule with ID *rule_id*. The indexing of RHS symbols is zero-based.

Soft fails if *rule_id* is well-formed (a non-negative integer), but a rule with that ID does not exist.

A common hard failure is for *ix* not to be a valid index of the RHS. This happens if *ix* is less than zero, or or if *ix* is greater than or equal to the length of the rule.

Return value: On success, a symbol ID, which is always non-negative. On soft failure, −1. On hard failure, −2.

13.6 Sequence methods

`int marpa_g_rule_is_proper_separation` (*Marpa_Grammar* `g`, *Marpa_Rule_ID* `rule_id`) [Function]

[Accessor] When successful, returns

- 1 if *rule_id* is the ID of a sequence rule whose proper separation flag is set,
- 0 if *rule_id* is the ID of a sequence rule whose proper separation flag is not set,
- 0 if *rule_id* is the ID of a rule that is not a sequence rule.

Does not distinguish sequence rules without proper separation from non-sequence rules. That is, does not distinguish an unset proper separation flag from a proper separation flag which value is undefined because *rule_id* is the ID of a BNF rule. Applications which want to determine whether or not a rule is a sequence rule can use `marpa_g_sequence_min()` to do this. See [marpa_g_sequence_min], page 29.

Soft fails if *rule_id* is well-formed (a non-negative integer), but a rule with that ID does not exist.

Return value: On success, 1 or 0. On soft failure, −1. On hard failure, −2.

`int marpa_g_sequence_min` (*Marpa_Grammar* `g`, *Marpa_Rule_ID* `rule_id`) [Function]

[Accessor] On success, returns the mininum length of a sequence rule. Soft fails if a rule with ID *rule_id* exists, but is not a sequence rule. This soft failure can used to test whether or not a rule is a sequence rule.

Hard fails irrecoverably if *rule_id* is not well-formed (a non-negative number). Also, hard fails irrecoverably if no rule with ID *rule_id* exists, even when *rule_id* is well formed. Note that, in its handling of the non-existence of a rule for its rule argument, this method differs from many of the other grammar methods. Grammar methods which take a rule ID argument more often treat the non-existence of rule for a well-formed rule ID as a soft, recoverable, failure.

Return value: On success, the minimum length of the sequence rule with ID *rule_id*, which is always non-negative. On soft failure, −1. On hard failure, −2.

`Marpa_Rule_ID marpa_g_sequence_new` (*Marpa_Grammar* `g`, *Marpa_Symbol_ID* `lhs_id`, *Marpa_Symbol_ID* `rhs_id`, *Marpa_Symbol_ID* `separator_id`, *int* `min`, *int* `flags`) [Function]

[Mutator] When successful, adds a new sequence rule to grammar *g*, and return its ID. The ID of the sequence rule will be a non-negative integer, which is unique to that rule. All rules are numbered in the same series, so that a BNF rule will never have the same rule ID as a sequence rule, and vice versa.

Sequence rules are "sugar" — their presence in the Libmarpa interface does not extend its power. Every Libmarpa grammar which can be written using sequence rules can be rewritten as a grammar without sequence rules.

The LHS of the sequence is *lhs_id*, and the item to be repeated on the RHS of the sequence is *rhs_id*. The sequence must be repeated at least *min* times, where *min* is 0 or 1. If *separator_id* is non-negative, it is a separator symbol.

The LHS symbol cannot be the LHS of any other rule, whether a BNF rule or a sequence rule. On an attempt to create an sequence rule with a duplicate LHS, this method hard fails, with an error code of `MARPA_ERR_SEQUENCE_LHS_NOT_UNIQUE`.

The sequence RHS, or item, is restricted to a single symbol, and that symbol cannot be nullable. If *separator_id* is a symbol, it also cannot be a nullable symbol. Nullables on the RHS of sequence rules are prohibited because it is not completely clear what an application intends when it asks for a sequence of items, some of which are nullable — the most natural interpretation of this usually results in a highly ambiguous grammar.

Libmarpa allows highly ambiquous grammars and a programmer who wants a grammar with sequences containing nullable items or separators can can write that grammar using BNF rules. The use of BNF rules make it clearer that ambiguity is what the programmer intended, and allows the programmer more flexibility.

If `flags & MARPA_PROPER_SEPARATION` is non-zero, separation is "proper", that is, a trailing separator is not allowed. The term *proper* is based on the idea that properly-speaking, separators should actually separate items. Proper separation has no effect

at the Libmarpa level — it is tracked as a convenience for the higher-level interfaces to Libmarpa, which may want to offer the ability to discard separators in the semantics. (Some higher-level interfaces, in fact, may choose to discard separation by default.) At the Libmarpa level, sequences always "keep separators".

Return value: On success, the ID of the newly added sequence rule, which is always non-negative. On hard failure, −2.

`int marpa_g_sequence_separator` (*Marpa_Grammar* `g`, *Marpa_Rule_ID* `rule_id`) [Function]

[Accessor] On success, returns the symbol ID of the separator of the sequence rule with ID *rule_id*. Soft fails if there is no separator. The causes of hard failure include *rule_id* not being well-formed; *rule_id* not being the ID of a rule which exists; and *rule_id* not being the ID a sequence rule.

Return value: On success, a symbol ID, which is always non-negative. On soft failure, −1. On hard failure, −2.

`int marpa_g_symbol_is_counted` (*Marpa_Grammar* `g`, *Marpa_Symbol_ID* `sym_id`) [Function]

[Accessor] On success, returns a boolean whose value is 1 iff the symbol with ID *sym_id* is counted. A symbol is *counted* iff

- it appears on the RHS of a sequence rule, or
- it is used as the separator symbol of a sequence rule.

Soft fails iff *sym_id* is well-formed (a non-negative integer), but a symbol with that ID does not exist.

Return value: On success, a boolean. On soft failure, −1. On hard failure, −2.

13.7 Rank methods

`Marpa_Rank marpa_g_rule_rank` (*Marpa_Grammar* `g`, *Marpa_Rule_ID rule_id*) [Function]

[Accessor] When successful, returns the rank of the rule with ID *rule_id*. When a rule is created, its rank is initialized to the default rank of the grammar. The default rank of the grammar is 0.

Return value: On success, returns a rule rank, and sets the error code to `MARPA_ERR_NONE`. The rule rank is an integer. On hard failure, returns −2, and sets the error code to an appropriate value, which will never be `MARPA_ERR_NONE`. Note that −2 is a valid rule rank, so that when −2 is returned, the error code is the only way to distinguish success from failure. The error code can be determined using `marpa_g_error()`. See [marpa_g_error], page 64.

`Marpa_Rank marpa_g_rule_rank_set` (*Marpa_Grammar* `g`, *Marpa_Rule_ID* `rule_id`, *Marpa_Rank* `rank`) [Function]

[Mutator] When successful, sets the rank of the rule with ID *rule_id* to *rank* and returns *rank*.

Return value: On success, returns *rank*, which will be an integer, and sets the error code to `MARPA_ERR_NONE`. On hard failure, returns −2, and sets the error code to

an appropriate value, which will never be `MARPA_ERR_NONE`. Note that −2 is a valid rule rank, so that when −2 is returned, the error code is the only way to distinguish success from failure. The error code can be determined using `marpa_g_error()`. See [marpa_g_error], page 64.

`int marpa_g_rule_null_high` (*Marpa_Grammar g*, *Marpa_Rule_ID rule_id*) [Function]

[Accessor] On success, returns a boolean whose value is 1 iff "null ranks high" is set in the rule with ID *rule_id*. When a rule is created, it has "null ranks high" set.

For more on the "null ranks high" setting, read the description of `marpa_g_rule_null_high_set()`. See [marpa_g_rule_null_high_set], page 31.

Soft fails iff *rule_id* is well-formed (a non-negative integer), but a rule with that ID does not exist.

Return value: On success, a boolean. On soft failure, −1. On hard failure, −2.

`int marpa_g_rule_null_high_set` (*Marpa_Grammar g*, *Marpa_Rule_ID* `rule_id`, *int* `flag`) [Function]

[Mutator] On success,

- sets "null ranks high" in the rule with ID *rule_id* if the value of the boolean *flag* is 1;
- unsets "null ranks high" in the rule with ID *rule_id* if the value of the boolean *flag* is 0; and
- returns *flag*.

The "null ranks high" setting affects the ranking of rules with properly nullable symbols on their right hand side. If a rule has properly nullable symbols on its RHS, each instance in which it appears in a parse will have a pattern of nulled and non-nulled symbols. Such a pattern is called a "null variant".

If the "null ranks high" is set, nulled symbols rank high. If the "null ranks high" is unset is the default), nulled symbols rank low. Ranking of a null variants is done from left-to-right.

Soft fails iff *rule_id* is well-formed (a non-negative integer), but a rule with that ID does not exist.

Hard fails if the grammar has been precomputed.

Return value: On success, a boolean. On soft failure, −1. On hard failure, −2.

13.8 Precomputing the Grammar

`int marpa_g_has_cycle` (*Marpa_Grammar g*) [Function]

[Accessor] On success, returns a boolean which is 1 iff *g* has a cycle. Cycles make a grammar infinitely ambiguous, and are considered useless in current practice. Cycles make processing the grammar less efficient, sometimes considerably so. Applications will almost always want to treat cycles as mistakes on the part of the writer of the grammar. To determine which rules are in the cycle, `marpa_g_rule_is_loop()` can be used.

Return value: On success, a boolean. On hard failure, −2.

`int marpa_g_is_precomputed` (*Marpa_Grammar g*) [Function]

[Accessor] **Return value**: On success, a boolean which is 1 iff grammar *g* is precomputed. On hard failure, −2.

`int marpa_g_precompute` (*Marpa_Grammar g*) [Function]

[Mutator] On success, and on fully recoverable hard failure, precomputes the grammar *g*. Precomputation involves running a series of grammar checks and "precomputing" some useful information which is kept internally to save repeated calculations. After precomputation, the grammar is "frozen" in many respects, and many grammar mutators which succeed before precomputation will cause hard failures after precomputation. Precomputation is necessary for a recognizer to be generated from a grammar.

When called, clears any events already in the event queue. May return one or more events. The types of event that this method may return are A `MARPA_EVENT_LOOP_RULES`, `MARPA_EVENT_COUNTED_NULLABLE`, `MARPA_EVENT_NULLING_TERMINAL`. All of these events occur only on failure. Applications must be prepared for this method to return additional events, including events which occur on success. Events may be queried using the `marpa_g_event()` method. See [marpa_g_event], page 56.

The fully recoverable hard failure is `MARPA_ERR_GRAMMAR_HAS_CYCLE`. Recall that for fully recoverable hard failures this method precomputes the grammar. Most appplications, however, will want to treat a grammar with cycles as if it were a library-recoverable error. A `MARPA_ERR_GRAMMAR_HAS_CYCLE` error occurs iff a `MARPA_EVENT_LOOP_RULES` event occurs. For more details on cycles, see [marpa_g_has_cycle], page 31.

The error code `MARPA_ERR_COUNTED_NULLABLE` is library-recoverable. This failure occurs when a symbol on the RHS of a sequence rule is nullable, which Libmarpa does not allow in a grammar. Error code `MARPA_ERR_COUNTED_NULLABLE` occurs iff one or more `MARPA_EVENT_COUNTED_NULLABLE` events occur. There is one `MARPA_EVENT_COUNTED_NULLABLE` event for every symbol which is a nullable on the right hand side of a sequence rule. An application may use these events to inform the user of the problematic symbols, and this detail may help the user fix the grammar.

The error code item `MARPA_ERR_NULLING_TERMINAL` is library-recoverable. This failure occurs when a nulling symbol is also flagged as a terminal. Since terminals cannot be of zero length, this is a logical impossibility, and Libmarpa does not allow nulling terminals in a grammar. Error code item `MARPA_ERR_NULLING_TERMINAL` occurs iff one or more `MARPA_EVENT_NULLING_TERMINAL` events occur. There is one `MARPA_EVENT_NULLING_TERMINAL` events for every nulling terminal in the grammar. An application may use these events to inform the user of the problematic symbols, and this detail may help the user fix the grammar.

Among the other error codes which may case this method to fail are the following:

- `MARPA_ERR_NO_RULES`: The grammar has no rules.
- `MARPA_ERR_NO_START_SYMBOL`: No start symbol was specified.
- `MARPA_ERR_INVALID_START_SYMBOL`: A start symbol ID was specified, but it is not the ID of a valid symbol.
- `MARPA_ERR_START_NOT_LHS`: The start symbol is not on the LHS of any rule.

- `MARPA_ERR_UNPRODUCTIVE_START`: The start symbol is not productive.

More details of these can be found under the description of the appropriate code. See Section 21.3 [External error codes], page 64.

Return value: On success, a non-negative number, whose value is otherwise indeterminate. On hard failure, -2. For the error code `MARPA_ERR_GRAMMAR_HAS_CYCLE`, the hard failure is fully recoverable. For the error codes `MARPA_ERR_COUNTED_NULLABLE` and `MARPA_ERR_NULLING_TERMINAL`, the hard failure is library-recoverable.

14 Recognizer methods

14.1 Recognizer overview

An archetypal application uses a recognizer to read input. To create a recognizer, use the `marpa_r_new()` method. When a recognizer is no longer in use, its memory can be freed using the `marpa_r_unref()` method.

To make a recognizer ready for input, use the `marpa_r_start_input()` method.

The recognizer starts with its current earleme at location 0. To read a token at the current earleme, use the `marpa_r_alternative()` call.

To complete the processing of the current earleme, and move forward to a new one, use the `marpa_r_earleme_complete()` call.

14.2 Creating a new recognizer

`Marpa_Recognizer marpa_r_new` (*Marpa_Grammar g*) [Function]

[Constructor] On success, creates a new recognizer and increments the reference count of *g*, the base grammar, by one. In the new recognizer,

- the reference count will be 1;
- the furthest earleme will be 0; and
- latest and current earleme will be undefined.

Return value: On success, the newly created recognizer, which is never `NULL`. If *g* is not precomputed, or on other hard failure, `NULL`.

14.3 Keeping the reference count of a recognizer

`Marpa_Recognizer marpa_r_ref` (*Marpa_Recognizer r*) [Function]

[Mutator] Increases the reference count by 1. This method is not needed by most applications.

Return value: On success, the recognizer object, *r*, which is never `NULL`. On hard failure, `NULL`.

`void marpa_r_unref` (*Marpa_Recognizer r*) [Function]

[Destructor] Decreases the reference count by 1, destroying *r* once the reference count reaches zero. When *r* is destroyed, the reference count of its base grammar is decreased by one. If this takes the reference count of the base grammar to zero, the base grammar is also destroyed.

14.4 Life cycle mutators

`int marpa_r_start_input` (*Marpa_Recognizer r*) [Function]

[Mutator] When successful, does the following:

- Readies *r* to accept input.

- Completes the first Earley set, which is the Earley set whose ID is 0 and which is located at earleme 0.
- Leaves the latest, current and furthest earlemes all at 0.
- Clears any events that were in the event queue before this method was called.
- If this method exhausts the parse, generates a `MARPA_EVENT_EXHAUSTED` event. See Chapter 6 [Exhaustion], page 10.
- May generate one or more `MARPA_EVENT_SYMBOL_NULLED`, `MARPA_EVENT_SYMBOL_PREDICTED`, or `MARPA_EVENT_SYMBOL_EXPECTED` events. See Chapter 20 [Events], page 56.

Return value: On success, a non-negative value, whose value is otherwise indeterminate. On hard failure, −2.

`int marpa_r_alternative` (*Marpa_Recognizer* `r`, *Marpa_Symbol_ID* `token_id`, *int* `value`, *int* `length`) [Function]

The *token_id* argument must be the symbol ID of a terminal. The *value* argument is an integer that represents the "value" of the token, and which should not be zero. The *length* argument is the length of the token, which must be greater than zero.

On success, does the following, where *current* is the value of the current earleme before the call and *furthest* is the value of the furthest earleme before the call:

- Reads a new token into *r*. The symbol ID of the token will be *token_id*. The token will start at *current* and end at `current+length`.
- Sets the value of the furthest earleme to `max(current+length,furthest)`.
- Leaves the values of the latest and current earlemes unchanged.

After recoverable failure, the following are the case:

- The tokens read into *r* are unchanged. Specifically, no new token has been read into *r*.
- The values of the latest, current and furthest earlemes are unchanged.

Libmarpa allows tokens to be ambiguous. Two tokens are ambiguous if they end at the same earleme location. If two tokens are ambiguous, Libmarpa will attempt to produce all the parses that include either of them.

Libmarpa allows tokens to overlap. Let the notation *t*@*s*-*e* indicate that token *t* starts at earleme *s* and ends at earleme *e*. Let *t1*@*s1*-*e1* and *t2*@*s2*-*e2* be two tokens such that *s1*<=*s2*. We say that *t1* and *t2* overlap iff *e1*>*s2*.

The *value* argument is not used inside Libmarpa — it is simply stored to be returned by the valuator as a convenience for the application. In applications where the token's actual value is not an integer, it is expected that the application will use *value* as a "virtual" value, perhaps finding the actual value by using *value* to index an array. Some applications may prefer to track token values on their own, perhaps based on the earleme location and *token_id*, instead of using Libmarpa's token values.

A *value* of 0 does not cause a failure, but it is reserved for unvalued symbols, a now-deprecated feature. See Section 25.1 [Valued and unvalued symbols], page 82.

Hard fails irrecoverably with `MARPA_ERR_DUPLICATE_TOKEN` if the token added would be a duplicate. Two tokens are duplicates iff all of the following are true:

- They would have the same start earleme. In other words, if `marpa_r_alternative()` attempts to read them while at the same current earleme.
- They have the same *token_id*.
- They have the same *length*.

If a token was not accepted because of its token ID, hard fails with the `MARPA_ERR_UNEXPECTED_TOKEN_ID`. This hard failure is fully recoverable so that, for example, the application may retry this method with different token IDs until it succeeds. These retries are efficient, and are quite useable as a parsing technique — so much so we have given the technique a name: *the Ruby Slippers*. The Ruby Slippers are used in several applications.

Return value: On success, `MARPA_ERR_NONE`. On failure, an error code other than `MARPA_ERR_NONE`. The hard failure for `MARPA_ERR_UNEXPECTED_TOKEN_ID` is fully recoverable.

`int marpa_r_earleme_complete` (*Marpa_Recognizer* `r`) [Function]

For the purposes of this method description, we define the following:

- *current* is the value of the current earleme before the call of `marpa_r_earleme_complete`.
- *latest* is the value of the latest earleme before the call of `marpa_r_earleme_complete`.
- An "expected" terminal is one expected at a current earleme, in the same sense that `marpa_r_terminal_is_expected()` determines if a terminal is "expected" at the current earleme. See [marpa_r_terminals_expected], page 40.
- An "anticipated" terminal is one that was accepted by the `marpa_r_alternative()` to end at an earleme after the current earleme. An anticipated terminal will have length greater than one. "Anticipated" terminals only occur if the application is using an advanced model of input. See Chapter 23 [Advanced input models], page 77.

On success, does the final processing for the current earleme, including the following:

- Advances the current earleme, incrementing its value by 1. That is, sets the current earleme to *`current`*`+1`.
- If any token was accepted at *current*, creates a new Earley set which will be the latest Earley set. After the call, the latest earleme will be equal to the new current earleme, *`current`*`+1`.
- If no token was accepted at *current*, no Earley set is created. After the call, the value of the latest earleme will be unchanged — that is, it will remain at *latest*. Success when no tokens were accepted at *current* can only occur if the application is using an advanced model of input. See Chapter 23 [Advanced input models], page 77.
- The value of the furthest earleme is never changed by a call to `marpa_r_earleme_complete()`.

- Clears the event queue of any events which occured before this method was called.
- May generate one or more `MARPA_EVENT_SYMBOL_COMPLETED`, `MARPA_EVENT_SYMBOL_NULLED`, `MARPA_EVENT_SYMBOL_PREDICTED`, or `MARPA_EVENT_SYMBOL_EXPECTED` events. See Chapter 20 [Events], page 56.
- If an application-settable threshold on the number of Earley items has been reached or exceeded, generates a `MARPA_EVENT_EARLEY_ITEM_THRESHOLD` event. Often, the application will want to treat this event as if it were a library-recoverable failure. See [marpa_r_earley_item_warning_threshold_set], page 39.
- If the parse is exhausted, triggers a `MARPA_EVENT_EXHAUSTED` event. Exhaustion on success only occurs if no terminals are expected at the current earleme after the call to this method (that is, at *`current`*`+1`) and no terminals are anticipated after *`current`*`+1`.

On hard failure with the code `MARPA_ERR_PARSE_EXHAUSTED`, does the following:

- Leaves the current earleme at *current*. The current earleme will be the same as the furthest earleme.
- The value of the furthest earleme is never changed by a call to `marpa_r_earleme_complete()`.
- Leaves the value of the latest earleme at *latest*. No new Earley set is created.
- Sets the parse exhausted, so that no more tokens will be accepted. See Chapter 6 [Exhaustion], page 10.
- Leaves the parse in a state where no terminals are expected or anticipated.
- Clears the event queue of any events which occured before the call to this method.
- Triggers a `MARPA_EVENT_EXHAUSTED` event and no others.
- Leaves valid any parses that were valid at the current or earlier earlemes. Processing with these can continue, and it for this reason that we consider hard failures with the code `MARPA_ERR_PARSE_EXHAUSTED` to be fully recoverable.

We note that exhaustion can occur when this method fails and when it succeeds. The distinction is that, on success, the call creates a new Earley set before becoming exhausted while, on failure, it becomes exhausted without creating a new Earley set.

Return value: On success, the number of events generated. On hard failure, -2. Hard failure with the code `MARPA_ERR_PARSE_EXHAUSTED` is fully recoverable.

14.5 Location accessors

`Marpa_Earleme marpa_r_current_earleme` (*Marpa_Recognizer* `r`) [Function]

Return value: If input has started, the current earleme. If input has not started, -1. Always succeeds.

`Marpa_Earleme marpa_r_earleme` (*Marpa_Recognizer* `r`, *Marpa_Earley_Set_ID* `set_id`) [Function]

In the default, token-stream model, Earley set ID and earleme are always equal, but this is not the case in other input models. (The ID of an Earley set ID is also called its ordinal.) If there is no Earley set whose ID is *set_id*, `marpa_r_earleme()` fails. If *set_id* was negative, the error code is set to `MARPA_ERR_INVALID_LOCATION`. If

set_id is greater than the ordinal of the latest Earley set, the error code is set to `MARPA_ERR_NO_EARLEY_SET_AT_LOCATION`.

At this writing, there is no method for the inverse operation (conversion of an earleme to an Earley set ID). One consideration in writing such a method is that not all earlemes correspond to Earley sets. Applications that want to map earlemes to Earley sets will have no trouble if they are using the standard input model — the Earley set ID is always exactly equal to the earleme in that model. For other applications that want an earleme-to-ID mapping, the most general method is create an ID-to-earleme array using the `marpa_r_earleme()` method and invert it.

Return value: On success, the earleme corresponding to Earley set *set_id*. On failure, −2.

`int marpa_r_earley_set_value` (*Marpa_Recognizer* **r**, *Marpa_Earley_Set_ID earley_set*) [Function]

Returns the integer value of *earley_set*. For more details, see the description of `marpa_r_earley_set_values()`.

Return value: On success, the value of *earley_set*. On failure, −2.

`int marpa_r_earley_set_values` (*Marpa_Recognizer* **r**, *Marpa_Earley_Set_ID earley_set*, *int* p_value*, *void** p_pvalue*) [Function]

If *p_value* is non-zero, sets the location pointed to by *p_value* to the integer value of the Earley set. Similarly, if *p_pvalue* is non-zero, sets the location pointed to by *p_pvalue* to the pointer value of the Earley set.

The "value" and "pointer" of an Earley set are an arbitrary integer and an arbitrary pointer that the application can use for its own purposes. In character-per-earleme input models, for example, the integer can be the codepoint of the current character. In a traditional token-per-earleme input model, they could be used to indicate the string value of the token – the pointer could point to the start of the string, and the integer could indicate its length.

The Earley set value and pointer can be set using the `marpa_r_latest_earley_set_values_set()` method. The Earley set integer value defaults to −1, and the pointer value defaults to `NULL`.

Return value: On success, returns a non-negative integer. On failure, returns −2.

`unsigned int marpa_r_furthest_earleme` (*Marpa_Recognizer* **r**) [Function]

Always returns the furthest earleme.

Return value: On success, the furthest earleme. Always succeeds.

`Marpa_Earley_Set_ID marpa_r_latest_earley_set` (*Marpa_Recognizer* **r**) [Function]

This method returns the Earley set ID (ordinal) of the latest Earley set. Applications that want the value of the latest earleme can convert this value using the `marpa_r_earleme()` method.

Return value: On success, the ID of the latest Earley set. Always succeeds.

`int marpa_r_latest_earley_set_value_set ( Marpa_Recognizer r, int value)` [Function]

Sets the integer value of the latest Earley set. For more details, see the description of `marpa_r_latest_earley_set_values_set()`.

Return value: On success, the new value of *earley_set*. On failure, −2.

`int marpa_r_latest_earley_set_values_set ( Marpa_Recognizer r, int value, void* pvalue)` [Function]

Sets the integer and pointer value of the latest Earley set. For more about the "integer value" and "pointer value" of an Earley set, see the description of the `marpa_r_earley_set_values()` method.

Return value: On success, returns a non-negative integer. On failure, returns −2.

14.6 Other parse status methods

`int marpa_r_earley_item_warning_threshold (Marpa_Recognizer r)` [Function]

Returns the Earley item warning threshold. See [marpa_r_earley_item_warning_threshold_set], page 39.

Return value: The Earley item warning threshold. Always succeeds.

`int marpa_r_earley_item_warning_threshold_set (Marpa_Recognizer r, int threshold)` [Function]

[Mutator] On success, sets the Earley item warning threshold. The *Earley item warning threshold* is a number that is compared with the count of Earley items in each Earley set. When it is matched or exceeded, a `MARPA_EVENT_EARLEY_ITEM_THRESHOLD` event is created. See [MARPA_EVENT_EARLEY_ITEM_THRESHOLD], page 62.

If *threshold* is zero or less, an unlimited number of Earley items will be allowed without warning. This will rarely be what the user wants.

By default, Libmarpa calculates a value based on the grammar. The formula Libmarpa uses is the result of some experience, and most applications will be happy with it.

What should be done when the threshold is exceeded, depends on the application, but exceeding the threshold means that it is very likely that the time and space resources consumed by the parse will prove excessive. This is often a sign of a bug in the grammar. Applications often will want to smoothly shut down the parse, in effect treating the `MARPA_EVENT_EARLEY_ITEM_THRESHOLD` event as equivalent to library-recoverable hard failure.

Return value: The value that the Earley item warning threshold has after the method call is finished. Always succeeds.

`int marpa_r_is_exhausted (Marpa_Recognizer r)` [Function]

A parser is "exhausted" if it cannot accept any more input. Both successful and failed parses can be exhausted. In many grammars, the parse is always exhausted as soon as it succeeds. Good parses may also exist at earlemes prior to the current one.

Return value: 1 if the parser is exhausted, 0 otherwise. Always succeeds.

`int marpa_r_terminals_expected` (*Marpa_Recognizer* `r`, *Marpa_Symbol_ID** `buffer`) [Function]

Returns a list of the ID's of the symbols that are acceptable as tokens at the current earleme. *buffer* is expected to be large enough to hold the result. This is guaranteed to be the case if the buffer is large enough to hold a number of `Marpa_Symbol_ID`'s that is greater than or equal to the number of symbols in the grammar.

Return value: On success, the number of `Marpa_Symbol_ID`'s in *buffer*. On failure, -2.

`int marpa_r_terminal_is_expected` (*Marpa_Recognizer* `r`, *Marpa_Symbol_ID* `symbol_id`) [Function]

Return values on success: If *symbol_id* is the ID of a valid terminal symbol that is expected at the current earleme, a number greater than zero. If *symbol_id* is the ID of a valid terminal symbol that is **not** expected at the current earleme, or if *symbol_id* is the ID of a valid symbol that is not a terminal, zero.

Failure cases: Returns -2 on failure. It is a failure if *symbol_id* is not the ID of a valid symbol.

15 Progress reports

An important advantage of the Marpa algorithm is the ability to easily get full information about the state of the parse.

To start a progress report, use the `marpa_r_progress_report_start()` command. Only one progress report can be in use at any one time.

To get the information in a progress report, it is necessary to step through the progress report items. To get the data for the current progress report item, and advance to the next one, use the `marpa_r_progress_item()` method.

To destroy a progress report, freeing the memory it uses, call the `marpa_r_progress_report_finish()` method.

`int marpa_r_progress_report_reset ( Marpa_Recognizer r)` [Function]

Resets the progress report. Assumes a report of the progress has already been initialized at some Earley set for recognizer *r*, with `marpa_r_progress_report_start()`. The reset progress report will be positioned before its first item.

Return value: On success, a non-negative value. On failure, −2.

`int marpa_r_progress_report_start ( Marpa_Recognizer r, Marpa_Earley_Set_ID set_id)` [Function]

Initializes a report of the progress at Earley set *set_id* for recognizer *r*. If a progress report already exists, it is destroyed and its memory is freed. Initially, the progress report is positioned before its first item.

If no Earley set with ID *set_id* exists, `marpa_r_progress_report_start()` fails. The error code is `MARPA_ERR_INVALID_LOCATION` if *set_id* is negative. The error code is `MARPA_ERR_NO_EARLEY_SET_AT_LOCATION` if *set_id* is greater than the ID of the latest Earley set.

Return value: On success, the number of report items available. If the recognizer has not been started; if *set_id* does not exist; or on other failure, −2.

`int marpa_r_progress_report_finish ( Marpa_Recognizer r )` [Function]

Destroys the report of the progress at Earley set *set_id* for recognizer *r*, freeing the memory and other resources. It is often not necessary to call this method. Any previously existing progress report is destroyed automatically whenever a new progress report is started, and when the recognizer is destroyed.

Return value: −2 if no progress report has been started, or on other failure. On success, a non-negative value.

`Marpa_Rule_ID marpa_r_progress_item ( Marpa_Recognizer r, int* position, Marpa_Earley_Set_ID* origin )` [Function]

This method allows access to the data for the next item of a progress report. If there are no more progress report items, it returns −1 as a termination indicator and sets the error code to `MARPA_ERR_PROGRESS_REPORT_EXHAUSTED`. Either the termination indicator, or the item count returned by `marpa_r_progress_report_start()`, can be used to determine when the last item has been seen.

On success, the dot position is returned in the location pointed to by the *position* argument, and the origin is returned in the location pointed to by the *origin* argument. On failure, the locations pointed to by the *position* and *origin* arguments are unchanged.

Return value: On success, the rule ID of the next progress report item. If there are no more progress report items, -1. If either the *position* or the *origin* argument is `NULL`, or on other failure, -2.

16 Bocage methods

16.1 Overview

A bocage is structure containing the full set of parses found by processing the input according to the grammar. The bocage structure is new with Libmarpa, but is very similar in purpose to the more familar parse forests.

To create a bocage, use the `marpa_b_new()` method.

When a bocage is no longer in use, its memory can be freed using the `marpa_b_unref()` method.

16.2 Creating a new bocage

`Marpa_Bocage marpa_b_new` (*Marpa_Recognizer* `r`, *Marpa_Earley_Set_ID* `earley_set_ID`) [Function]

Creates a new bocage object, with a reference count of 1. The reference count of its parent recognizer object, *r*, is increased by 1. If *earley_set_ID* is -1, the Earley set at the current earleme is used, if there is one.

If *earley_set_ID* is -1 and there is no Earley set at the current earleme; or if *earley_set_ID* is -1 and there is no parse ending at Earley set *earley_set_ID*, `marpa_b_new()` fails and the error code is set to `MARPA_ERR_NO_PARSE`.

Success return value: On success, the new bocage object. On failure, `NULL`.

16.3 Reference counting

`Marpa_Bocage marpa_b_ref` (*Marpa_Bocage* `b`) [Function]

Increases the reference count by 1. Not needed by most applications.

Return value: On success, *b*. On failure, `NULL`.

`void marpa_b_unref` (*Marpa_Bocage* `b`) [Function]

Decreases the reference count by 1, destroying *b* once the reference count reaches zero. When *b* is destroyed, the reference count of its parent recognizer is decreased by 1. If this takes the reference count of the parent recognizer to zero, it too is destroyed. If the parent recognizer is destroyed, the reference count of its base grammar is decreased by 1. If this takes the reference count of the base grammar to zero, it too is destroyed.

16.4 Accessors

`int marpa_b_ambiguity_metric` (*Marpa_Bocage* `b`) [Function]

Returns an ambiguity metric. The metric is 1 is the parse is unambiguous. If the metric is 2 or greater, the parse is ambiguous. It was originally intended to have values greater than 2 be an cheaply computed estimate of the degree of ambiguity, but a satisfactory scheme for this has yet to be implemented.

Return value on success: 1 if the bocage is not for an ambiguous parse; 2 or greater if the bocage is for an ambiguous parse.

Failures: On failure, -2.

`int marpa_b_is_null` (*Marpa_Bocage* `b`) [Function]

Return value on success: A number greater than or equal to 1 if the bocage is for a null parse; otherwise, 0.

Failures: On failure, −2.

17 Ordering methods

17.1 Overview

Before iterating the parses in the bocage, they must be ordered. To create an ordering, use the `marpa_o_new()` method. When an ordering is no longer in use, its memory can be freed using the `marpa_o_unref()` method.

An ordering is *frozen* once the first tree iterator is created using it. A frozen ordering cannot be changed.

As of this writing, the only methods to order parses are internal and undocumented. This is expected to change.

17.2 Creating an ordering

`Marpa_Order marpa_o_new` (*Marpa_Bocage* `b`) [Function]

Creates a new ordering object, with a reference count of 1. The reference count of its parent bocage object, *b*, is increased by 1.

Return value: On success, the new ordering object. On failure, `NULL`.

17.3 Reference counting

`Marpa_Order marpa_o_ref` (*Marpa_Order* `o`) [Function]

Increases the reference count by 1. Not needed by most applications.

Return value: On success, *o*. On failure, `NULL`.

`void marpa_o_unref` (*Marpa_Order* `o`) [Function]

Decreases the reference count by 1, destroying *o* once the reference count reaches zero. Beginning with *o*'s parent bocage, Libmarpa then proceeds up the chain of parent objects. Every time a child is destroyed, the reference count of its parent is decreased by 1. Every time the reference count of an object is decreased by 1, if that reference count is now zero, that object is destroyed. Libmarpa follows this chain of decrements and destructions as required, all the way back to the base grammar, if necessary.

17.4 Accessors

`int marpa_o_ambiguity_metric` (*Marpa_Order* `o`) [Function]

Returns an ambiguity metric. The metric is 1 is the parse is unambiguous. If the metric is 2 or greater, the parse is ambiguous. It was originally intended to have values greater than 2 be an cheaply computed estimate of the degree of ambiguity, but a satisfactory scheme for this has yet to be implemented.

If the ordering is not already frozen, it will be frozen on return from `marpa_o_ambiguity_metric()`. `marpa_o_ambiguity_metric()` is considered an "accessor", because it is assumed that the ordering is frozen when `marpa_o_ambiguity_metric()` is called.

Return value on success: 1 if the ordering is not for an ambiguous parse; 2 or greater if the ordering is for an ambiguous parse.

Failures: On failure, −2.

`int marpa_o_is_null` (*Marpa_Order* `o`) [Function]
Return value on success: A number greater than or equal to 1 if the ordering is for a null parse; otherwise, 0.

Failures: On failure, −2.

17.5 Non-default ordering

`int marpa_o_high_rank_only_set` (*Marpa_Order* `o`, *int* `flag`) [Function]
`int marpa_o_high_rank_only` (*Marpa_Order* `o`) [Function]
These methods, respectively, set and query the "high rank only" flag of ordering *o*. A *flag* of 1 indicates that, when ranking, all choices should be discarded except those of the highest rank. A *flag* of 0 indicates that no choices should be discarded on the basis of their rank.

A value of 1 is the default. The value of the "high rank only" flag has no effect unless ranking has been turned on using the `marpa_o_rank()` method.

Return value: On success, the value of the "high rank only" flag **after** the call. On failure, −2.

`int marpa_o_rank` (*Marpa_Order* `o`) [Function]
By default, the ordering of parse trees is arbitrary. This method causes the ordering to be ranked according to the ranks of symbols and rules, the "null ranks high" flags of the rules, and the "high rank only" flag of the ordering. Once this method returns, the ordering is frozen.

Return value: On success, a non-negative value. On failure, −2.

18 Tree methods

18.1 Overview

Once the bocage has an ordering, the parses trees can be iterated. Marpa's *parse tree iterators* iterate the parse trees contained in a bocage object. In Libmarpa, "parse tree iterators" are usually just called *trees*.

To create a tree, use the `marpa_t_new()` method. A newly created tree iterator is positioned before the first parse tree. When a tree iterator is no longer in use, its memory can be freed using the `marpa_t_unref()` method.

To position a newly created tree iterator at the first parse tree, use the `marpa_t_next()` method. Once the tree iterator is positioned at a parse tree, the same `marpa_t_next()` method is used to position it to the next parse tree.

18.2 Creating a new tree iterator

`Marpa_Tree marpa_t_new` (*Marpa_Order* `o`) [Function]

Creates a new tree iterator, with a reference count of 1. The reference count of its parent ordering object, *o*, is increased by 1.

When initialized, a tree iterator is positioned before the first parse tree. To position the tree iterator to the first parse, the application must call `marpa_t_next()`.

Return value: On success, a newly created tree. On failure, `NULL`.

18.3 Reference counting

`Marpa_Tree marpa_t_ref` (*Marpa_Tree* `t`) [Function]

Increases the reference count by 1. Not needed by most applications.

Return value: On success, *t*. On failure, `NULL`.

`void marpa_t_unref` (*Marpa_Tree* `t`) [Function]

Decreases the reference count by 1, destroying *t* once the reference count reaches zero. Beginning with *t*'s parent ordering, Libmarpa then proceeds up the chain of parent objects. Every time a child is destroyed, the reference count of its parent is decreased by 1. Every time the reference count of an object is decreased by 1, if that reference count is now zero, that object is destroyed. Libmarpa follows this chain of decrements and destructions as required, all the way back to the base grammar, if necessary.

18.4 Iterating through the trees

`int marpa_t_next` (*Marpa_Tree* `t`) [Function]

Positions *t* at the next parse tree in the iteration. Tree iterators are initialized to the position before the first parse tree, so this method must be called before creating a valuator from a tree.

If a tree iterator is positioned after the last parse, the tree is said to be "exhausted". A tree iterator for a bocage with no parse trees is considered to be "exhausted"

when initialized. If the tree iterator is exhausted, `marpa_t_next()` returns -1 as a termination indicator, and sets the error code to `MARPA_ERR_TREE_EXHAUSTED`.

Return value: On success, a non-negative value. If the tree iterator is exhausted, -1. On failure, -2.

`int marpa_t_parse_count (` *Marpa_Tree* `t)` [Function]

The parse counter counts the number of parse trees traversed so far. The count includes the current iteration of the tree, so that a value of 0 indicates that the tree iterator is at its initialized position, before the first parse tree.

Return value: The number of parses traversed so far. Always succeeds.

19 Value methods

19.1 Overview

The archetypal application needs a value object (or *valuator*) to produce the value of the parse. To create a valuator, use the `marpa_v_new()` method. When a valuator is no longer in use, its memory can be freed using the `marpa_v_unref()` method.

The application is required to maintain the stack, and the application is also required to implement most of the semantics, including the evaluation of rules. Libmarpa's valuator provides instructions to the application on how to manipulate the stack. To iterate through this series of instructions, use the `marpa_v_step()` method.

When successful, `marpa_v_step()` returns the type of step. Most step types have values associated with them. To access these values use the methods described in the section Section 19.9 [Basic step accessors], page 54. How to perform the steps is described in the sections Section 19.2 [How to use the valuator], page 49, and Section 19.7 [Stepping through the valuator], page 53.

19.2 How to use the valuator

Libmarpa's valuator provides the application with "steps", which are instructions for stack manipulation. Libmarpa itself does not maintain a stack. This leaves the upper layer in total control of the stack and the values which are placed on it.

As example may make this clearer. Suppose the evalution is at a place in the parse tree where an addition is being performed. Libmarpa does not know that the operation is an addition. It will tell the application that rule number R is to be applied to the arguments at stack locations N and N+1, and that the result is to placed in stack location N.

In this system the application keeps track of the semantics for all rules, so it looks up rule R and determines that it is an addition. The application can do this by using R as an index into an array of callbacks, or by any other method it chooses. Let's assume a callback implements the semantics for rule R. Libmarpa has told the application that two arguments are available for this operation, and that they are at locations N and N+1 in the stack. They might be the numbers 42 and 711. So the callback is called with its two arguments, and produces a return value, let's say, 753. Libmarpa has told the application that the result belongs at location N in the stack, so the application writes 753 to location N.

Since Libmarpa knows nothing about the semantics, the operation for rule R could be string concatenation instead of addition. Or, if it is addition, it could allow for its arguments to be floating point or complex numbers. Since the application maintains the stack, it is up to the application whether the stack contains integers, strings, complex numbers, or polymorphic objects which are capable of being any of these things and more.

19.3 Advantages of step-driven valuation

Step-driven valuation hides Libmarpa's grammar rewrites from the application, and is quite efficient. Libmarpa knows which rules are sequences. Libmarpa optimizes stack manipulations based on this knowledge. Long sequences are very common in practical grammars.

For these, the stack manipulations suggested by Libmarpa's step-driven valuator will be significantly faster than the traditional stack evaluation algorithm.

Step-driven evalution has another advantage. To illustrate this, consider what is a very common case: The semantics are implemented in a higher-level language, using callbacks. If Libmarpa did not use step-driven valuation, it would need to provide for this case. But for generality, Libmarpa would have to deal in C callbacks. Therefore, a middle layer would have to create C language wrappers for the callbacks in the higher level language.

The implementation that results is this: The higher level language would need to wrap each callback in C. When calling Libmarpa, it would pass the wrappered callback. Libmarpa would then need to call the C language "wrappered" callback. Next, the wrapper would call the higher-level language callback. The return value, which would be data native to the higher-level language, would need to be passed to the C language wrapper, which will need to make arrangements for it to be based back to the higher-level language when appropriate.

A setup like this is not terribly efficient. And exception handling across language boundaries would be very tricky. But neither of these is the worst problem.

Callbacks are hard to debug. Wrappered callbacks are even worse. Calls made across language boundaries are harder yet to debug. In the system described above, by the time a return value is finally consumed, a language boundary will have been crossed four times.

How do Libmarpa users deal with difficulties like this? Usually, by doing the absolute minimum possible in the callbacks. A horrific debugging enviroment can become a manageable one if there is next to no code to be debugged. And this can be accomplished by doing as much as possible in pre- and post-processing.

In essence, callbacks force applications to do most of the programming via side effects. One need not be a functional programming purist to find this a very undesirable style of design to force on an application. But the ability to debug can make the difference between code that does work and code that does not. Unfairly or not, code is rarely considered well-designed when it does not work.

So, while step-driven valuation seems a roundabout approach, it is simpler and more direct than the likely alternatives. And there is something to be said for pushing semantics up to the higher levels — they can be expected to know more about it.

These advantages of step-driven valuation are strictly in the context of a low-level interface. The author is under no illusion that direct use of Libmarpa's valuator will be found satisfactory by most Libmarpa users, even those using the C language. The author certainly avoids using step-driven valuation directly. Libmarpa's valuator is intended to be used via an upper layer, one which **does** know about semantics.

19.4 Maintaining the stack

This section discusses in detail the requirements for maintaining the stack. In some cases, such as implementation using a Perl array, fulfilling these requirements is trivial. Perl auto-extends its arrays, and initializes the element values, on every read or write. For the C programmer, things are not quite so easy.

In this section, we will assume a C90 or C99 standard-conformant C application. This assumption is convenient on two grounds. First, this will be the intended use for many

readers. Second, standard-conformant C is a "worst case". Any issue faced by a programmer of another environment is likely to also be one that must be solved by the C programmer.

Libmarpa often optimizes away unnecessary stack writes to stack locations. When it does so, it will not necessarily optimize away all reads to that stack location. This means that a location's first access, as suggested by the Libmarpa step instructions, may be a read. This possibility requires a special awareness from the C programmer, as discussed in the sections Section 19.4.1 [Sizing the stack], page 51, and Section 19.4.2 [Initializing locations in the stack], page 51.

In the discussions in this document, stack locations are non-negative integers. The bottom of the stack is location 0. In moving from the bottom of the stack to the top, the numbers increase. Stack location Y is said to be "greater" than stack location X if stack location Y is closer to the top of stack than location X, and therefore stack locations are considered greater or lesser if the integers that represent them are greater or lesser. Another way to state that a stack location Y is greater (lesser) than stack location X is to say that a stack location Y is later (earlier) than stack location X.

19.4.1 Sizing the stack

If an implementation applies Libmarpa's step instructions literally, using a physical stack, it must make sure the stack is large enough. Specifically, the application must do the following

- Ensure location 0 exists — in other words that the stack is at least length 1.
- For `MARPA_STEP_TOKEN` steps, ensure that location `marpa_v_result(v)` exists.
- For `MARPA_STEP_NULLING_SYMBOL` steps, ensure that location `marpa_v_result(v)` exists.
- For `MARPA_STEP_RULE` steps, ensure that stack locations from `marpa_v_arg_0(v)` to `marpa_v_arg_n(v)` exist.

Three aspects of these requirements deserve special mention. First, note that the requirement for a `MARPA_STEP_RULE` is that the application size the stack to include the arguments to be read. Because stack writes may be optimized away, an application, when reading, cannot assume that the stack was sized appropriately by a prior write. The first access to a new stack location may be a read.

Second, note that there is no explicit requirement that the application size the stack to include the location for the result of the `MARPA_STEP_RULE` step. An application is allowed to assume that result will go into one of the locations that were read.

Third, special note should be made of the requirement that location 0 exist. By convention, the parse result resides in location 0 of the stack. Because of potential optimizations, an application cannot assume that it will receive a Libmarpa step instruction that either reads from or writes to location 0.

19.4.2 Initializing locations in the stack

Write optimizations also creates issues for implementations which require data to be initialized before reading. Every fully standard-conforming C application is such an implementation. Both C90 and C99 allow "trap values", and therefore conforming applications must be prepared for an uninitialized location to contain one of those. Reading a trap value may cause an abend. (It is safe, in standard-conforming C, to write to a location containing a trap value.)

The requirement that locations be initialized before reading occurs in other implementations. Any implementation that has a "universe" of "safe" values, may require special precautions. The required precautions may amount to a need to initialize "uninitialized" values. A practical example might be an implementation that expects all locations to contain a pointer which it can safely indirect from. In such implementations, just as in standard-conformant C, every stack location needs to be initialized before being read.

Due to write optimizations, an application cannot rely on Libmarpa's step instructions to initialize every stack location before its first read. One way to safely deal with the initialization of stack locations, is to do all of the following:

- When starting evaluation, ensure that the stack contains at least location 0.
- Also, when starting evaluation, initialize every location in the stack.
- Whenever the stack is extended, initialize every stack location added.

Applications which try to optimize out some of these initializations need to be aware that an application can never assume that activity in the stack is safely "beyond" an uninitialized location. Libmarpa steps often revisit earlier sections of the stack, and these revisits may include reads of previously unvisited stack locations.

19.5 Creating a new valuator

`Marpa_Value marpa_v_new ( Marpa_Tree t )` [Function]

Creates a new valuator. The parent object of the new valuator will be the tree iterator t, and the reference count of the new valuator will be 1. The reference count of t is increased by 1.

The parent tree iterator is "paused", so that the tree iterator cannot move on to a new parse tree until the valuator is destroyed. Many valuators of the same parse tree can exist at once. A tree iterator is "unpaused" when all of the valuators of that tree iterator are destroyed.

Return value: On success, the newly created valuator. On failure, `NULL`.

19.6 Reference counting

`Marpa_Value marpa_v_ref (Marpa_Value v)` [Function]

Increases the reference count by 1. Not needed by most applications.

Return value: On success, v. On failure, `NULL`.

`void marpa_v_unref ( Marpa_Value v)` [Function]

Decreases the reference count by 1, destroying v once the reference count reaches zero. Beginning with v's parent tree, Libmarpa then proceeds up the chain of parent objects. Every time a child is destroyed, the reference count of its parent is decreased by 1. Every time the reference count of an object is decreased by 1, if that reference count is now zero, that object is destroyed. Libmarpa follows this chain of decrements and destructions as required, all the way back to the base grammar, if necessary.

19.7 Stepping through the valuator

`Marpa_Step_Type marpa_v_step ( Marpa_Value v)` [Function]

This method "steps through" the valuator. The return value is a `Marpa_Step_Type`, an integer which indicates the type of step. How the application is expected to act on each step is described below (Section 19.8 [Valuator steps by type], page 53). When the iteration through the steps is finished, `marpa_v_step()` returns `MARPA_STEP_INACTIVE`.

Return value: On success, a `Marpa_Step_Type`, which always be a non-negative integer. On failure, −2.

19.8 Valuator steps by type

`Marpa_Step_Type MARPA_STEP_RULE` [Macro]

The semantics of a rule should be performed. The application can find the value of the rule's children in the stack locations from `marpa_v_arg_0(v)` to `marpa_v_arg_n(v)`. The semantics for the rule whose ID is `marpa_v_rule(v)` should be executed on these child values, and the result placed in `marpa_v_result(v)`. In the case of a `MARPA_STEP_RULE` step, the stack location of `marpa_v_result(v)` is guaranteed to be equal to `marpa_v_arg_0(v)`.

`Marpa_Step_Type MARPA_STEP_TOKEN` [Macro]

The semantics of a non-null token should be performed. The application's value for the token whose ID is `marpa_v_token(v)` should be placed in stack location `marpa_v_result(v)`. Its value according to Libmarpa will be in `marpa_v_token_value(v)`.

`Marpa_Step_Type MARPA_STEP_NULLING_SYMBOL` [Macro]

The semantics for a nulling symbol should be performed. The ID of the symbol is `marpa_v_symbol(v)` and its value should be placed in stack location `marpa_v_result(v)`.

`Marpa_Step_Type MARPA_STEP_INACTIVE` [Macro]

The valuator has gone through all of its steps and is now inactive. The value of the parse will be in stack location 0. Because of optimizations, it is possible for valuator to immediately became inactive — `MARPA_STEP_INACTIVE` could be both the first and last step.

`Marpa_Step_Type MARPA_STEP_INITIAL` [Macro]

The valuator is new and has yet to go through any steps.

`Marpa_Step_Type MARPA_STEP_INTERNAL1` [Macro]
`Marpa_Step_Type MARPA_STEP_INTERNAL2` [Macro]
`Marpa_Step_Type MARPA_STEP_TRACE` [Macro]

These step types are reserved for internal purposes.

19.9 Basic step accessors

The basic step accessors are so called because their information is basic to the stack manipulation. The basic step accessors are implemented as macros. They always succeed.

`int marpa_v_arg_0` (*Marpa_Value v*) [Macro]
: For a `MARPA_STEP_RULE` step, returns the stack location where the value of first child can be found.

`int marpa_v_arg_n` (*Marpa_Value v*) [Macro]
: For a `MARPA_STEP_RULE` step, returns the stack location where the value of the last child can be found.

`int marpa_v_result` (*Marpa_Value v*) [Macro]
: For `MARPA_STEP_RULE`, `MARPA_STEP_TOKEN`, and `MARPA_STEP_NULLING_SYMBOL` steps, returns the stack location where the result of the semantics should be placed.

`Marpa_Rule_ID marpa_v_rule` (*Marpa_Value v*) [Macro]
: For the `MARPA_STEP_RULE` step, returns the ID of the rule.

`Marpa_Step_Type marpa_v_step_type` (*Marpa_Value v*) [Macro]
: Returns the current step type: `MARPA_STEP_TOKEN`, `MARPA_STEP_RULE`, etc. Usually not needed since this is also the return value of `marpa_v_step()`.

`Marpa_Symbol_ID marpa_v_symbol` (*Marpa_Value v*) [Macro]
: For the `MARPA_STEP_NULLING_SYMBOL` step, returns the ID of the symbol. The value returned is the same as that returned by the `marpa_v_token()` macro.

`Marpa_Symbol_ID marpa_v_token` (*Marpa_Value v*) [Macro]
: For the `MARPA_STEP_TOKEN` step, returns the ID of the token. The value returned is the same as that returned by the `marpa_v_symbol()` macro.

`int marpa_v_token_value` (*Marpa_Value v*) [Macro]
: For the `MARPA_STEP_TOKEN` step, returns the integer which is (or which represents) the value of the token.

19.10 Other step accessors

This section contains the step accessors that are not basic to stack manipulation, but which provide other useful information about the parse. These step accessors are implemented as macros.

All of these accessors always succeed, but if called when they are irrelevant they return an unspecified value. In this context, an "unspecified value" is a value that is either -1 or the ID of a valid Earley set, but which is otherwise unpredictable.

`Marpa_Earley_Set_ID marpa_v_es_id` (*Marpa_Value v*) [Macro]
: Return value: If the current step type is `MARPA_STEP_RULE`, the Earley Set ordinal where the rule ends. If the current step type is `MARPA_STEP_TOKEN` or `MARPA_STEP_NULLING_SYMBOL`, the Earley Set ordinal where the symbol ends. If the current step type is anything else, an unspecified value.

`Marpa_Earley_Set_ID marpa_v_rule_start_es_id` (*Marpa_Value v*) [Macro]
Return value: If the current step type is `MARPA_STEP_RULE`, the Earley Set ordinal where the rule begins. If the current step type is anything else, an unspecified value.

`Marpa_Earley_Set_ID marpa_v_token_start_es_id` (*Marpa_Value v*) [Macro]
Return value: If the current step type is `MARPA_STEP_TOKEN` or `MARPA_STEP_NULLING_SYMBOL`, the Earley Set ordinal where the token begins. If the current step type is anything else, an unspecified value.

20 Events

20.1 Overview

Events are generated by the `marpa_g_precompute()`, `marpa_r_earleme_complete()`, and `marpa_r_start_input()` methods. The methods are called event-active. Event-active methods always clear all previous events, so that after an event-active method the only events available will be those generated by that method.

Some Libmarpa methods clear the event queue. The user is expected to query events immediately after the method that generated them. We note especially that events are kept in the base grammar, so that multiple recognizers using the same base grammar overwrite each other's events.

To find out how many events were generated by the last event-active method, use the `marpa_g_event_count()` method.

To query a specific event, use the `marpa_g_event()` and `marpa_g_event_value()` methods.

In reading this chapter, we will need to be aware that it contains a mixture of grammar and recognizer methods.

20.2 Basic event accessors

`Marpa_Event_Type marpa_g_event` (*Marpa_Grammar g*, *Marpa_Event** `event`, *int* `ix`) [Function]

On success, the type of the *ix*'th event is returned and the data for the *ix*'th event is placed in the location pointed to by *event*.

Event indexes are in sequence. Valid events will be in the range from 0 to *n*, where *n* is one less than the event count. The event count can be queried using the `marpa_g_event_count()` method.

Return value: On success, the type of event *ix*. If there is no *ix*'th event, if *ix* is negative, or on other failure, -2. On failure, the locations pointed to by *event* are not changed.

`int marpa_g_event_count` (*Marpa_Grammar g*) [Function]

Return value: On success, the number of events. On failure, -2.

`int marpa_g_event_value` (*Marpa_Event** `event`) [Macro]

This macro provides access to the "value" of the event. The semantics of the value varies according to the type of the event, and is described in the section on event codes (Section 20.7 [Event codes], page 62).

20.3 Completion events

`int marpa_g_completion_symbol_activate` (*Marpa_Grammar g*, *Marpa_Symbol_ID* `sym_id`, *int* `reactivate`) [Function]

Allows the user to deactivate and reactivate symbol completion events in the grammar. When a recognizer is created, the activation status of each of its events is initialized

to the activation status of that event in the base grammar. If *reactivate* is zero, the event is deactivated in the grammar. If *reactivate* is one, the event is activated in the grammar.

Symbol completion events are active by default if the symbol was set up for completion events in the grammar. If a symbol was not set up for completion events in the grammar, symbol completion events are inactive by default and any attempt to change that is a fatal error.

The activation status of a completion event in the grammar can only be changed if the symbol is marked as a completion event symbol in the grammar, and before the grammar is precomputed. However, if a symbol is marked as a completion event symbol in the recognizer, the completion event can be deactivated and reactivated in the recognizer.

Success cases: On success, the method returns the value of *reactivate*. The method succeeds trivially if the symbol is already set as indicated by *reactivate*.

Failure cases: If the active status of the completion event for *sym_id* cannot be set as indicated by *reactivate*, the method fails. On failure, -2 is returned.

`int marpa_r_completion_symbol_activate` (*Marpa_Recognizer* `r`, *Marpa_Symbol_ID* `sym_id`, *int* `reactivate`) [Function]

Allows the user to deactivate and reactivate symbol completion events in the recognizer. If *reactivate* is zero, the event is deactivated. If *reactivate* is one, the event is activated.

Symbol completion events are active by default if the symbol was set up for completion events in the grammar. If a symbol was not set up for completion events in the grammar, symbol completion events are inactive by default and any attempt to change that is a fatal error.

Success cases: On success, the method returns the value of *reactivate*. The method succeeds trivially if the symbol is already set as indicated by *reactivate*.

Failure cases: If the active status of the completion event for *sym_id* cannot be set as indicated by *reactivate*, the method fails. On failure, -2 is returned.

`int marpa_g_symbol_is_completion_event` (*Marpa_Grammar* `g`, *Marpa_Symbol_ID* `sym_id`) [Function]

`int marpa_g_symbol_is_completion_event_set` (*Marpa_Grammar* `g`, *Marpa_Symbol_ID* `sym_id`, *int* `value`) [Function]

Libmarpa can be set up to generate an `MARPA_EVENT_SYMBOL_COMPLETED` event whenever the symbol is completed. A symbol is said to be **completed** when a non-nulling rule with that symbol on its LHS is completed.

For completion events to occur, the symbol must be marked as a completion event symbol. The `marpa_g_symbol_is_completion_event_set()` function marks symbol *sym_id* as a completion event symbol if *value* is 1, and unmarks it it as a completion event symbol if *value* is 0. The `marpa_g_symbol_is_completion_event()` method returns the current value of the completion event marking for symbol *sym_id*.

Marking a completion event sets its activation status to on. Unmarking a completion event sets its activation status to off. The completion event marking cannot be changed once the grammar is precomputed.

If a completion event is marked, its activation status can be changed using the `marpa_g_completion_symbol_activate()` method. Note that, if a symbol is marked as a completion event symbol in the recognizer, its completion event can be deactivated and reactivated in the recognizer.

Nulled rules and symbols will never cause completion events. Nullable symbols may be marked as completion event symbols, but this will have an effect only when the symbol is not nulled. Nulling symbols may be marked as completion event symbols, but no completion events will ever be generated for a nulling symbol. Note that this implies at no completion event will ever be generated at earleme 0, the start of parsing.

Success: On success, 1 if symbol *sym_id* is a completion event symbol after the call, 0 otherwise.

Failures: If *sym_id* is well-formed, but there is no such symbol, -1. If the grammar *g* is precomputed; or on other failure, -2.

20.4 Symbol nulled events

`int marpa_g_nulled_symbol_activate` (*Marpa_Grammar* `g`, *Marpa_Symbol_ID* `sym_id`, *int* `reactivate`) [Function]

Allows the user to deactivate and reactivate symbol nulled events in the grammar. When a recognizer is created, the activation status of each of its events is initialized to the activation status of that event in the base grammar. If *reactivate* is zero, the event is deactivated in the grammar. If *reactivate* is one, the event is activated in the grammar.

Symbol nulled events are active by default if the symbol was set up for nulled events in the grammar. If a symbol was not set up for nulled events in the grammar, symbol nulled events are inactive by default and any attempt to change that is a fatal error.

The activation status of a nulled event in the grammar can only be changed if the symbol is marked as a nulled event symbol in the grammar, and before the grammar is precomputed. However, if a symbol is marked as a nulled event symbol in the recognizer, the nulled event can be deactivated and reactivated in the recognizer.

Success cases: On success, the method returns the value of *reactivate*. The method succeeds trivially if the symbol is already set as indicated by *reactivate*.

Failure cases: If the active status of the nulled event for *sym_id* cannot be set as indicated by *reactivate*, the method fails. On failure, -2 is returned.

`int marpa_r_nulled_symbol_activate` (*Marpa_Recognizer* `r`, *Marpa_Symbol_ID* `sym_id`, *int* `boolean`) [Function]

Allows the user to deactivate and reactivate symbol nulled events in the recognizer. If *boolean* is zero, the event is deactivated. If *boolean* is one, the event is activated.

Symbol nulled events are active by default if the symbol was set up for nulled events in the grammar. If a symbol was not set up for nulled events in the grammar, symbol nulled events are inactive by default and any attempt to change that is a fatal error.

Success cases: On success, the method returns the value of *boolean*. The method succeeds trivially if the symbol is already set as indicated by *boolean*.

Failure cases: If the active status of the nulled event for *sym_id* cannot be set as indicated by *boolean*, the method fails. On failure, −2 is returned.

`int marpa_g_symbol_is_nulled_event ( Marpa_Grammar g, Marpa_Symbol_ID sym_id)` [Function]

`int marpa_g_symbol_is_nulled_event_set ( Marpa_Grammar g, Marpa_Symbol_ID sym_id, int value)` [Function]

Libmarpa can set up to generate an `MARPA_EVENT_SYMBOL_NULLED` event whenever the symbol is nulled. A symbol is said to be **nulled** when a zero length instance of that symbol is recognized.

For nulled events to occur, the symbol must be marked as a nulled event symbol. The `marpa_g_symbol_is_nulled_event_set()` function marks symbol *sym_id* as a nulled event symbol if *value* is 1, and unmarks it it as a nulled event symbol if *value* is 0. The `marpa_g_symbol_is_nulled_event()` method returns the current value of the nulled event marking for symbol *sym_id*.

Marking a nulled event sets its activation status to on. Unmarking a nulled event sets its activation status to off. The nulled event marking cannot be changed once the grammar is precomputed.

If a nulled event is marked, its activation status can be changed using the `marpa_g_nulled_symbol_activate()` method. Note that, if a symbol is marked as a nulled event symbol in the recognizer, its nulled event can be deactivated and reactivated in the recognizer.

As a reminder, a symbol instance is a symbol at a specific location in the input, and with a specific length. Also, whenever a nulled symbol instance is recognized at a location, it is acceptable at that location, and vice versa.

When a symbol instance is recognized at a location, it will generate a nulled event or a prediction event, but never both. A symbol instance of zero length, when recognized at a location, generates a nulled event at that location, and does not generate a completion event. A symbol instance of non-zero length, when acceptable at a location, generates a completion event at that location, and does not generate a nulled event.

When a symbol instance is acceptable at a location, it will generate a nulled event or a prediction event, but never both. A symbol instance of zero length, when acceptable at a location, generates a nulled event at that location, and does not generate a prediction event. A symbol instance of non-zero length, when acceptable at a location, generates a prediction event at that location, and does not generate a nulled event.

While it is not possible for a **symbol instance** to generate both a nulled event and a completion event at a location, it is quite possible that a **symbol** might generate both kinds of event at that location. This is because multiple instances of the same symbol may be recognized at a given location, and these instances will have different lengths. If one instance is recognized at a given location as zero length and a second, non-zero-length, instance is recognized at the same location, the first will generate only nulled events, while the second will generate only completion events. For similar reasons, while a **symbol instance** will never generate both a null event and a prediction event at a location, multiple instances of the same symbol may do so.

Zero length derivations can be ambiguous. When a zero length symbol is recognized, all of its zero-length derivations are also considered to be recognized.

The `marpa_g_symbol_is_nulled_event_set()` method will mark a symbol as a nulled event symbol, even if the symbol is non-nullable. This is convenient, for example, for automatically generated grammars. Applications which wish to treat it as a failure if there is an attempt to mark a non-nullable symbol as a nulled event symbol, can check for this case using the `marpa_g_symbol_is_nullable()` method.

Success: On success, 1 if symbol *sym_id* is a nulled event symbol after the call, 0 otherwise.

Failures: If *sym_id* is well-formed, but there is no such symbol, -1. If the grammar *g* is precomputed; or on other failure, -2.

20.5 Prediction events

`int marpa_g_prediction_symbol_activate ( Marpa_Grammar g, Marpa_Symbol_ID sym_id, int reactivate )` [Function]

Allows the user to deactivate and reactivate symbol prediction events in the grammar. When a recognizer is created, the activation status of each of its events is initialized to the activation status of that event in the base grammar. If *reactivate* is zero, the event is deactivated in the grammar. If *reactivate* is one, the event is activated in the grammar.

Symbol prediction events are active by default if the symbol was set up for prediction events in the grammar. If a symbol was not set up for prediction events in the grammar, symbol prediction events are inactive by default and any attempt to change that is a fatal error.

The activation status of a prediction event in the grammar can only be changed if the symbol is marked as a prediction event symbol in the grammar, and before the grammar is precomputed. However, if a symbol is marked as a prediction event symbol in the recognizer, the prediction event can be deactivated and reactivated in the recognizer.

Success cases: On success, the method returns the value of *reactivate*. The method succeeds trivially if the symbol is already set as indicated by *reactivate*.

Failure cases: If the active status of the prediction event for *sym_id* cannot be set as indicated by *reactivate*, the method fails. On failure, -2 is returned.

`int marpa_r_prediction_symbol_activate ( Marpa_Recognizer r, Marpa_Symbol_ID sym_id, int boolean )` [Function]

Allows the user to deactivate and reactivate symbol prediction events in the recognizer. If *boolean* is zero, the event is deactivated. If *boolean* is one, the event is activated.

Symbol prediction events are active by default if the symbol was set up for prediction events in the grammar. If a symbol was not set up for prediction events in the grammar, symbol prediction events are inactive by default and any attempt to change that is a fatal error.

Success cases: On success, the method returns the value of *boolean*. The method succeeds trivially if the symbol is already set as indicated by *boolean*.

Failure cases: If the active status of the prediction event for *sym_id* cannot be set as indicated by *boolean*, the method fails. On failure, -2 is returned.

`int marpa_g_symbol_is_prediction_event ( Marpa_Grammar g, Marpa_Symbol_ID sym_id)` [Function]

`int marpa_g_symbol_is_prediction_event_set ( Marpa_Grammar g, Marpa_Symbol_ID sym_id, int value)` [Function]

Libmarpa can be set up to generate a `MARPA_EVENT_SYMBOL_PREDICTED` event when a non-nulled symbol is predicted. A non-nulled symbol is said to be **predicted** when a instance of it is acceptable at the current earleme according to the grammar. Nulled symbols do not generate predictions.

For predicted events to occur, the symbol must be marked as a predicted event symbol. The `marpa_g_symbol_is_predicted_event_set()` function marks symbol *sym_id* as a predicted event symbol if *value* is 1, and unmarks it it as a predicted event symbol if *value* is 0. The `marpa_g_symbol_is_predicted_event()` method returns the current value of the predicted event marking for symbol *sym_id*.

Marking a prediction event sets its activation status to on. Unmarking a prediction event sets its activation status to off. The prediction event marking cannot be changed once the grammar is precomputed.

If a prediction event is marked, its activation status can be changed using the `marpa_g_prediction_symbol_activate()` method. Note that, if a symbol is marked as a prediction event symbol in the recognizer, its prediction event can be deactivated and reactivated in the recognizer.

Success: On success, 1 if symbol *sym_id* is a predicted event symbol after the call, 0 otherwise.

Failures: If *sym_id* is well-formed, but there is no such symbol, -1. If the grammar *g* is precomputed; or on other failure, -2.

20.6 Symbol expected events

`int marpa_r_expected_symbol_event_set ( Marpa_Recognizer r, Marpa_Symbol_ID symbol_id, int value)` [Function]

Sets the "expected symbol event bit" for *symbol_id* to *value*. A recognizer event is created whenever symbol *symbol_id* is expected at the current earleme. if and only if the expected symbol event bit for *symbol_id* is 1. The "expected symbol event bit" must be 1 or 0.

In this context, "expected" means "expected as a terminal". Even if a symbol is predicted at the current earleme, if it is not acceptable as a terminal, it does not trigger an "expected symbol event".

By default, the "expected symbol event bit" is 0. It is an error to attempt to set the "expected symbol event bit" to 1 for a nulling symbol, an inaccessible symbol, or an unproductive symbol.

Return value: The value of the event bit after the method call is finished. -2 if *symbol_id* is not the ID of a valid symbol; if it is the ID of an nulling, inaccessible for unproductive symbol; or on other failure.

20.7 Event codes

`int MARPA_EVENT_NONE` [Macro]
Applications should never see this event. Event value: Undefined. Suggested message: "No event".

`int MARPA_EVENT_COUNTED_NULLABLE` [Macro]
A nullable symbol is either the separator for, or the right hand side of, a sequence. Event value: The ID of the symbol. Suggested message: "This symbol is a counted nullable".

`int MARPA_EVENT_EARLEY_ITEM_THRESHOLD` [Macro]
This event indicates that an application-settable threshold on the number of Earley items has been reached or exceeded. See [marpa_r_earley_item_warning_threshold_set], page 39.

Event value: The current Earley item count. Suggested message: "Too many Earley items".

`int MARPA_EVENT_EXHAUSTED` [Macro]
The parse is exhausted. Event value: Undefined. Suggested message: "Recognizer is exhausted".

`int MARPA_EVENT_LOOP_RULES` [Macro]
One or more rules are loop rules — rules that are part of a cycle. Cycles are pathological cases of recursion, in which the same symbol string derives itself a potentially infinite number of times. Nonetheless, Marpa parses in the presence of these, and it is up to the application to treat these as fatal errors, something they almost always will wish to do. Event value: The count of loop rules. Suggested message: "Grammar contains a infinite loop".

`int MARPA_EVENT_NULLING_TERMINAL` [Macro]
A nulling symbol is also a terminal. Event value: The ID of the symbol. Suggested message: "This symbol is a nulling terminal".

`int MARPA_EVENT_SYMBOL_COMPLETED` [Macro]
The recognizer can be set to generate an event a symbol is completed using its `marpa_g_symbol_is_completion_event_set()` method. (A symbol is "completed" if and only if any rule with that symbol as its LHS is completed.) This event code indicates that one of those events occurred. Event value: The ID of the completed symbol. Suggested message: "Completed symbol".

`int MARPA_EVENT_SYMBOL_EXPECTED` [Macro]
The recognizer can be set to generate an event when a symbol is expected as a terminal, using its `marpa_r_expected_symbol_event_set()` method. Note that this event only triggers if the symbol is expected as a terminal. Predicted symbols which are not expected as terminals do not trigger this event. This event code indicates that one of those events occurred. Event value: The ID of the expected symbol. Suggested message: "Expecting symbol".

`int MARPA_EVENT_SYMBOL_NULLED` [Macro]

The recognizer can be set to generate an event when a symbol is nulled – that is, recognized as a zero-length symbol. To set an nulled symbol event, use the recognizer's `marpa_r_nulled_symbol_event_set()` method. This event code indicates that a nulled symbol event occurred. Event value: The ID of the nulled symbol. Suggested message: "Symbol was nulled".

`int MARPA_EVENT_SYMBOL_PREDICTED` [Macro]

The recognizer can be set to generate an event when a symbol is predicted. To set an predicted symbol event, use the recognizer's `marpa_g_symbol_is_prediction_event_set()` method. Unlike the `MARPA_EVENT_SYMBOL_EXPECTED` event, the `MARPA_EVENT_SYMBOL_PREDICTED` event triggers for predictions of both non-terminals and terminals. This event code indicates that a predicted symbol event occurred. Event value: The ID of the predicted symbol. Suggested message: "Symbol was predicted".

21 Error methods, macros and codes

21.1 Error methods

`Marpa_Error_Code marpa_g_error` (*Marpa_Grammar g, const char** `p_error_string`) [Function]

When a method fails, this method allows the application to read the error code. *p_error_string* is reserved for use by the internals. Applications should set it to `NULL`.

Return value: The last error code from a Libmarpa method. Always succeeds.

`Marpa_Error_Code marpa_g_error_clear` (*Marpa_Grammar g*) [Function]

Sets the error code to `MARPA_ERR_NONE`. Not often used, but now and then it can be useful to force the error code to a known state.

Return value: `MARPA_ERR_NONE`. Always succeeds.

21.2 Error Macros

`int MARPA_ERRCODE_COUNT` [Macro]

The number of error codes. All error codes, whether internal or external, will be integers, non-negative but strictly less than `MARPA_ERRCODE_COUNT`.

21.3 External error codes

This section lists the external error codes. These are the only error codes that users of the Libmarpa external interface should ever see. Internal error codes are in their own section (Section 21.4 [Internal error codes], page 72).

`int MARPA_ERR_NONE` [Macro]

No error condition. The error code is initialized to this value. Methods which do not result in failure sometimes reset the error code to `MARPA_ERR_NONE`. Numeric value: 0. Suggested message: `"No error"`.

`int MARPA_ERR_BAD_SEPARATOR` [Macro]

A separator was specified for a sequence rule, but its ID was not that of a valid symbol. Numeric value: 6. Suggested message: `"Separator has invalid symbol ID"`.

`int MARPA_ERR_BEFORE_FIRST_TREE` [Macro]

A tree iterator is positioned before the first tree, and it was specified in a context where that is not allowed. A newly created tree is positioned before the first tree. To position a newly created tree iterator to the first tree use the `marpa_t_next()` method. Numeric value: 91. Suggested message: `"Tree iterator is before first tree"`.

`int MARPA_ERR_COUNTED_NULLABLE` [Macro]

A “counted” symbol was found that is also a nullable symbol. A “counted” symbol is one that appears on the RHS of a sequence rule. If a symbol is nullable, counting its occurrences becomes difficult. Questions of definition and problems of implementation arise. At a minimum, a sequence with counted nullables would be wildly ambigious.

Sequence rules are simply an optimized shorthand for rules that can also be written in ordinary BNF. If the equivalent of a sequence of nullables is really what your application needs, nothing in Libmarpa prevents you from specifying that sequence with ordinary BNF rules.

Numeric value: 8. Suggested message: `"Nullable symbol on RHS of a sequence rule"`.

`int MARPA_ERR_DUPLICATE_RULE` [Macro]

This error indicates an attempt to add a BNF rule which is a duplicate of a BNF rule already in the grammar. Two BNF rules are considered duplicates if

- Both rules have the same left hand symbol, and
- Both rules have the same right hand symbols in the same order.

Duplication of sequence rules, and duplication between BNF rules and sequence rules, is dealt with by requiring that the LHS of a sequence rule not be the LHS of any other rule.

Numeric value: 11. Suggested message: `"Duplicate rule"`.

`int MARPA_ERR_DUPLICATE_TOKEN` [Macro]

This error indicates an attempt to add a duplicate token. A token is a duplicate if one already read at the same earleme has the same symbol ID and the same length. Numeric value: 12. Suggested message: `"Duplicate token"`.

`int MARPA_ERR_YIM_COUNT` [Macro]

This error code indicates that an implementation-defined limit on the number of Earley items per Earley set was exceedeed. This limit is different from the Earley item warning threshold, an optional limit on the number of Earley items in an Earley set, which can be set by the application.

The implementation defined-limit is very large, at least 500,000,000 earlemes. An application is unlikely ever to see this error. Libmarpa's use of memory would almost certainly exceed the implementation's limits before it occurred. Numeric value: 13. Suggested message: `"Maximum number of Earley items exceeded"`.

`int MARPA_ERR_EVENT_IX_NEGATIVE` [Macro]

A negative event index was specified. That is not allowed. Numeric value: 15. Suggested message: `"Negative event index"`.

`int MARPA_ERR_EVENT_IX_OOB` [Macro]

An non-negative event index was specified, but there is no event at that index. Since the events are in sequence, this means it was too large. Numeric value: 16. Suggested message: `"No event at that index"`.

`int MARPA_ERR_GRAMMAR_HAS_CYCLE` [Macro]

The grammar has a cycle — one or more loop rules. This is a recoverable error, although most applications will want to treat it as fatal. For more see the description of [marpa_g_precompute], page 32. Numeric value: 17. Suggested message: `"Grammar has cycle"`.

`int MARPA_ERR_HEADERS_DO_NOT_MATCH` [Macro]
This is an internal error, and indicates that Libmarpa was wrongly built. Libmarpa was compiled with headers which do not match the rest of the code. The solution is to find a correctly built Libmarpa. Numeric value: 98. Suggested message: "Internal error: Libmarpa was built incorrectly"

`int MARPA_ERR_I_AM_NOT_OK` [Macro]
The Libmarpa base grammar is in a "not ok" state. Currently, the only way this can happen is if Libmarpa memory is being overwritten. Numeric value: 29. Suggested message: "Marpa is in a not OK state".

`int MARPA_ERR_INACCESSIBLE_TOKEN` [Macro]
This error code indicates that the token symbol is an inaccessible symbol — one which cannot be reached from the start symbol.

Since the inaccessibility of a symbol is a property of the grammar, this error code typically indicates an application error. Nevertheless, a retry at this location, using another token ID, may succeed. At this writing, the author knows of no uses of this technique.

Numeric value: 18. Suggested message: "Token symbol is inaccessible".

`int MARPA_ERR_INVALID_BOOLEAN` [Macro]
A function was called that takes a boolean argument, but the value of that argument was not either 0 or 1. Numeric value: 22. Suggested message: "Argument is not boolean".

`int MARPA_ERR_INVALID_LOCATION` [Macro]
The location (Earley set ID) is not valid. It may be invalid for one of two reasons:

- It is negative, and it is being used as the argument to a method for which that negative value does not have a special meaning.
- It is after the latest Earley set.

For users of input models other than the standard one, the term “location”, as used in association with this error code, means Earley set ID or Earley set ordinal. In the standard input model, this will always be identical with Libmarpa’s other idea of location, the earleme.

Numeric value: 25. Suggested message: "Location is not valid".

`int MARPA_ERR_INVALID_START_SYMBOL` [Macro]
A start symbol was specified, but its symbol ID is not that of a valid symbol. Numeric value: 27. Suggested message: "Specified start symbol is not valid".

`int MARPA_ERR_INVALID_ASSERTION_ID` [Macro]
A method was called with an invalid assertion ID. This is a assertion ID which not only does not exist, but cannot exist. Currently that means its value is less than zero. Numeric value: 96. Suggested message: "Assertion ID is malformed".

`int MARPA_ERR_INVALID_RULE_ID` [Macro]
A method was called with an invalid rule ID. This is a rule ID which not only does not exist, but cannot exist. Currently that means its value is less than zero. Numeric value: 26. Suggested message: "Rule ID is malformed".

`int MARPA_ERR_INVALID_SYMBOL_ID` [Macro]
A method was called with an invalid symbol ID. This is a symbol ID which not only does not exist, but cannot exist. Currently that means its value is less than zero. Numeric value: 28. Suggested message: "Symbol ID is malformed".

`int MARPA_ERR_MAJOR_VERSION_MISMATCH` [Macro]
There was a mismatch in the major version number between the requested version of libmarpa, and the actual one. Numeric value: 30. Suggested message: "Libmarpa major version number is a mismatch".

`int MARPA_ERR_MICRO_VERSION_MISMATCH` [Macro]
There was a mismatch in the micro version number between the requested version of libmarpa, and the actual one. Numeric value: 31. Suggested message: "Libmarpa micro version number is a mismatch".

`int MARPA_ERR_MINOR_VERSION_MISMATCH` [Macro]
There was a mismatch in the minor version number between the requested version of libmarpa, and the actual one. Numeric value: 32. Suggested message: "Libmarpa minor version number is a mismatch".

`int MARPA_ERR_NO_EARLEY_SET_AT_LOCATION` [Macro]
A non-negative Earley set ID (also called an Earley set ordinal) was specified, but there is no corresponding Earley set. Since the Earley set ordinals are in sequence, this means that the specified ID is greater than that of the latest Earley set. Numeric value: 39. Suggested message: "Earley set ID is after latest Earley set".

`int MARPA_ERR_NOT_PRECOMPUTED` [Macro]
The grammar is not precomputed, and attempt was made to do something with it that is not allowed for unprecomputed grammars. For example, a recognizer cannot be created from a grammar until it is precomputed. Numeric value: 34. Suggested message: "This grammar is not precomputed".

`int MARPA_ERR_NO_PARSE` [Macro]
The application attempted to create a bocage from a recognizer without a parse. Applications will often want to treat this as a soft error. Numeric value: 41. Suggested message: "No parse".

`int MARPA_ERR_NO_RULES` [Macro]
A grammar which has no rules is being used in a way that is not allowed. Usually the problem is that the user is trying to precompute the grammar. Numeric value: 42. Suggested message: "This grammar does not have any rules".

`int MARPA_ERR_NO_START_SYMBOL` [Macro]
The grammar has no start symbol, and an attempt was made to perform an operation which requires one. Usually the problem is that the user is trying to precompute the grammar. Numeric value: 43. Suggested message: "This grammar has no start symbol".

`int MARPA_ERR_NO_SUCH_ASSERTION_ID` [Macro]
A method was called with an assertion ID which is well-formed, but the assertion does not exist. Numeric value: 97. Suggested message: "No assertion with this ID exists".

`int MARPA_ERR_NO_SUCH_RULE_ID` [Macro]
A method was called with a rule ID which is well-formed, but the rule does not exist. Numeric value: 89. Suggested message: "No rule with this ID exists".

`int MARPA_ERR_NO_SUCH_SYMBOL_ID` [Macro]
A method was called with a symbol ID which is well-formed, but the symbol does not exist. Numeric value: 90. Suggested message: "No symbol with this ID exists".

`int MARPA_ERR_NO_TOKEN_EXPECTED_HERE` [Macro]
This error code indicates that no tokens at all were expected at this earleme location. This can only happen in alternative input models.

Typically, this indicates an application programming error. Retrying input at this location will always fail. But if the application is able to leave this earleme empty, a retry at a later location, using this or another token, may succeed. At this writing, the author knows of no uses of this technique.

Numeric value: 44. Suggested message: "No token is expected at this earleme location".

`int MARPA_ERR_NOT_A_SEQUENCE` [Macro]
This error occurs in situations where a rule is required to be a sequence, and indicates that the rule of interest is, in fact, not a sequence.

Numeric value: 99. Suggested message: "Rule is not a sequence".

`int MARPA_ERR_NULLING_TERMINAL` [Macro]
Marpa does not allow a symbol to be both nulling and a terminal. Numeric value: 49. Suggested message: "A symbol is both terminal and nulling".

`int MARPA_ERR_ORDER_FROZEN` [Macro]
The Marpa order object has been frozen. If a Marpa order object is frozen, it cannot be changed.

Multiple tree iterators can share a Marpa order object, but that order object is frozen after the first tree iterator is created from it. Applications can order an bocage in many ways, but they must do so by creating multiple order objects.

Numeric value: 50. Suggested message: "The ordering is frozen".

`int MARPA_ERR_PARSE_EXHAUSTED` [Macro]
The parse is exhausted. Numeric value: 53. Suggested message: "The parse is exhausted".

`int MARPA_ERR_PARSE_TOO_LONG` [Macro]
The parse is too long. The limit on the length of a parse is implementation dependent, but it is very large, at least 500,000,000 earlemes.

This error code is unlikely in the standard input model. Almost certainly memory would be exceeded before it could occur. If an application sees this error, it almost certainly using one of the non-standard input models.

Most often this message will occur because of a request to add a single extremely long token, perhaps as a result of an application error. But it is also possible this error condition will occur after the input of a large number of long tokens.

Numeric value: 54. Suggested message: `"This input would make the parse too long"`.

`int MARPA_ERR_POINTER_ARG_NULL` [Macro]

In a method which takes pointers as arguments, one of the pointer arguments is `NULL`, in a case where that is not allowed. One such method is `marpa_r_progress_item()`. Numeric value: 56. Suggested message: `"An argument is null when it should not be"`.

`int MARPA_ERR_PRECOMPUTED` [Macro]

An attempt was made to use a precomputed grammar in a way that is not allowed. Often this is an attempt to change the grammar. Nearly every change to a grammar after precomputation invalidates the precomputation, and is therefore not allowed. Numeric value: 57. Suggested message: `"This grammar is precomputed"`.

`int MARPA_ERR_PROGRESS_REPORT_NOT_STARTED` [Macro]

No recognizer progress report is currently active, and an action has been attempted which is inconsistent with that. One such action would be a `marpa_r_progress_item()` call. Numeric value: 59. Suggested message: `"No progress report has been started"`.

`int MARPA_ERR_PROGRESS_REPORT_EXHAUSTED` [Macro]

The progress report is "exhausted" — all its items have been iterated through. Numeric value: 58. Suggested message: `"The progress report is exhausted"`.

`int MARPA_ERR_RANK_TOO_LOW` [Macro]

A symbol or rule rank was specified which was less than an implementation-defined minimum. Implementations will always allow at least those ranks in the range between −134,217,727 and 134,217,727. Numeric value: 85. Suggested message: `"Rule or symbol rank too low"`.

`int MARPA_ERR_RANK_TOO_HIGH` [Macro]

A symbol or rule rank was specified which was greater than an implementation-defined maximum. Implementations will always allow at least those ranks in the range between −134,217,727 and 134,217,727. Numeric value: 86. Suggested message: `"Rule or symbol rank too high"`.

`int MARPA_ERR_RECCE_IS_INCONSISTENT` [Macro]

The recognizer is "inconsistent", usually because the user has rejected one or more rules or terminals, and has not yet called the `marpa_r_consistent()` method. Numeric value: 95. Suggested message: `"The recognizer is inconsistent.`

`int MARPA_ERR_RECCE_NOT_ACCEPTING_INPUT` [Macro]

The recognizer is not accepting input, and the application has attempted something that is inconsistent with that fact. Numeric value: 60. Suggested message: `"The recognizer is not accepting input"`.

`int MARPA_ERR_RECCE_NOT_STARTED` [Macro]
The recognizer has not been started. and the application has attempted something that is inconsistent with that fact. Numeric value: 61. Suggested message: "The recognizer has not been started".

`int MARPA_ERR_RECCE_STARTED` [Macro]
The recognizer has been started. and the application has attempted something that is inconsistent with that fact. Numeric value: 62. Suggested message: "The recognizer has been started".

`int MARPA_ERR_RHS_IX_NEGATIVE` [Macro]
The index of a RHS symbol was specified, and it was negative. That is not allowed. Numeric value: 63. Suggested message: "RHS index cannot be negative".

`int MARPA_ERR_RHS_IX_OOB` [Macro]
A non-negative index of RHS symbol was specified, but there is no symbol at that index. Since the indexes are in sequence, this means the index was greater than or equal to the rule length. Numeric value: 64. Suggested message: "RHS index must be less than rule length".

`int MARPA_ERR_RHS_TOO_LONG` [Macro]
An attempt was made to add a rule with too many right hand side symbols. The limit on the RHS symbol count is implementation dependent, but it is very large, at least 500,000,000 symbols. This is far beyond what is required in any current practical grammar. An application with rules of this length is almost certain to run into memory and other limits. Numeric value: 65. Suggested message: "The RHS is too long".

`int MARPA_ERR_SEQUENCE_LHS_NOT_UNIQUE` [Macro]
The LHS of a sequence rule cannot be the LHS of any other rule, whether a sequence rule or a BNF rule. An attempt was made to violate this restriction. Numeric value: 66. Suggested message: "LHS of sequence rule would not be unique".

`int MARPA_ERR_START_NOT_LHS` [Macro]
The start symbol is not on the LHS on any rule. That means it could never match any possible input, not even the null string. Presumably, an error in writing the grammar. Numeric value: 73. Suggested message: "Start symbol not on LHS of any rule".

`int MARPA_ERR_SYMBOL_IS_NOT_COMPLETION_EVENT` [Macro]
An attempt was made to use a symbol in a way that requires it to be set up for completion events, but the symbol was not set set up for completion events. The archetypal case is an attempt to activate completion events for the symbol in the recognizer. The archetypal case is an attempt to activate a completion event in the recognizer for a symbol that is not set up as a completion event. Numeric value: 92. Suggested message: "Symbol is not set up for completion events".

`int MARPA_ERR_SYMBOL_IS_NOT_NULLED_EVENT` [Macro]
An attempt was made to use a symbol in a way that requires it to be set up for nulled events, but the symbol was not set set up for nulled events. The archetypal case is

an attempt to activate a nulled events in the recognizer for a symbol that is not set up as a nulled event. Numeric value: 93. Suggested message: "Symbol is not set up for nulled events".

`int MARPA_ERR_SYMBOL_IS_NOT_PREDICTION_EVENT` [Macro]
An attempt was made to use a symbol in a way that requires it to be set up for predictino events, but the symbol was not set set up for predictino events. The archetypal case is an attempt to activate a prediction event in the recognizer for a symbol that is not set up as a prediction event. Numeric value: 94. Suggested message: "Symbol is not set up for prediction events".

`int MARPA_ERR_SYMBOL_VALUED_CONFLICT` [Macro]
Unvalued symbols are a deprecated Marpa feature, which may be avoided with the `marpa_g_force_valued()` method. An unvalued symbol may take on any value, and therefore a symbol which is unvalued at some points cannot safely to be used to contain a value at others. This error indicates that such an unsafe use is being attempted. Numeric value: 74. Suggested message: "Symbol is treated both as valued and unvalued".

`int MARPA_ERR_TERMINAL_IS_LOCKED` [Macro]
An attempt was made to change the terminal status of a symbol to a different value after it was locked. Numeric value: 75. Suggested message: "The terminal status of the symbol is locked".

`int MARPA_ERR_TOKEN_IS_NOT_TERMINAL` [Macro]
A token was specified whose symbol ID is not a terminal. Numeric value: 76. Suggested message: "Token symbol must be a terminal".

`int MARPA_ERR_TOKEN_LENGTH_LE_ZERO` [Macro]
A token length was specified which is less than or equal to zero. Zero-length tokens are not allowed in Libmarpa. Numeric value: 77. Suggested message: "Token length must greater than zero".

`int MARPA_ERR_TOKEN_TOO_LONG` [Macro]
The token length is too long. The limit on the length of a token is implementation dependent, but it is at least 500,000,000 earlemes. An application using a token that long is almost certain to run into some other limit. Numeric value: 78. Suggested message: "Token is too long".

`int MARPA_ERR_TREE_EXHAUSTED` [Macro]
A Libmarpa parse tree iterator is “exhausted”, that is, it has no more parses. Numeric value: 79. Suggested message: "Tree iterator is exhausted".

`int MARPA_ERR_TREE_PAUSED` [Macro]
A Libmarpa tree is “paused” and an operation was attempted which is inconsistent with that fact. Typically, this operation will be a call of the `marpa_t_next()` method. Numeric value: 80. Suggested message: "Tree iterator is paused".

`int MARPA_ERR_UNEXPECTED_TOKEN_ID` [Macro]
An attempt was made to read a token where a token with that symbol ID is not expected. This message can also occur when an attempt is made to read a token at a location where no token is expected. Numeric value: 81. Suggested message: "Unexpected token".

`int MARPA_ERR_UNPRODUCTIVE_START` [Macro]
The start symbol is unproductive. That means it could never match any possible input, not even the null string. Presumably, an error in writing the grammar. Numeric value: 82. Suggested message: "Unproductive start symbol".

`int MARPA_ERR_VALUATOR_INACTIVE` [Macro]
The valuator is inactive in a context where that should not be the case. Numeric value: 83. Suggested message: "Valuator inactive".

`int MARPA_ERR_VALUED_IS_LOCKED` [Macro]
Unvalued symbols are a deprecated Marpa feature, which may be avoided with the `marpa_g_force_valued()` method. This error code indicates that the valued status of a symbol is locked, and an attempt was made to change it to a status different from the current one. Numeric value: 84. Suggested message: "The valued status of the symbol is locked".

`int MARPA_ERR_SYMBOL_IS_NULLING` [Macro]
An attempt was made to do something with a nulling symbol that is not allowed. For example, the ID of a nulling symbol cannot be an argument to `marpa_r_expected_symbol_event_set()` — because it is not possible to create an "expected symbol" event for a nulling symbol. Numeric value: 87. Suggested message: "Symbol is nulling".

`int MARPA_ERR_SYMBOL_IS_UNUSED` [Macro]
An attempt was made to do something with an unused symbol that is not allowed. An "unused" symbol is a inaccessible or unproductive symbol. For example, the ID of a unused symbol cannot be an argument to `marpa_r_expected_symbol_event_set()` — because it is not possible to create an "expected symbol" event for an unused symbol. Numeric value: 88. Suggested message: "Symbol is not used".

21.4 Internal error codes

An internal error code may be one of two things: First, it can be an error code which arises from an internal Libmarpa programming issue (in other words, something happening in the code that was not supposed to be able to happen.) Second, it can be an error code which only occurs when a method from Libmarpa's internal interface is used. Both kinds of internal error message share one common trait — users of the Libmarpa's external interface should never see them.

Internal error messages require someone with knowledge of the Libmarpa internals to follow up on them. They usually do not have descriptions or suggested messages.

`int MARPA_ERR_AHFA_IX_NEGATIVE` [Macro]
Numeric value: 1.

`int MARPA_ERR_AHFA_IX_OOB` [Macro]
Numeric value: 2.

`int MARPA_ERR_ANDID_NEGATIVE` [Macro]
Numeric value: 3.

`int MARPA_ERR_ANDID_NOT_IN_OR` [Macro]
Numeric value: 4.

`int MARPA_ERR_ANDIX_NEGATIVE` [Macro]
Numeric value: 5.

`int MARPA_ERR_BOCAGE_ITERATION_EXHAUSTED` [Macro]
Numeric value: 7.

`int MARPA_ERR_DEVELOPMENT` [Macro]
"Development" errors were used heavily during Libmarpa's development, when it was not yet clear how precisely to classify every error condition. Unless they are using a developer's version, users of the external interface should never see development errors.

Development errors have an error string associated with them. The error string is a short 7-bit ASCII error string which describes the error. Numeric value: 9. Suggested message: "Development error, see string".

`int MARPA_ERR_DUPLICATE_AND_NODE` [Macro]
Numeric value: 10.

`int MARPA_ERR_YIM_ID_INVALID` [Macro]
Numeric value: 14.

`int MARPA_ERR_INTERNAL` [Macro]
A "catchall" internal error. Numeric value: 19.

`int MARPA_ERR_INVALID_AHFA_ID` [Macro]
The AHFA ID was invalid. There are no AHFAs any more, so this message should not occur. Numeric value: 20.

`int MARPA_ERR_INVALID_AIMID` [Macro]
The AHM ID was invalid. The term "AIMID" is a legacy of earlier implementations and must be kept for backward compatibility. Numeric value: 21.

`int MARPA_ERR_INVALID_IRLID` [Macro]
Numeric value: 23.

`int MARPA_ERR_INVALID_NSYID` [Macro]
Numeric value: 24.

`int MARPA_ERR_NOOKID_NEGATIVE` [Macro]
Numeric value: 33.

`int MARPA_ERR_NOT_TRACING_COMPLETION_LINKS` [Macro]
Numeric value: 35.

`int MARPA_ERR_NOT_TRACING_LEO_LINKS` [Macro]
Numeric value: 36.

`int MARPA_ERR_NOT_TRACING_TOKEN_LINKS` [Macro]
Numeric value: 37.

`int MARPA_ERR_NO_AND_NODES` [Macro]
Numeric value: 38.

`int MARPA_ERR_NO_OR_NODES` [Macro]
Numeric value: 40.

`int MARPA_ERR_NO_TRACE_YS` [Macro]
Numeric value: 46.

`int MARPA_ERR_NO_TRACE_PIM` [Macro]
Numeric value: 47.

`int MARPA_ERR_NO_TRACE_YIM` [Macro]
Numeric value: 45.

`int MARPA_ERR_NO_TRACE_SRCL` [Macro]
Numeric value: 48.

`int MARPA_ERR_ORID_NEGATIVE` [Macro]
Numeric value: 51.

`int MARPA_ERR_OR_ALREADY_ORDERED` [Macro]
Numeric value: 52.

`int MARPA_ERR_PIM_IS_NOT_LIM` [Macro]
Numeric value: 55.

`int MARPA_ERR_SOURCE_TYPE_IS_NONE` [Macro]
Numeric value: 70.

`int MARPA_ERR_SOURCE_TYPE_IS_TOKEN` [Macro]
Numeric value: 71.

`int MARPA_ERR_SOURCE_TYPE_IS_COMPLETION` [Macro]
Numeric value: 68.

`int MARPA_ERR_SOURCE_TYPE_IS_LEO` [Macro]
Numeric value: 69.

`int MARPA_ERR_SOURCE_TYPE_IS_AMBIGUOUS` [Macro]
Numeric value: 67.

`int MARPA_ERR_SOURCE_TYPE_IS_UNKNOWN` [Macro]
Numeric value: 72.

22 Technical notes

This section contains technical notes that are not necessary for the main presentation, but which may be useful or interesting.

22.1 Data types used by Libmarpa

Libmarpa does not use any floating point data or strings. All data are either integers or pointers.

22.2 Why so many time objects?

Marpa is an aggressively multi-pass algorithm. Marpa achieves its efficiency, not in spite of making multiple passes over the data, but because of it. Marpa regularly substitutes two fast O(n) passes for a single O(n log n) pass. Marpa's proliferation of time objects is in keeping with its multi-pass approach.

Bocage objects come at no cost, even for unambiguous parses, because the same pass which creates the bocage also deals with other issues which are of major significance for unambiguous parses. It is the post-processing of the bocage pass that enables Marpa to do both left- and right-recursion in linear time.

Of the various objects, the best case for elimination is of the ordering object. In many cases, the ordering is trivial. Either the parse is unambiguous, or the application does not care about the order in which parses are returned. But while it would be easy to add an option to bypass creation of an ordering object, there is little to be gained from it. When the ordering is trivial, its overhead is very small — essentially a handful of subroutine calls. Many orderings accomplish nothing, but these cost next to nothing.

Tree objects come at minimal cost to unambiguous grammars, because the same pass that allows iteration through multiple parse trees does the tree traversal. This eliminates much of the work that otherwise would need to be done in the valuation time object. In the current implement, the valuation time object needs only to step through a sequence already determined in the tree iterator.

22.3 Numbered objects

As the name suggests, the choice was made to implement numbered objects as integers, and not as pointers. In standard-conformant C, integers can be safely checked for validity, while pointers cannot.

There are efficiency tradeoffs between pointers and integers but they are complicated, and they go both ways. Pointers can be faster, but integers can be used as indexes into more than one data structure. Which is actually faster depends on the design. Integers allow for a more flexible design, so that once the choice is settled on, careful programming can make them a win, possibly a very big one.

The approach taken in Libmarpa was to settle, from the outset, on integers as the implementation for numbered objects, and to optimize on that basis. The author concedes that it is possible that others redoing Libmarpa from scratch might find that pointers are faster. But the author is confident that they will also discover, on modern architectures,

that the lack of safe validity checking is far too high a price to pay for the difference in speed.

22.4 LHS terminals

Marpa's idea in losing the sharp division between terminals and non-terminals is that the distinction, while helpful for proving theorems, is not essential in practice. LHS symbols in the input might be useful for "short circuiting" the rules in which they occur. This may prove helpful in debugging, or have other applications.

However, it also can be useful, for checking input validity as well as for efficiency, to follow tradition and distinguish non-terminals from terminals. For this reason, the traditional behavior is the default in Libmarpa.

23 Advanced input models

In an earlier chapter, we introduced Libmarpa's concept of input, and described its basic input models. See Chapter 5 [Input], page 7. In this chapter we describe Libmarpa's advanced models of input. These advanced input models have attracted considerable interest. However, they have seen little actual use so far, and for that reason we delayed their consideration until now.

A Libmarpa input model is *advanced* if it allows tokens of length other than 1. The advanced input models are also called *variable-length token models* because they allow the token length to vary from the "normal" length of 1.

23.1 The dense variable-length token model

In the *dense variable-length model of input*, one or more successful calls of `marpa_r_alternative()` must be immediately previous to every call to `marpa_r_earleme_complete()`. Note that, for a variable-length input model to be "dense" according to this definition, at least one successful call of `marpa_r_alternative()` must be immediately previous to each call to `marpa_r_earleme_complete()`. Recall that, in this document, we say that a `marpa_r_alternative()` call is "immediately previous" to a `marpa_r_earleme_complete()` call iff that `marpa_r_earleme_complete()` call is the first `marpa_r_earleme_complete()` call after the `marpa_r_alternative()` call.

In the dense model of input, after a successful call of `marpa_r_alternative()`, the earleme variables are as follows:

- The furthest earleme will be `max(`*`old_f, old_c+length`*`)`,
  - where *old_f* is the furthest earleme before the call to `marpa_r_alternative()`,
  - *old_c* is the value of the current earleme before the call to `marpa_r_alternative()`, and
  - *length* is the length of the token read.
- `marpa_r_alternative()` never changes the latest or current earleme.

In the dense variable-length model of input, the effect of the `marpa_r_earleme_complete()` mutator on the earleme variables is the same as for the basic models of input. See Section 5.2.1 [The standard model of input], page 8.

In the dense model of input, the latest earleme is always the same as the current earleme. In fact, the latest earleme and the current earleme are always the same, except in the fully general model of input.

23.2 The fully general input model

In the *sparse variable-length model of input*, zero or more successful calls of `marpa_r_alternative()` must be immediately previous to every call to `marpa_r_earleme_complete()`. The sparse model is the dense variable-length model, with its only restriction lifted — the sparse variable-length input model allows calls to `marpa_r_earleme_complete()` that are not immediately preceded by calls to `marpa_r_alternative()`.

Since it is unrestricted, the sparse input model is Libmarpa's fully general input model. Because of this, it may be useful for us specify the effect of mutators on the earleme variables in detail, even at the expense of some repetition.

In the sparse input model, *empty earlemes* are now possible. An empty earleme is an earleme with no tokens and no Earley set. An empty earleme occurs iff `marpa_r_earleme_complete()` is called when there is no immediately previous call to `marpa_r_alternative()`. The sparse model takes its name from the fact that there may be earlemes with no Earley set. In the sparse model, Earley sets are "sparsely" distributed among the earlemes.

In the dense model of input, the effect on the earleme variables of a successful call of the `marpa_r_alternative()` mutator is the same as for the sparse model of input:

- The furthest earleme will be `max(`*`old_f`*`,` *`old_c`*`+`*`length`*`)`,
  - where *old_f* is the furthest earleme before the call to `marpa_r_alternative()`,
  - *old_c* is the value of the current earleme before the call to `marpa_r_alternative()`, and
  - *length* is the length of the token read.
- `marpa_r_alternative()` never changes the latest or current earleme.

In the sparse model, when the earleme is not empty, the effect of a call to `marpa_r_earleme_complete()` on the earleme variables is the same as in the dense and the basic models of input. Specifically, the following will be true:

- The current earleme will be advanced to *`old_c`*`+1`, where *old_c* is the current earleme before the call.
- The latest earleme will be *`old_c`*`+1`, and therefore will be equal to the current earleme.
- The value of the furthest earleme is never changed by a call to `marpa_r_earleme_complete()`.

Recall that, in the dense and basic input models, as a matter of definition, there are no empty earlemes. For the sparse input model, in the case of an empty earleme, the effect of the `marpa_r_earleme_complete()` mutator on the earleme variables is the following:

- The current earleme will be advanced to *`old_c`*`+1`, where *old_c* is the current earleme before the call.
- The latest earleme will remain at *old_l*, where the latest earleme before the call is *old_l*. This implies that the latest earleme will be less than the current earleme.
- The furthest earleme is never changed by a call to `marpa_r_earleme_complete()`.

After a call to `marpa_r_earleme_complete()` for an empty earleme, the lastest and current earlemes will have different values. In a parse that never calls `marpa_r_earleme_complete()` for an empty earleme, the lastest and current earlemes will always be the same.

24 Futures

This chapter discusses features that are **not** in the external interface, but that might be added to the external interface in the future.

24.1 Orthogonal treatment of exhaustion

The treatment of parse exhaustion is very awkward. `marpa_r_start_input()` returns success on exhaustion, while `marpa_r_earleme_complete()` either returns success or a hard failure, depending on circumstances. See [marpa_r_earleme_complete], page 36, and [marpa_r_start_input], page 34.

Ideally the treatment should be simpler, more intuitive and more orthogonal. Better, perhaps, would be to always treat parse exhaustion as a soft failure.

24.2 Furthest earleme values

`marpa_r_furthest_earleme` returns `unsigned int` which is non-orthogonal with `marpa_r_current_earleme`. This leaves no room for an failure return value, which we deal with by not checking for failures, of which the only important one is calling `marpa_r_furthest_earleme` before the start of input. To consider `marpa_r_furthest_earleme` we consider furthese earleme to have been initialized when the recognizer was created, which is another non-orthogonality with `marpa_r_current_earleme`.

All this might be fine, if something were gained, but in fact in the furthest earleme, unless there is a problem, always becomes the current earleme, and no use cases for extremely long variable-length tokens are envisioned, so that the two should never be far apart. Additionally, the additional values for the furthest earleme only come into play if the parse is to large for the computer memories as of this writing. Summarizing, `marpa_r_furthest_earleme`, should return an `int`, like `marpa_r_current_earleme`, and the non-orthogonalities should be eliminated.

24.3 Additional recoverable failures in marpa_r_alternative()

Among the hard failures that marpa_r_alternative() returns are the error codes `MARPA_ERR_DUPLICATE_TOKEN`, `MARPA_ERR_NO_TOKEN_EXPECTED_HERE` and `MARPA_ERR_INACCESSIBLE_TOKEN`. These are currently irrecoverable. They may in fact be fully recoverable, but are not documented as such because this has not been tested.

At this writing, we know of no applications which attempt to recover from these errors. It is possible that these error codes may also be useable for the techniques similar to the Ruby Slippers, as of this writing, we know of no proposals to use them in this way.

24.4 Untested methods

The methods of this section are not in the external interface, because they have not been adequately tested. Their fate is uncertain. Users should regard these methods as unsupported.

24.4.1 Ranking methods

`Marpa_Rank marpa_g_default_rank_set ( Marpa_Grammar g, Marpa_Rank rank)` [Function]

`Marpa_Rank marpa_g_default_rank ( Marpa_Grammar g)` [Function]

These methods, respectively, set and query the default rank of the grammar. When a grammar is created, the default rank is 0. When rules and symbols are created, their rank is the default rank of the grammar.

Changing the grammar's default rank does not affect those rules and symbols already created, only those that will be created. This means that the grammar's default rank can be used to, in effect, assign ranks to groups of rules and symbols. Applications may find this behavior useful.

Return value: On success, returns the rank **after** the call, and sets the error code to `MARPA_ERR_NONE`. On failure, returns -2, and sets the error code to an appropriate value, which will never be `MARPA_ERR_NONE`. Note that when the rank is -2, the error code is the only way to distinguish success from failure. The error code can be determined by using the `marpa_g_error()` call.

`Marpa_Rank marpa_g_symbol_rank_set ( Marpa_Grammar g, Marpa_Symbol_ID sym_id, Marpa_Rank rank)` [Function]

`Marpa_Rank marpa_g_symbol_rank ( Marpa_Grammar g, Marpa_Symbol_ID sym_id)` [Function]

These methods, respectively, set and query the rank of a symbol *sym_id*. When *sym_id* is created, its rank initialized to the default rank of the grammar.

Return value: On success, returns the rank **after** the call, and sets the error code to `MARPA_ERR_NONE`. On failure, returns -2, and sets the error code to an appropriate value, which will never be `MARPA_ERR_NONE`. Note that when the rank is -2, the error code is the only way to distinguish success from failure. The error code can be determined by using the `marpa_g_error()` call.

24.4.2 Zero-width assertion methods

`Marpa_Assertion_ID marpa_g_zwa_new ( Marpa_Grammar g, int default_value)` [Function]

`int marpa_g_zwa_place ( Marpa_Grammar g, Marpa_Assertion_ID zwaid, Marpa_Rule_ID xrl_id, int rhs_ix)` [Function]

`int marpa_r_zwa_default ( Marpa_Recognizer r, Marpa_Assertion_ID zwaid)` [Function]

On success, returns previous default value of the assertion.

`int marpa_r_zwa_default_set ( Marpa_Recognizer r, Marpa_Assertion_ID zwaid, int default_value)` [Function]

Changes default value to *default_value*. On success, returns previous default value of the assertion.

`Marpa_Assertion_ID marpa_g_highest_zwa_id ( Marpa_Grammar g )` [Function]

24.4.3 Methods for revising parses

Marpa allows an application to "change its mind" about a parse, rejected rule previously recognized or predicted, and terminals previously scanned. The methods in this section provide that capability.

`Marpa_Earleme marpa_r_clean (` *Marpa_Recognizer* `r`) [Function]

25 Deprecated techniques and methods

25.1 Valued and unvalued symbols

25.1.1 What unvalued symbols were

Libmarpa symbols can have values, which is the traditional way of doing semantics. Libmarpa also allows symbols to be unvalued. An *unvalued* symbol is one whose value is unpredictable from instance to instance. If a symbol is unvalued, we sometimes say that it has "whatever" semantics.

Situations where the semantics can tolerate unvalued symbols are surprisingly frequent. For example, the top-level of many languages is a series of major units, all of whose semantics are typically accomplished via side effects. The compiler is typically indifferent to the actual value produced by these major units, and tracking them is a waste of time. Similarly, the value of the separators in a list is typically ignored.

Rules are unvalued if and only if their LHS symbols are unvalued. When rules and symbols are unvalued, Libmarpa optimizes their evaluation.

It is in principle unsafe to check the value of a symbol if it can be unvalued. For this reason, once a symbol has been treated as valued, Libmarpa marks it as valued. Similarly, once a symbol has been treated as unvalued, Libmarpa marks it as unvalued. Once marked, a symbol's valued status is *locked* and cannot be changed later.

The valued status of terminals is marked the first time they are read. The valued status of LHS symbols must be explicitly marked by the application when initializing the valuator — this is Libmarpa's equivalent of registering a callback.

LHS terminals are disabled by default. If allowed, the user should be aware that the valued status of a LHS terminal will be locked in the recognizer if it is used as a terminal, and the symbol's use as a rule LHS in the valuator must be consistent with the recognizer's marking.

Marpa reports an error when a symbol's use conflicts with its locked valued status. Doing so usually saves the Libmarpa user some tricky debugging further down the road.

25.1.2 Grammar methods dealing with unvalued symbols

`int marpa_g_symbol_is_valued_set` (*Marpa_Grammar* `g`, *Marpa_Symbol_ID* `symbol_id`, *int value*) [Function]

`int marpa_g_symbol_is_valued` (*Marpa_Grammar* `g`, *Marpa_Symbol_ID* `symbol_id`) [Function]

These methods, respectively, set and query the "valued status" of a symbol. Once set to a value with the `marpa_g_symbol_is_valued_set()` method, the valued status of a symbol is "locked" at that value. It cannot thereafter be changed. Subsequent calls to `marpa_g_symbol_is_valued_set()` for the same *sym_id* will fail, leaving *sym_id*'s valued status unchanged, unless *value* is the same as the locked-in value.

Return value: On success, 1 if the symbol *symbol_id* is valued after the call, 0 if not. If the valued status is locked and *value* is different from the current status, −2. If *value* is not 0 or 1; or on other failure, −2.

25.1.3 Registering semantics in the valuator

By default, Libmarpa's valuator objects assume that non-terminal symbols have no semantics. The archetypal application will need to register symbols that contain semantics. The primary method for doing this is `marpa_v_symbol_is_valued()`. Applications will typically register semantics by rule, and these applications will find the `marpa_v_rule_is_valued()` method more convenient.

`int marpa_v_symbol_is_valued_set ( Marpa_Value v, Marpa_Symbol_ID sym_id, int status )` [Function]

`int marpa_v_symbol_is_valued ( Marpa_Value v, Marpa_Symbol_ID sym_id )` [Function]

These methods, respectively, set and query the valued status of symbol *sym_id*. `marpa_v_symbol_is_valued_set()` will set the valued status to the value of its *status* argument. A valued status of 1 indicates that the symbol is valued. A valued status of 0 indicates that the symbol is unvalued. If the valued status is locked, an attempt to change to a status different from the current one will fail (error code `MARPA_ERR_VALUED_IS_LOCKED`).

Return value: On success, the valued status **after** the call. If *value* is not either 0 or 1, or on other failure, −2.

`int marpa_v_rule_is_valued_set ( Marpa_Value v, Marpa_Rule_ID rule_id, int status )` [Function]

`int marpa_v_rule_is_valued ( Marpa_Value v, Marpa_Rule_ID rule_id )` [Function]

These methods, respectively, set and query the valued status for the LHS symbol of rule *rule_id*. `marpa_v_rule_is_valued_set()` sets the valued status to the value of its *status* argument.

A valued status of 1 indicates that the symbol is valued. A valued status of 0 indicates that the symbol is unvalued. If the valued status is locked, an attempt to change to a status different from the current one will fail (error code `MARPA_ERR_VALUED_IS_LOCKED`).

Rules have no valued status of their own. The valued status of a rule is always that of its LHS symbol. These methods are conveniences — they save the application the trouble of looking up the rule's LHS.

Return value: On success, the valued status of the rule *rule_id*'s LHS symbol **after** the call. If *value* is not either 0 or 1, or on other failure, −2.

`int marpa_v_valued_force ( Marpa_Value v)` [Function]

This methods locks the valued status of all symbols to 1, indicated that the symbol is valued. If this is not possible, for example because one of the grammar's symbols already is locked at a valued status of 0, failure is returned.

Return value: On success, a non-negative number. On failure, returns −2, and sets the error code to an appropriate value, which will never be `MARPA_ERR_NONE`.

Index of terms

This index is of terms that are used in a special sense in this document. Not every use of these terms is indexed — only those uses which are in some way defining.

A

B

C

D

E

F

H

I

L

M

N

O

P

R

S

T

U

V

W