Libmarpa

Version 11.0.1
7 December 2022

Jeffrey Kegler

This manual (7 December 2022) is for Libmarpa 11.0.1.

Published 7 December 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.

1.1 Updates

For important information that has changed since the last stable release, there is an "updates" document (`https://github.com/jeffreykegler/libmarpa/blob/updated/UPDATES.md`). The updates document includes

item descriptions of bugs in the latest stable release;

- notices which are useful to current users, but which do not justify a full new stable distribution; and
- other information that we want to be able to update without issuing a new stable release.

To allow that information to be kept current without issuing a new stable release, we describe how to obtain support in the updates document. See Chapter 27 [Support], page 95.

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 13 [Introduction to the method descriptions], page 31, 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. See Section 4.2 [Parsing theory preliminaries], page 4.

3 Overview of Libmarpa

This chapter contains a quick overview of Libmarpa, using standard parsing terminology. It is intended to help a prospective reader of the whole document to know what to expect. Details and careful definitions will be provided in later chapters.

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 a "bocage", which is an optimized parse forest. Libmarpa also supports the ordering, iteration and evaluation of the parse trees in the bocage.

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 Terms, definitions and notation

4.1 Miscellaneous definitions

- *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 that uses Libmarpa.
- A *boolean* value, or *boolean*, is an integer that is 0 or 1.
- *iff* abbreviates “if and only if”.
- `max(x,y)` is the maximum of `x` and `y`, where `x` and `y` are two numbers.
- An *indeterminate value* is either an unspecified value or a trap value. Our use of this term is consistent with its use in the C99 standard.
- *Libmarpa method*, or just *method* means a C function or a function-like macro of the Libmarpa library.
- A *trap value*, also called a *trap representation*, is a value that when accessed causes undefined behavior. Our use of this term is consistent with its use in the C99 standard. See Section 25.5 [Trap representations], page 92.
- An *undefined behavior* is a behavior that this document does not specify. One implication is that this behavior might be problematic. Our use of this term is consistent with its use in the C99 standard.
- An *unspecified behavior* is a behavior that, within a range of possibilities, is not further specified by this document. This is usually not problematic. Our use of this term is consistent with its use in the C99 standard.
- An *unspecified value* is a value on which this document imposes no restrictions, except that it cannot be a trap value. Our use of this term is consistent with its use in the C99 standard.
- *User* means a “user” of the Libmarpa library. A user of the library is also a programmer, so that in this document, “user” and “programmer” are essentially synonyms.
- *We* (and “us” and ”our”) refer to the authors. As of this writing, there is a one 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.

4.2 Parsing theory preliminaries

This document assumes the reader is familiar with parsing theory. The following exposition is **not** intended an introduction or a reference. Instead, it is intended to serve as a guide to the definitions of parsing terms as used in this document.

Where a narrow or specialized sense of the term is the one that applies within Marpa, that is the only definition given. Marpa also sometimes uses a standard term with a definition which is slightly different from the standard one. “Ambiguous grammar” is one example: See Section 4.11 [Ambiguity], page 10. The term “grammar” itself is another. See [grammar-non-standard], page 5. When a definition is non-standard, this is explicitly pointed out.

Readers who want a textbook or tutorial in parsing theory can look at Mark Jason Dominus's excellent chapter on parsing in the Perl context. See [Bibliography-Dominus-2005], page 104. It is available on-line. Wikipedia is also an excellent place to start. See [Bibliography-Wikipedia], page 105.

A *grammar* is a set of rules, associated with a set of symbols, one of which is distinguished as the start symbol. A *symbol string*, or simply *string* where the meaning is clear, is an ordered series of symbols. The *length* of a string is the number of symbols in it. A symbol string is also called a *sentential form*.

Some of the symbols are terminals. For the purposes of this subsection, a terminal is a symbol which may occur in an input to a parse of a grammar. In a parse, an input is either accepted or rejected. A potential input string, that is, a sentential form which is made up entirely of terminal symbols, is called a *sentence*. The set of sentences that a grammar accepts is the *language* of the grammar.

It is important to note that the term language, as it is used in parsing theory, means something very different from what it means in ordinary use. The meaning of the strings is an essential part of the ordinary idea of what a language is. In ordinary use, the word "language" means far more than a unordered list of its sentences. In parsing terminology, meaning (or *semantics* as it is called) is a separate issue. For parsing theory a language is exactly a set of strings – that and nothing more.

The Marpa definition of a grammar differs slightly from the various standard ones. Standard definitions usually sharply distinguish terminal symbols from non-terminals. Marpa does not. Further discussion of Marpa's handling of terminal is below (see Section 6.3 [Terminals], page 17).

4.3 Stages of parsing

A *recognizer* is a program that determines whether its *input* is in the language of a grammar and a start symbol. A *parser* is a program which finds the structure of that input.

The term *parsing* is used in a strict and a loose sense. *Parsing in the loose sense* is all phases of finding a grammar's structure, including a separate recognition phase if the parser has one. (Marpa does.) If a parser has phases, *parsing in the strict sense* refers specifically to the phase that finds the structure of the input. When the Marpa documents use the term *parsing* in its strict sense, they will speak explicitly of "parsing in the strict sense". Otherwise, *parsing* will mean parsing in the loose sense.

Parsers often use a *lexical analyzer* to convert *raw input*, usually *input text*, into a *token stream*, which is a series of *tokens*. Each token represents a *symbol* of the grammar and has a *value*. A lexical analyzer is often called a *lexer* or a *scanner*, and *lexical analysis* is often called *lexing* or *scanning*.

The series of symbols represented by the series of tokens becomes the *symbol string input* seen by the recognizer. The *symbol string input* is more often called the *input sentence*.

The output of the Marpa parser is a parse forest. See [def-forest], page 10.

4.4 Rules

A standard way of describing rules is Backus-Naur Form, or *BNF*. A rule of a grammar is sometimes called a *rule*. In one common way of writing BNF, a rule looks like this:

```
Expression ::= Term Factor
```

In the rule above, *Expression*, *Term* and *Factor* are symbols. A rule consists of a *left hand side* and a *right hand side*. In a *context-free grammar*, like those Marpa parses, the left hand side of a rule is always a symbol string of length 1. The right hand side of a rule is a symbol string of zero or more symbols. In the example, *Expression* is the left hand side, and *Term* and *Factor* are right hand side symbols.

Left hand side and right hand side are often abbreviated as *RHS* and *LHS*. If the RHS of a rule has no symbols, the rule is called an *empty rule* or an *empty rule*.

In a standard grammar, all rules are BNF rules, as just described. Marpa grammars differ from standard grammars in allowing a second kind of rule: a *sequence rule*. The RHS of a sequence rules is a single symbol, which is repeated zero or more times. Libmarpa allows the application to specify other parameters, including a separator symbol. See Chapter 10 [Sequence rules], page 22.

4.5 Derivations

A *step* of a derivation, or *derivation step*, is a change made to a symbol string by applying one of the rules from the grammar. The rule must be one of those with a LHS that occurs in the symbol string. The result of the derivation step is another symbol string, one in which every occurence of the LHS symbol from the rule is replaced by the RHS of the rule. For example, if *A*, *B*, *C*, *D*, and *X* are symbols, and

```
X ::= B C
```

is a rule, then

```
A X D -> A B C D
```

is a derivation step,

- with "`A X D`" as its beginning,
- "`A B C D`" as its end or result, and
- `X ::= B C` as its rule.

A *derivation* is a sequence of derivation steps. The *length* of a derivation is its length in steps.

- A string *X* *derives* a string *Y* iff there is a derivation of zero or more steps which begins with the string *X* and ends in the string *Y*. In the example above (see [derivation-example], page 6), we say that the symbol string "`A X D`" derives the symbol string "`A B C D`" in one step.
- We say that a first symbol string *directly derives* a second symbol string if and only if there is a derivation of length 1 from the first symbol string to the second symbol string. In the example above (see [derivation-example], page 6), we say that the symbol string "`A X D`" directly derives the symbol string "`A B C D`".
- Every symbol string is said to derive itself in a derivation of length 0. A zero length derivation is a *trivial derivation*.
- A derivation which is not trivial (that is, a derivation which has one or more steps) is a *non-trivial* derivation.
- If a derivation is not trivial or direct, that is, if it has more than one step, then it is an *indirect* derivation.

Technically, a symbol X and a string that consists of only that symbol are two different things. But we often say “the symbol X” as shorthand for “the string of length 1 whose only symbol is X”. For example, if the string containing only the symbol X derives a string Y, we will usually say simply that “X derives Y”.

Wherever symbol or string X derives Y, we may also say X *produces* Y. Derivations are often described as symbol matches. Wherever symbol or string X derives Y, we may also say that Y *matches* X or that X *matches* Y. It is particularly common to say that X matches Y when X or Y is a sentence.

The parse of an input by a grammar is *successful* if and only if, according to the grammar, the start symbol produces the input sentence. The set of all input sentences that a grammar will successfully parse is the *language* of the grammar.

4.6 Nulling

The zero length symbol string is called the *empty string*. The empty string can be considered to be a sentence, in which case it is the *empty sentence*. A string of one or more symbols is *non-empty*. A derivation which produces the empty string is a *null derivation*. A derivation from the start symbol which produces the empty string is a *null parse*.

If a symbol has a null derivation, it is a *nullable symbol*. If the only sentence produced by a symbol is the empty sentence, it is a *nulling symbol*. All nulling symbols are nullable symbols.

If a symbol is not nullable, it is *non-nullable*. If a symbol is not nulling, it is *non-nulling*.

A rule is *nullable* iff it is the rule of the first step of a null derivation. A rule is nullable iff its LHS symbol is nullable.

A rule R is *nulling* iff every derivation whose first step has R as its rule is a null derivation. A rule is nulling iff its LHS symbol is nulling.

If a rule is not nullable, it is *non-nullable*. If a rule is not nulling, it is *non-nulling*.

4.7 Useless rules

If any derivation from the start symbol uses a rule, that rule is called *reachable* or *accessible*. A rule that is not accessible is called *unreachable* or *inaccessible*. If any derivation which results in a sentence uses a rule, that rule is said to be *productive*. A rule that is not productive is called *unproductive*. A rule is productive iff every symbol on its RHS is productive. A symbol is productive iff it is a terminal or it is the LHS of a productive rule. A rule which is inaccessible or unproductive is called a *useless* rule. Marpa can handle grammars with useless rules.

A symbol is *reachable* or *accessible* if it appears in a reachable rule. If a symbol is not reachable, it is *unreachable* or *inaccessible*. A symbol is *productive* if it appears on the LHS of a productive rule, or if it is a nullable symbol. If a symbol is not productive, it is *unproductive*. A symbol which is inaccessible or unproductive is called a *useless* symbol. Marpa can handle grammars with useless symbols.

4.8 Recursion and cycles

If any symbol in the grammar non-trivially produces a symbol string containing itself, the grammar is said to be *recursive*. If any symbol non-trivially produces a symbol string with itself on the left, the grammar is said to be *left-recursive*. If any symbol non-trivially produces a symbol string with itself on the right, the grammar is said to be *right-recursive*. Marpa can handle all recursive grammars, including grammars which are left-recursive, grammars which are right-recursive, and grammars which contain both left- and right-recursion.

A *cycle* is a non-trivial derivation of a string of symbols from itself. If it is not possible for any derivation using a grammar to contain a cycle, then that grammar is said to be *cycle-free*. Traditionally, a grammar is considered useless if it is not cycle-free.

The traditional deprecation of cycles is well-founded. A cycle is the parsing equivalent of an infinite loop. Once a cycle appears, it can be repeated over and over again. Even a very short input sentence can have an infinite number of parses when the grammar is not cycle-free.

For that reason, a grammar which contains a cycle is also called *infinitely ambiguous*. Marpa can parse with grammars which are not cycle-free, and will even parse inputs that cause cycles. When a parse is infinitely ambiguous, Marpa limits cycles to a single loop, so that only a finite number of parses is returned.

4.9 Trees

In this document, unless otherwise stated,

- by *tree*, we mean a *labeled ordered tree*; and
- by *tree node*, we mean a *labeled ordered tree node*.

For brevity, in contexts where the meaning is clear, we refer to a tree node simply as a *node*. Especially when looked at from the point of view of its labels, a node is often called an *instance*.

A node is a pair of tuples:

- The first element tuple of a node is a “label tuple”. The label tuple is a triple of symbol ID, start Earley set ID, and end Earley set ID. For more about the Earley set IDs, see Chapter 6 [Input], page 15.
- The second element tuple of a node is a list (ordered set) of nodes.

We note that this definition of a tree node is recursive.

In the following list of definitions and assertions, let

```
nd = [ [ sym,  start, end ], children ]
```

be a tree node:

- We say that that *sym* is the symbol of *nd*.
- We say that *nd* is an *instance* of the symbol with ID *sym* starting at *start* and ending at *end*.
- We say that *nd* is an *instance* of the symbol with ID *sym* at location *end*.
- We say that the *length* of *nd* is the difference between its start and end, that is `end−start`.
- The length of *nd* is zero iff *start* is the same as *end*. Put another way, the length of *nd* is zero iff `start = end`.
- We say that the elements of *children* are the *children* of *nd*.

- We say that every element of *children* is a *child* of *nd*.
- For brevity, we say that the symbol *sym* is *at end*. Note that this means we consider the location of a symbol to be where it ends.
- *nd* is a *leaf node* iff *children* is the empty list. A leaf node is also call a *leaf*.
- *nd* is an *rule node* iff it is not a leaf node.
- Every node is either a leaf node or a rule node, but no node is both a leaf and a rule node.
- We say that *nd* is a *terminal node* iff *nd* is a leaf node and *sym* is a terminal. A terminal node is also called a *token node*.
- We say that *nd* is a *nulled node* iff *nd* is a leaf node and *sym* is **not** a terminal. A nulled node is also called a *nulling node*.
- Every leaf node is either a nulled node or a terminal node. But, because nullable LHS terminals are not allowed, no node is both nulled and terminal.
- We say that *nd* is a *BNF node* iff *nd* is not a terminal node and *sym* is the LHS of a BNF rule.
- We say that *nd* is a *sequence node* iff *nd* is not a terminal node and *sym* is the LHS of a sequence rule.
- Every node is a terminal node, a BNF node or a sequence node. But no node is more than one of the these three. This is because sequence rules never share a LHS with a BNF rule, and no BNF node or sequence node is a terminal node.
- If *nd* is a rule node, its *LHS* is *sym*.
- If *nd* is a rule node, its *RHS* is the concatenation, from first to last, of the symbols of the nodes in *children*.
- All nulled nodes are zero-length. No terminal node is zero-length.
- We say that *nd* is an instance of *sym* starting at *start* and ending at *end*. We also say that *nd* is an instance of *sym* at *end* or, simply, that *nd* is an instance of *sym*.
- Let *r* be the rule whose LHS is equal to the LHS of *nd*, and whose RHS is equal to the RHS of *nd*. If *nd* is a BNF rule node, there must be such a rule. In that case, We say that *nd* is an instance of *r* starting at *start* and ending at *end*. We also say that *nd* is an instance of *r* at *end* or, simply, that *nd* is an instance of *r*.
- Let *r* be the sequence rule whose LHS is equal to the LHS of *nd*. If *nd* is a sequence rule node, there must be such a rule. In that case, We say that *nd* is an instance of *r* starting at *start* and ending at *end*. We also say that *nd* is an instance of *r* at *end* or, simply, that *nd* is an instance of *r*.
- If *nd* is a nulled instance, that *sym* is *nulled* at location *end* or, simply, say that the symbol *sym* is nulled.

Let *nd1* and *nd2* be two nodes. If *nd2* is a child of *nd1*, then *nd1* is the *parent* of *nd2*.

We define *ancestor* recursively such that *nd1* is the ancestor of a node *nd2* iff one of the following are true:

- *nd1* and *nd2* are the same node. In this case we say that *nd1* is the *trivial ancestor* of *nd2*.
- *nd1* is the parent of an ancestor of *nd2*. In this case we say that *nd1* is a *proper ancestor* of *nd2*.

Simlarly, we define *descendant* recursively such that *nd1* is the descendant of a node *nd2* iff one of the following are true:

- *nd1* and *nd2* are the same node. In this case we say that *nd1* is the *trivial descendant* of *nd2*.
- *nd1* is the parent of an descendant of *nd2*. In this case we say that *nd1* is a *proper descendant* of *nd2*.

A tree is its own *root node*. That implies that, in fact, tree and node are just two different terms for the same thing. We usually speak of trees when we are thinking of the nodes/trees as a collection of nodes, and we speak of nodes when we are more focused on the individual nodes.

A *parse forest* is a set of one or more parse trees. Each tree represents a *parse*.

We have used "parse" as a noun in several senses. Depending on context a "parse" may be

- the process of parsing an input using a grammar,
- a parse tree, or
- a parse forest.

When the meaning of "parse" is not clear in context, we will be explicit about which sense is intended.

[TODO: give example of tree] [TODO: define path] [TODO: define left vs. right] [TODO: define cut] [TODO: define frontier] [TODO: define top-down traversal] [TODO: define bottom-down traversal]

4.10 Traversal

The structure of a parse can be represented as a series of derivation steps from the start symbol to the input. The node at the root of the tree is also called the *start node*.

4.11 Ambiguity

Marpa allows ambiguous grammars. Traditionally we say that a parse is *ambiguous* if, for a given grammar and a given input, more than one derivation tree is possible. However, Marpa allows ambiguous input tokens, which the traditional definition does not take into account. If Marpa used the traditional definition, all grammars would be ambiguous except those grammars which allowed only the null parse.

[TODO: Rewrite two reasons to differ from traditional definition – ambiguous tokens and pruned null forests. Def is that cardinality of forest > 1.]

It is easiest if the Marpa definition and the traditional definition were extensionally equivalent — that is, if Marpa's set of ambiguous grammars was exactly the same as the set of traditionally ambiguous grammars. This can be accomplished by using a slightly altered definition. In the Marpa context, a grammar is *ambiguous* if and only if, for some UNAMBIGUOUS stream of input tokens, that grammar produces more than one parse tree.

4.12 Evaluating a parse

A parser is an algorithm that takes a string of symbols (tokens or characters) and finds a structure in it. Traditionally, that structure is a tree.

Rarely is an application interested only in the tree. Usually the idea is that the string "means" something: the idea is that the string has a *semantics*. Traditionally and most often, the tree is an intermediate step in producing a value, a value which represents the "meaning" or "semantics" of the string. *Evaluating* a tree means finding its semantics.

4.13 Semantics terms

In real life, the structure of a parse is usually a means to an end. Grammars usually have a *semantics* associated with them, and what the user actually wants is the *value* of the parse according to the semantics.

The tree representation is especially useful when evaluating a parse. In the traditional method of evaluating a parse tree, every node which represents a terminal symbol has a value associated with it on input. Recall that nodes are often called "instances" of their symbols or rules. Semantics is associated with instances of rules or of lexemes.

Non-null inner nodes take their semantics from the rule whose LHS they represent. Nulled nodes are dealt with as special cases.

The semantics for a rule describe how to calculate the value of the node which represents the LHS (the parent node) from the values of zero or more of the nodes which represent the RHS symbols (child nodes). Values are computed recursively, bottom-up. The value of a parse tree is the value of its start symbol.

4.14 Application and diagnostic behavior

An *application behavior* is a behavior on which it is intended that the design of applications will be based. In this document, a behavior is an application behavior unless otherwise stated. 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 that 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 that this document suggests that the programmer may attempt in the face of an irrecoverable failure, for purpose of testing, diagnostics and debugging. Diagnostic behaviors are hoped for, rather than expected, and intended to allow the programmer to deal with irrecoverable failures as smoothly as possible. (See Chapter 12 [Failure], page 26.)

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

5 Architecture

5.1 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. A bocages is a way of representing a parse forest.
- 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

5.2 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 some machine-determined limit, such as memory.

An object is the *ancestor* of another object if it is the parent of that object, or if it is the parent of an ancestor of that object. An object is the *descendant* of another object if it is the child of that object, or if it is the child of an descendant of that object. The following three statements are mutually exclusive:

- Object `X` is of class `C`.
- Object `X` has an ancestor of class `C`.
- Object `X` has a descendant of class `C`.

It follows from the definitions of "parent" and "ancestor" that, for any time object class, an object can have at most one ancestor of that class. On the other hand, if an object has descendants in a class, there can be many of them.

An object is a *base* of another object, if it is that object, or if it is the ancestor of the object. For each time object class, an object has at most one base object. For example, a recognizer is its own base recognizer, and has exactly one base grammar.

The *base grammar* of a time object is of special importance. Every time object has a base grammar. A grammar object is its own base grammar. The base grammar of a recognizer is its parent grammar, the one 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.

5.3 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 that 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 that 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.

5.4 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.

6 Input

6.1 Earlemes

6.1.1 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 specified 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.

6.1.2 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 47, and [marpa_r_earleme_complete], page 49.

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.

6.1.3 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 47, and [marpa_r_earleme_complete], page 49.

6.1.4 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 47, and [marpa_r_alternative], page 48.

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.

6.2 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 26 [Advanced input models], page 93.

6.2.1 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.

6.2.2 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 6.2.1 [The standard model of input], page 16.

6.3 Terminals

Traditionally, a terminal symbol is a symbol that may appear in the input. Traditional grammars divide all symbols sharply into terminals and non-terminals: A terminal symbol must **always** be used as a terminal. A non-terminal symbol can **never** be used as a terminal.

In Libmarpa, by default, a symbol is a terminal, and therefore may appear in the input iff both of the following are true:

- The symbol is non-nulling. It is a logical contradiction for a nulling symbol to appear in the input. For this reason, Marpa does not allow it.
- The symbol does not appear on the LHS of any rule.

Marpa's default behavior follows tradition. A now-deprecated feature of Marpa allowed for LHS terminals. See Section 29.1 [LHS terminals], page 99. Most readers will want to stick to Marpa's default behavior, and can and should ignore the possibility of LHS terminals. Even when LHS terminals are allowed, terminals can never be zero length.

In Libmarpa, every terminal instance has a token value associated with it. Token values are `int`'s. Libmarpa does nothing with token values except accept them from the application and return them during parse evaluation.

7 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 that 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 that 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 49, and [marpa_r_start_input], page 47. 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 53.

8 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 22 [Value methods], page 64).

9 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.

10 Sequence rules

Traditionally, grammars only allow BNF rules. Libmarpa allows sequence rules, which express sequences by allowing a single RHS symbol to be repeated.

A sequence rule consists of a LHS and a RHS symbol. Additionally, the application must indicate the minimum number of repetitions. The minimum count must be 0 or 1.

Optionally, a separator symbol may be specified. For example, a comma-separated sequence of numbers

```
1,42,7192,711,
```

may be recognized by specifying the rule `Seq ::= num` and the separator `comma ::= ','`. By default, an optional final separator, as shown in the example above, is recognized, but "proper separation" may also be specified. In proper separation separators must, in fact, come between ("separate") items of the sequence. A final separator is not a separator in the strict sense, and therefore is not recognized when proper separation is in effect. For more on specifying sequence rules, see [marpa_g_sequence_new], page 42.

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

The RHS symbol and the separator, if there is one, must not be nullable. This is 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 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.

A sequence rule must have a dedicated LHS — that is, the LHS of a sequence rule must not be the LHS of any other rule. This implies that the LHS of a sequence rule can never be the LHS of a BNF rule.

The requirement that the LHS of a sequence rule be unique is imposed for reasons similar to those for the prohibition against RHS and separator nullables. Often reuse of the LHS of a sequence rule is simply a mistake. Even when deliberate, reuse of the LHS results in a complex grammar, one which often parses in ways that the programmer did not intend.

A programmer who believes they know what they are doing, and really does want alternative sequences starting at the same input location, can specify this behavior indirectly. They can do this by creating two sequence rules with distinct LHS's:

```
Seq1 ::= Item1
Seq2 ::= Item2
```

and adding a new "parent" LHS which recognizes the sequences as alternatives.

```
SeqChoice ::= Seq1
SeqChoice ::= Seq2
```

11 Nullability

In Libmarpa, there is no direct way to mark a symbol nullable or nulling. All Libmarpa's terminal symbols are non-nullable. By default, Libmarpa's non-terminal symbols are nullable or nulling depending on the rules in which they appear on the LHS. The default behavior for non-terminals can be changed (see Section 29.1 [LHS terminals], page 99), but this is deprecated.

To make a symbol *x* nullable, a user must create an nulling rule whose LHS is *x*. The empty rule is nulling, so that one way a user can ensure *x* is nullable is by making it the LHS of an empty rule. If every rule with *x* on the LHS is nulling, *x* will be not just nullable, but nulling as well.

11.1 Nullability in the valuator

In the valuator, every nulling tree is pruned back to its topmost nulling symbol. This means that there are no nulling rules in the valuator, only nulling symbols. For an example of how this works, see Section 11.4 [Example of nulled symbol], page 24.

While this may sound draconian, the “lost” semantics of the nulled rules and non-topmost nulled symbols are almost never missed. Nulled subtrees cannot contain input, and therefore do not contain token symbols. So no token values are lost when nulled subtrees are pruned, and we are dealing with the semantics of the empty string. See Section 11.3 [Evaluating nulled symbols], page 23.

11.2 Assigning semantics to nulled symbols

Libmarpa leaves the semantics to an upper layer, so that we usually treat semantics as outside the scope of this document. But most upper layers will find that nulled symbols are a corner case for their semantics, and we therefore offer the writers of upper layers some hints.

Typically, upper layers will assign semantics to a LHS symbol based on the rule instance in which the LHS occurs. All nulled symbols are LHS symbols, but the valuator prunes all nulled rules, forcing the application to determine the semantics of a nulled symbol instance based on its symbol. One method of making this determination is the one which is implemented in `Marpa::R2`. Call a grammar *g*; let *x* be a symbol that is nulled in a parse that uses *g*; and call a rule in *g* with *x* on its LHS, an “*x* LHS rule”. `Marpa::R2` assigns a semantics to *x* using the first of following guidelines that applies:

- If every *x* LHS rule in *g* has the same semantics, `Marpa::R2` assigns that shared semantics to *x*.
- If there is an empty *x* LHS rule in *g*, `Marpa::R2` assigns the semantics of that empty rule to *x*.
- If none of the previous guidelines apply, `Marpa::R2` reports an error.

11.3 Evaluating nulled symbols

In theory, the semantics of nulled symbols, like any semantics, can be arbitrarily complex. In practice, we are dealing with the semantics of the empty string, which is literally the "semantics of nothing". If what we are dealing with truly is primarily a parsing problem, we can usually expect that the semantics of nothing will be simple.

The possible subtrees below a nulled symbol can be seen as a set, and that set is a constant that depends on the grammar. Since the input corresponding to the nulled symbol is also a constant (the empty string), the semantics of a nulled symbol will also be constant, with a few exceptions:

- The semantics are non-deterministic.
- The semantics depend on parse location. For example, the nulled symbol instance of *x* at location 5 might mean something different from an nulled instance of *x* at location 50.
- The semantics take into account, not just the input, but its context.

All of these exceptions are unusual or rare. When they do occur, the upper layer can implement the semantics of the nulled symbols with a function or a closure.

11.4 Example of nulled symbol

As already stated, Marpa prunes every null subtree back to its topmost null symbol. Here is an example grammar, with *S* as the start symbol.

```
S ::= L R
L ::= A B X
L ::=
R ::= A B Y
R ::=
A ::=
B ::=
X ::=
X ::= "x"
Y ::=
Y ::= "y"
```

If we let the input be '`x`', we can write the unpruned parse tree in pre-order, depth-first, indenting children below their parents, like this:

```
0: Visible Rule: S := L R
     1: Visible Rule L := A B X
         1.1: Nulled Symbol A
         1.2: Nulled Symbol B
         1.3: Token, Value is "x"
     2: Nulled Rule, Rule R := A B Y
         2.1: Nulled Symbol A
         2.2: Nulled Symbol B
         2.3: Nulled Symbol Y
```

In this example, five symbols and a rule are nulled. The nulled rule and three of the nulled symbols are in a nulled subtree: 2, 2.1, 2.2 and 2.3. Marpa prunes every null subtree back to its topmost symbol, which in this case is the LHS of the rule numbered 2. The pruned tree looks like this:

```
0: Visible Rule: S := L R
     1: Visible Rule L := A B X
         1.1: Nulled Symbol A
         1.2: Nulled Symbol B
         1.3: Token, Value is "x"
     2: LHS of Nulled Rule, Symbol R
```

Nulled nodes 1.1, 1.2 and 2 were all kept, because they are topmost in their nulled subtree. All the other nulled nodes were discarded.

12 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.

12.1 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 that 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.

12.2 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.

12.3 Classifying failure

A Libmarpa application programming failure, unless stated 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 4.14 [Application and diagnostic behavior], page 11) in an effort to handle irrecoverable failures as smoothly as possible.

A Libmarpa application programming failure is not recoverable, unless this document states otherwise.

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 12.12 [Error codes], page 30.)

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**.

12.4 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.

12.5 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.

12.6 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 that detects the hard failure. At a minimum, this diagnostic behavior will be returning from the method that detects the hard failure with the return value specified for hard failure, and setting the error code as specified for hard failure.

12.7 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 12.8 [Library-recoverable hard failure], page 28.

12.8 Library-recoverable hard failure

A *library-recoverable hard failure* is a type of partially recoverable hard failure. Loosely described, it is a hard failure that allows the programmer to continue to use many of the Libmarpa methods in the library, but that 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 that 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 that is not an application behavior is an irrecoverable failure.

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

- All non-Libmarpa interfaces, including calls to other libraries and the C language environment.
- All Libmarpa static method calls.
- All Libmarpa accessor and destructor method calls.
- All Libmarpa mutator and constructor method calls whose base grammar is not the failure grammar.

Note that Libmarpa destructors remain available after a library recoverable failure. An application will often want to destroy all Libmarpa objects whose base grammar is the failure grammar, in order to clear memory of problematic objects.

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 45.

12.9 Ancestry-recoverable hard failure

An *ancestry-recoverable hard failure* is a type of partially recoverable hard failure. An ancestry-recoverable failure allows a superset of the application behaviors allowed by a library-recoverable hard failure. More precisely, let the "failure object" be the object that detected the ancestry-recoverable hard failure. After an ancestry-recoverable hard failure, the following behaviors are no longer applcation behaviors:

- Libmarpa mutator and constructor method calls where the object is the failure object, or one of its descendants.

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

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

- All non-Libmarpa interfaces, including calls to other libraries and the C language environment.
- All Libmarpa static method calls.
- All Libmarpa accessor and destructor method calls.
- All Libmarpa mutator and constructor method calls for time objects that are not the failure object, or one of its descendants.

Note that all Libmarpa destructors remain available after an ancestry-recoverable failure. An application will often want to destroy the failure object and all of its descendants, in order to clear memory of problematic objects.

As an example, users calling `marpa_g_precompute()` will often want to treat a `MARPA_EVENT_EARLEY_ITEM_THRESHOLD` event as if it were an ancestry-recoverable hard failure. See [marpa_g_precompute], page 45.

Library-recoverable failure is a special case of ancestry-recoverable failure. When the failure object is a grammar, ancestry-recoverable failure is synonymous with library-recoverable failure.

12.10 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 49), which has seen extensive usage, is based on Libmarpa’s ability to recover from a `MARPA_ERR_UNEXPECTED_TOKEN_ID` error fully and efficiently,

12.11 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.

12.12 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 24.3 [External error codes], page 80).

How the error code is accessed depends on the method that 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 24.1 [Error methods], page 80). 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.

13 Introduction to the method descriptions

The following chapters describe Libmarpa's methods in detail.

13.1 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.

13.2 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.

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

The words "success" and "failure" are heavily overloaded in these documents. But in contexts where our meaning is clear we will usually abbreviate "method success" and "method failure" to "success" and "failure", respectively.

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 method success, and a potential indication of hard method 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 16.7 [Rank methods], page 43.

13.4 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 24.1 [Error methods], page 80.
- 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 unspecified.
- If a method call succeeds, the value of the error code is unspecified.
- 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.

14 Static methods

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

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 11.0.1, 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*) [Accessor function]

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 unspecified.

15 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***) [Mutator function]

> 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 unspecified.

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

> 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.

16 Grammar methods

16.1 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.

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.

16.2 Creating a new grammar

`Marpa_Grammar marpa_g_new` (*Marpa_Config** *`configuration`*) [Constructor function]

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 35.) 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*) [Mutator function]

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 unspecified. On failure, -2.

16.3 Tracking the reference count of the grammar

`Marpa_Grammar marpa_g_ref` (*Marpa_Grammar g*) [Mutator function]

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`) [Destructor function]

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

16.4 Symbol methods

`Marpa_Symbol_ID marpa_g_start_symbol` (*Marpa_Grammar* `g`) [Accessor function]

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`) [Mutator function]

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`) [Accessor function]

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`) [Accessor function]

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*) [Accessor function]

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`) [Accessor function]

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`) [Accessor function]

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`) [Accessor function]

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`) [Accessor function]

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.

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.

`Marpa_Symbol_ID marpa_g_symbol_new` (*Marpa_Grammar* `g`) [Mutator function]

When successful, creates a new symbol in grammar *g*. The symbol ID's are non-negative integers. Within each grammar, a symbol's ID is unique to that symbol.

Symbols are numbered consecutively, starting at 0. That is, the first successful call of this method for a grammar returns the symbol with ID 0. The *n*'th successful call returns the symbol for a grammar with ID $n-1$. This makes it convenient for applications to store additional information about the symbols in an array.

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

16.5 Rule methods

`int marpa_g_highest_rule_id` (*Marpa_Grammar* `g`) [Accessor function]

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`) [Accessor function]

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`) [Accessor function]

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`) [Accessor function]

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`) [Accessor function]

A rule is a loop rule if it non-trivially produces the string of length one that 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`) [Accessor function]

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`) [Accessor function]

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`) [Accessor function]

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`) [Mutator function]

On success, creates a new external BNF rule in grammar *g*. 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 42. We call `marpa_g_rule_new()` and `marpa_g_sequence_new()` *rule creation methods*.

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.

Each grammar's rule ID's are a consecutive sequence of non-negative integers, starting at 0. This is intended to make it convenient for applications to store additional information about a grammar's rules in an array. Within each grammar, the following is true:

- A rule's ID is unique to that rule.
- The first successful call of a rule creation method returns the rule with ID 0.
- The *n*'th successful call of a rule creation method returns the rule with ID `n−1`.

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`) [Accessor function]

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.

16.6 Sequence methods

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

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 whose value is unspecified because *rule_id* is the ID of a BNF rule. Applications that 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 41.

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`) [Accessor function]

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 that 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`) [Mutator function]

When successful, adds a new sequence rule to grammar *g*, and returns its ID. In addition to sequence rules, Marpa also allows BNF rules, which are created by the `marpa_g_rule_new()` method. See [marpa_g_rule_new], page 40. We call `marpa_g_rule_new()` and `marpa_g_sequence_new()` *rule creation methods*. For details on the use of sequence rules, see Chapter 10 [Sequence rules], page 22.

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.

Each grammar's rule ID's are a consecutive sequence of non-negative integers, starting at 0. This is intended to make it convenient for applications to store additional information about a grammar's rules in an array. Within each grammar, the following is true:

- A rule's ID is unique to that rule.
- The first successful call of a rule creation method returns the rule with ID 0.
- The *n*'th successful call of a rule creation method returns the rule with ID `n−1`.

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. The sequence RHS, or item, is restricted to a single symbol, and that symbol cannot be nullable. If *separator_id* is non-negative, it is a separator symbol, which cannot be nullable. *flags* is a bit vector. Use of any other bit except `MARPA_PROPER_SEPARATION` results in undefined behavior.

By default, a sequence rule recognizes a trailing separator. If `flags & MARPA_PROPER_SEPARATION` is non-zero, separation is “proper”. Proper separation means the the rule does not recognize a trailing separator. Specifying proper separation has no effect unless a separator symbol has also been specified.

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`.

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`) [Accessor function]

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 that 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`) [Accessor function]

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.

16.7 Rank methods

`Marpa_Rank marpa_g_default_rank` (*Marpa_Grammar* `g`) [Accessor function]

On success, returns the default rank of the grammar *g*. For more about the default rank of a grammar, see [marpa_g_default_rank_set], page 43.

Return value: On success, returns the default rank of the grammar, 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 default rank of the grammar 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. See [marpa_g_error], page 80.

`Marpa_Rank marpa_g_default_rank_set` (*Marpa_Grammar* `g`, *Marpa_Rank* `rank`) [Mutator function]

On success, sets the default rank of the grammar *g* to *rank*. 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 *rank* 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. See [marpa_g_error], page 80.

`Marpa_Rank marpa_g_symbol_rank` (*Marpa_Grammar* `g`, *Marpa_Symbol_ID* `sym_id`) [Accessor function]

When successful, returns the rank of the symbol with ID *sym_id*. When a symbol is created, its rank is initialized to the default rank of the grammar.

Return value: On success, returns a symbol rank, 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 symbol 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 80.

`Marpa_Rank marpa_g_symbol_rank_set` (*Marpa_Grammar g, Marpa_Symbol_ID* `sym_id`*, Marpa_Rank* `rank`) [Mutator function]

When successful, sets the rank of the symbol with ID *sym_id* to *rank*. When a symbol is created, its rank is initialized to the default rank of the grammar.

Return value: On success, returns *rank*, 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 *rank* may be −2, and in this case 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 80.

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

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.

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 80.

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

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 80.

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

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 45.

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**) [Mutator function]

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.

16.8 Precomputing the Grammar

`int marpa_g_has_cycle` (*Marpa_Grammar* **g**) [Accessor function]

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**) [Accessor function]

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**) [Mutator function]

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 that 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 that occur on success. Events may be queried using the `marpa_g_event()` method. See [marpa_g_event], page 71.

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 45.

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 that 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 `MARPA_ERR_NULLING_TERMINAL` occurs only if LHS terminals are enabled. The LHS terminals feature is deprecated. See Section 29.1 [LHS terminals], page 99. Error code `MARPA_ERR_NULLING_TERMINAL` is library-recoverable. One or more `MARPA_EVENT_NULLING_TERMINAL` events will occur iff this method fails with error code `MARPA_ERR_NULLING_TERMINAL`. See Section 29.1.5 [Nulling terminals], page 100.

Among the other error codes that 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 24.3 [External error codes], page 80.

Return value: On success, a non-negative number, whose value is otherwise unspecified. 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.

17 Recognizer methods

17.1 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.

17.2 Creating a new recognizer

`Marpa_Recognizer marpa_r_new` (*Marpa_Grammar g*) [Constructor function]

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
- the values of the latest and current earleme will be unspecified.

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

17.3 Keeping the reference count of a recognizer

`Marpa_Recognizer marpa_r_ref` (*Marpa_Recognizer r*) [Mutator function]

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*) [Destructor function]

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.

17.4 Life cycle mutators

`int marpa_r_start_input` (*Marpa_Recognizer r*) [Mutator function]

When successful, does the following:

- Readies *r* to accept input.
- Completes the first Earley set. The ID of the first Earley set is 0, and it 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 7 [Exhaustion], page 18.
- May generate one or more `MARPA_EVENT_SYMBOL_NULLED`, `MARPA_EVENT_SYMBOL_PREDICTED`, or `MARPA_EVENT_SYMBOL_EXPECTED` events. See Chapter 23 [Events], page 71.

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

`int marpa_r_alternative` (*Marpa_Recognizer* ***`r`***, *Marpa_Symbol_ID* ***`token_id`***, *int* ***`value`***, *int* ***`length`***) [Mutator 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 29.2 [Valued and unvalued symbols], page 100.

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`) [Mutator 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 53.
- 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 26 [Advanced input models], page 93.

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 26 [Advanced input models], page 93.
- The value of the furthest earleme is never changed by a call to `marpa_r_earleme_complete()`.
- Clears the event queue of any events that 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 23 [Events], page 71.

- 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 ancestry-recoverable failure. See [marpa_r_earley_item_warning_threshold_set], page 53.
- 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 7 [Exhaustion], page 18.
- Leaves the parse in a state where no terminals are expected or anticipated.
- Clears the event queue of any events that 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.

This method is commonly called at the top of a loop. Almost all applications will want to check the return value and take special action in case of a value other than zero. If the value is greater than zero, an event will have occurred and almost all applications should react to `MARPA_EVENT_EARLEY_ITEM_THRESHOLD` events, as described above, and to unexpected events. If the value is less than zero, it may be due to an irrecoverable error, and only in very unusual circumstances will an application wish to ignore these.

How an application reacts to exhaustion will depend on the kind of parsing it is doing:

- Very often an application knows the length of its input in advance. In this case, the application will treat exhaustion before the end of input as a parse error. Depending on the application, exhaustion on success at the end of input might be ignored.
- An exhaustion-loving application that does not know the length of its input will often want terminate the parse in case of exhaustion on method success, treating exhaustion as an end-of-input indicator. These application will usually want to treat exhaustion on method failure as a parse error.
- Occasionally, an exhaustion-hating application may not know the length of its input in advance. Since these applications will not know from the length of the

input, or from exhaustion, that they are at end of input, they will need some other way of determining this. One way they may do this is with a `MARPA_EVENT_SYMBOL_COMPLETED` event. Typically, these applications will treat exhaustion on method failure and exhaustion before the end of input as parse errors. They may wish to ignore exhaustion on method success at the end of input.

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.

17.5 Location accessors

`Marpa_Earleme marpa_r_current_earleme` (*Marpa_Recognizer* `r`) [Accessor function]

Successful iff input has started. If input has not started, returns soft failure.

Return value: On success, the current earleme, which is always non-negative. On soft failure, −1. Never returns a hard failure.

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

On success, returns the earleme of the Earley set with ID `set_id`. The ID of an Earley set ID is also called its ordinal. In the default, token-stream model, Earley set ID and earleme are always equal, but this is not the case in other input models.

Hard fails if there is no Earley set whose ID is *set_id*. This hard failure is fully recoverable. If *set_id* was negative, the error code of the hard failure is `MARPA_ERR_INVALID_LOCATION`. If *set_id* is greater than the ordinal of the latest Earley set, the error code of the hard failure is `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*, which is always non-negative. On hard failure, −2. The hard failures with error codes `MARPA_ERR_INVALID_LOCATION` and `MARPA_ERR_NO_EARLEY_SET_AT_LOCATION` are fully recoverable.

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

On success, returns the "integer value" of *earley_set*. For more about the integer value of an Earley set, see [marpa_r_earley_set_values], page 52.

Return value: On success, returns the the integer value of *earley_set*, and sets the error code to `MARPA_ERR_NONE`. On hard failure, returns −2, and sets the error code to the error code of the hard failure, which will never be `MARPA_ERR_NONE`. Note that −2 is a valid "integer value" for an Earley set, 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 80.

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

On success, does the following:

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

The "value" and "pointer" of an Earley set are an arbitrary integer and an arbitrary pointer. Libmarpa never examines them and the application is free to use them for its own purposes. In an application with a character-per-earleme input model, for example, the integer value of the Earley set can used to store the codepoint of the current character. In a traditional token-per-earleme input model, the integer and pointer values could be used to track 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 integer value defaults to -1, and the pointer value defaults to `NULL`. The Earley set value and pointer can be set using the `marpa_r_latest_earley_set_values_set()` method. See [marpa_r_latest_earley_set_values_set], page 52.

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

`unsigned int marpa_r_furthest_earleme` (*Marpa_Recognizer* `r`) [Accessor function]

Return value: The furthest earleme. Always succeeds.

`Marpa_Earley_Set_ID marpa_r_latest_earley_set` (*Marpa_Recognizer* `r`) [Accessor function]

Returns the Earley set ID of the latest Earley set. The ID of an Earley set ID is also called its ordinal. Applications that want the value of the latest earleme can convert this value using the `marpa_r_earleme()` method. See [marpa_r_earleme], page 51.

Return value: The ID of the latest Earley set. Always succeeds.

`int marpa_r_latest_earley_set_value_set (` *Marpa_Recognizer* `r`, *int* `value`) [Mutator function]

Sets the "integer value" of the latest Earley set to *value*. For more about the integer value of an Earley set, see [marpa_r_earley_set_values], page 52.

Return value: On success, returns the newly set integer value of the latest earley set, and sets the error code to `MARPA_ERR_NONE`. On hard failure, returns -2, and sets the error code to the error code of the hard failure, which will never be `MARPA_ERR_NONE`. Note that -2 is a valid "integer value" for an Earley set, 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 80.

`int marpa_r_latest_earley_set_values_set (` *Marpa_Recognizer* `r`, *int value*, *void* pvalue*) [Mutator 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 [marpa_r_earley_set_values], page 52.

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

17.6 Other parse status methods

`int marpa_r_earley_item_warning_threshold` (*Marpa_Recognizer* `r`) [Accessor function]

For details about the "earley item warning threshold", see [marpa_r_earley_item_warning_threshold_set], page 53.

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

`int marpa_r_earley_item_warning_threshold_set` (*Marpa_Recognizer* `r`, *int* `threshold`) [Mutator function]

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 78.

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`) [Accessor function]

A parser is "exhausted" if it cannot accept any more input. See Chapter 7 [Exhaustion], page 18.

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

`int marpa_r_terminals_expected` (*Marpa_Recognizer* `r`, *Marpa_Symbol_ID** `buffer`) [Accessor 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 an array of `Marpa_Symbol_ID`'s whose length 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*, which is always non-negative. On hard failure, −2.

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

On success, does the folloing:

- If *symbol_id* is not the ID of a terminal symbol, returns 0.
- If *symbol_id* is the ID of a terminal symbol, but that symbol is **not** expected at the current earleme, returns 0.
- If *symbol_id* is the ID of a terminal symbol, but that symbol **is** expected at the current earleme, returns 1.

Hard fails if the symbol with ID *symbol_id* does not exist.

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

18 Progress reports

It is an important property of the Marpa algorithm that the Earley sets are added one at a time, so that before we have started the construction of the Earley set at `n+1`, we know the full state of the parse at and before location `n`. Libmarpa's progress reports allow access to the Earley items in an Earley set.

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

To step through the Earley items, use the `marpa_r_progress_item()` method.

`int marpa_r_progress_report_reset` (*Marpa_Recognizer* `r`) [Mutator function]

On success, sets the current vertex of the report traverser to the null vertex. For more about the report traverser, including details about the current and null vertices, see [marpa_r_progress_report_start], page 55.

This method is not usually needed. Its effect is to leave the traverser in the same state as it is immediately after the `marpa_r_progress_report_start()` method. Loosely speaking, it allows the traversal to "start over".

Hard fails if the recognizer is not started, or if no progress report traverser is active.

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`) [Mutator function]

Creates a progress report traverser in recognizer *r* for the Earley set with ID *set_id*. A *progress report traverser* is a non-empty directed cycle graph whose vertices consist of the following:

- For every Earley item, exactly one vertex that corresponds to that Earley item. This vertex is called the *progress report item* or, when the meaning is clear, the *report item*. In this method description, we will say the the number of report items is `n`, and we will write `ritem[i]` for the `i`'th report item.
- A special vertex, called the *null vertex*, which does not correspond to any Earley item. In this method description, we will write `null` for the null vertex.

There may be no Earley items in an Earley set, and therefore a progress report traverser may contain no report items. A progress report traverser with no report items is called a "trivial traverser". A trivial traverser has exactly one edge: `(null, null)`.

The edges of a non-trivial traverser are

- `(null, ritem[0])`,
- `(ritem[n-1], null)`, and
- for every `0 <= i < v-1`, `(ritem[i-1], ritem[i])`.

This implies that every vertex has exactly one direct successor. The report items are a subgraph, and this graph can be seen as inducing the sequence `ritem[0] ... ritem[n-1]`.

When a progress report traverser is active, one vertex is distinguished as the *current vertex*, which we will write as `current`. We call the direct successor of the current vertex, the *next vertex*.

On success, does the following:

- If a progress report traverser was active in this recognizer before this method call, it is destroyed and its memory is freed..
- Creates a new progress report traverser from the Earley items for the Earley set with ID *set_id*.
- Activates the newly created progress report traverser, setting the current vertex to the null vertex. Intuitively, in a non-trivial traverser, this can be thought of as positioning the traverser before the first report item.
- Returns `n`, the number of report items. `n` may be zero.

Hard fails if no Earley set with ID *set_id* exists. 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, which will always be non-negative. On hard failure, −2.

`int marpa_r_progress_report_finish` (*Marpa_Recognizer* `r`) [Mutator function]

On success, destroys the progress report traverser for recognizer *r*, freeing its memory. For details about the report traverser, see [marpa_r_progress_report_start], page 55.

It is often not necessary to call this method. `marpa_r_progress_report_start()` destroys any previously existing progress report. And, when a recognizer is destroyed, its progress report is destroyed as a side effect.

Hard fails if no progress report is active.

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

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

This method allows access to the data for the next progress report item of a progress report. For details about progress reports, see [marpa_r_progress_report_start], page 55.

In the event of success:

- Advances the current vertex to the next vertex. More precisely, let `c_before` be the vertex that is the current vertex immediately before the call to this method. The report item traverser has exactly one edge such that `c_before` is its first element. Let this edge be `(c_before,c_after)`. This method sets the current vertex to `c_after`. In this method description, we will write `current` as an alias for `c_after`.
- `current` will be a report item vertex and therefore there will be an Earley item corresponding to `current`.
- Writes the "cooked dot position" of the Earley item corresponding to `current` to the location pointed to by the *position* argument.

- Writes the origin of the Earley item corresponding to `current` to the location pointed to by the *origin* argument.
- Returns the rule ID of the Earley item corresponding to `current`.

The "cooked dot position" is

- the standard 0-based start-relative dot position, if the dotted rule is **not** a completion; and
- the end-relative dot position, where the last position is -1, if the dotted rule **is** a completion.

Use of the cooked dot position allows an application to quickly determine if the dotted rule is a completion. The cooked dot position is -1 iff the dotted rule is a completion.

In the event of soft failure:

- `current` is the null vertex.
- Sets the error code to `MARPA_ERR_PROGRESS_REPORT_EXHAUSTED`.
- Leaves unchanged the locations pointed to by the *position* and *origin* arguments.
- Returns -1.

In addition to watching for soft failure, the application can use the item count returned by `marpa_r_progress_report_start()` to determine when the last item has been seen.

Return value: On success, the rule ID of the progress report item, which is always non-negative. On soft failure, -1. If either the *position* or the *origin* argument is `NULL`, or on other hard failure, -2.

19 Bocage methods

19.1 Overview

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.

19.2 Bocage data structure

A bocage is a data structure containing the parses found by processing the input according to the grammar. It is related to a parse forest, but is in a form that is more compact and easily traversable. “Bocage” is our term, and we discovered this structure independently, but our work was preceded by Elizabeth Scott. And, unlike us, Prof. Scott did the all-important work of documenting it and providing the appropriate mathematical apparatus. See Section 25.1 [Elizabeth Scott’s SPPFs], page 91.

The bocage contains the data for the parse trees whose root is an instance of the start symbol that begins at Earley set 0 and ends at the *end of parse Earley set*. Applications usually use the Earley set at the current earleme as the “end of parse Earley set”, so that the bocage is for parses of the entire input. But some applications may be interested in parsing prefixes of the input, and these applications can choose other end of parse Earley sets in their constructor. See [marpa_b_new], page 58.

19.3 Creating a new bocage

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

On success, the following is the case:

- If `earley_set_ID` is non-negative, creates a new bocage object, whose “end of parse Earley set” is the Earley set with ID `earley_set_ID`.
- If `earley_set_ID` is −1, creates a new bocage object, whose “end of parse Earley set” is the Earley set at the current earleme.
- The new bocage object has a reference count of 1.
- The reference count of its parent recognizer object, *r*, is increased by 1.

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

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

19.4 Reference counting

`Marpa_Bocage marpa_b_ref` (*Marpa_Bocage* `b`) [Mutator function]

On success, increases the reference count by 1. This method is not needed by most applications.

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

`void marpa_b_unref` (*Marpa_Bocage* `b`) [Destructor 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.

19.5 Accessors

`int marpa_b_ambiguity_metric` (*Marpa_Bocage* `b`) [Accessor function]

On success, returns an ambiguity metric. If the parse is unambiguous, the metric is 1. If the parse is ambiguous, the metric is 2 or greater, and is otherwise unspecified. See Section 28.4 [Better defined ambiguity metric], page 96.

Return value: On success, the ambiguity metric, which is always non-negative. On hard failure, −2.

`int marpa_b_is_null` (*Marpa_Bocage* `b`) [Accessor function]

Return value On success, a non-negative integer: 1 or greater if the bocage **is** for a null parse, and 0 if the bocage is **not** for a null parse. On hard failure, −2.

20 Ordering methods

20.1 Overview

Before iterating through the parse trees in the bocage, the parse trees 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.

20.2 Freezing the ordering

An ordering is *frozen* under the following circumstances:

- The first tree iterator is created using the ordering. See [marpa_t_new], page 62.
- `marpa_o_ambiguity_metric()` is successfully called. See [marpa_o_ambiguity_metric], page 60.
- `marpa_o_rank()` is successfully called. See [marpa_o_rank], page 61.

A frozen ordering cannot be changed. There is no way to "unfreeze" an ordering.

20.3 Creating an ordering

`Marpa_Order marpa_o_new (` *Marpa_Bocage* `b)` [Constructor function]

On success, does the following:

- Creates a new ordering object, with a reference count of 1.
- Increases the reference count of its parent bocage object, *b*, by 1.

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

20.4 Reference counting

`Marpa_Order marpa_o_ref (` *Marpa_Order* `o)` [Mutator function]

On success, increases the reference count by 1. Not needed by most applications.

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

`void marpa_o_unref (` *Marpa_Order* `o)` [Destructor function]

Decreases the reference count by 1, destroying *o* once the reference count reaches zero.

20.5 Accessors

`int marpa_o_ambiguity_metric (`*Marpa_Order* `o)` [Accessor function]

On success, returns an ambiguity metric. If the parse is unambiguous, the metric is 1. If the parse is ambiguous, the metric is 2 or greater, and is otherwise unspecified. See Section 28.4 [Better defined ambiguity metric], page 96.

If "high rank only" is in effect, this ambiguity metric may differ from that returned by `marpa_b_ambiguity_metric()`. In particular, a "high rank only" ordering may be unambiguous even if its base bocage is ambiguous. But note also, because multiple parses choices may have the same rank, a "high rank only" ordering may be ambiguous.

If the ordering is not already frozen, it will be frozen on return from `marpa_o_ambiguity_metric()`. For our purposes, `marpa_o_ambiguity_metric()` is considered an "accessor", because it treats its ordering as if it was frozen before the call to `marpa_o_ambiguity_metric()`.

Return value: On success, the ambiguity metric, which is non-negative. On hard failure, −2.

`int marpa_o_is_null` (*Marpa_Order* `o`) [Accessor function]

Return value: On success: A number greater than or equal to 1 if the ordering is for a null parse; otherwise, 0. On hard failure, −2.

20.6 Non-default ordering

`int marpa_o_high_rank_only` (*Marpa_Order* `o`) [Accessor function]

On success, returns, the "high rank only" flag of ordering *o*. See [marpa_o_high_rank_only_set], page 61.

Return value: On success, the value of the "high rank only" flag, which is a boolean. On hard failure, −2.

`int marpa_o_high_rank_only_set` (*Marpa_Order* `o`, *int* `flag`) [Mutator function]

Sets 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 until ranking is turned on using the `marpa_o_rank()` method.

Hards fails if the ordering is frozen.

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

`int marpa_o_rank` (*Marpa_Order* `o`) [Mutator function]

By default, the ordering of parse trees is arbitrary. On success, the following happens:

- The ordering is 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.
- The ordering is *frozen*. See Section 20.2 [Freezing the ordering], page 60.

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

21 Tree methods

21.1 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.

21.2 Creating a new tree iterator

`Marpa_Tree marpa_t_new` (*Marpa_Order* `o`) [Constructor function]

On success, does the following:

- Creates a new tree iterator, with a reference count of 1.
- Increases the reference count of its parent ordering object, *o*, by 1.
- Positions the new tree iterator before the first parse tree.

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

21.3 Reference counting

`Marpa_Tree marpa_t_ref` (*Marpa_Tree* `t`) [Mutator function]

On success, increases the reference count by 1. Not needed by most applications.

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

`void marpa_t_unref` (*Marpa_Tree* `t`) [Destructor function]

Decreases the reference count by 1, destroying *t* once the reference count reaches zero.

21.4 Iterating through the trees

`int marpa_t_next` (*Marpa_Tree* `t`) [Mutator function]

On success, 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, soft fails, and sets the error code to `MARPA_ERR_TREE_EXHAUSTED`. See Section 28.6 [Orthogonal treatment of soft failures], page 96.

It the tree iterator is paused, hard fails, and sets the error code to `MARPA_ERR_TREE_PAUSED`. This hard failure is fully recoverable. See [marpa_v_new], page 66.

Return value: On success, a non-negative value. On soft failure, −1. On hard failure, −2. The hard failure with error code `MARPA_ERR_TREE_PAUSED` is fully recoverable.

`int marpa_t_parse_count ( ` *Marpa_Tree* `t)` [Accessor function]

Returns the count of the number of parse trees traversed so far. The count includes the current iteration of the tree. 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.

22 Value methods

22.1 Overview

The archetypal application needs a value object (or *valuator*) to produce the value of the parse tree. To create a valuator, use the `marpa_v_new()` 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. See Section 22.9 [Basic step accessors], page 68, see Section 22.2 [How to use the valuator], page 64, and see Section 22.7 [Stepping through the valuator], page 67.

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

22.2 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 that 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 that are capable of being any of these things and more.

22.3 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. The worst problem is that 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. The ability to debug can make the difference between code that works and code that 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. We are under no illusion that direct use of Libmarpa's valuator will be found satisfactory by most Libmarpa users, even those using the C language. Libmarpa's valuator is intended to be used via an upper layer, one that **does** know about semantics.

22.4 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 C89 standard-conformant C application. This assumption is convenient on two grounds. First, this will be the intended use for many readers. Second, standard-conformant C89 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. See Section 22.4.1 [Sizing the stack], page 66.

22.4.1 Sizing the stack

In our discussion of the stack handler for the valuator, we will treat the stack as a 0-based array. If an implementation applies Libmarpa's step instructions literally, using a physical stack, it must make sure that all locations in the stack are initialized. The range of locations that must be initialized is from stack location 0 to the "end of stack" location. The result of the parse tree is always stored in stack location 0, so that a stack location 0 is always needed. Therefore, the end of stack location is always a specified value, and greater than or equal to 0. The end of stack location must also be greater than or equal to

- `marpa_v_result(v)` for every `MARPA_STEP_TOKEN` step,
- `marpa_v_result(v)` for every `MARPA_STEP_NULLING_SYMBOL` step, and
- `marpa_v_arg_n(v)` for every `MARPA_STEP_RULE` step.

In practice, an application will often extend the stack as it iterates through the steps, initializing the new stack locations as they are created.

Note that our requirement is not merely that the stack locations exist and be writable, but that they be initialized. This is necessary for C89 conformance. Because of write optimizations in our implementation, the first access to any stack location may be a read. C89 allows trap values, so that a read of an uninitialized location could result in undefined behavior. See Section 25.5 [Trap representations], page 92.

22.5 Creating a new valuator

`Marpa_Value marpa_v_new (` *Marpa_Tree* `t )` [Constructor function]

On success, does the following:

- Creates a new valuator. The parent object of the new valuator will be the tree iterator *t*.
- Sets the reference count of the new valuator to 1.
- Sets new valuator to *active*. A valuator is always either active or inactive.
- Increases the reference count of *t* by 1.
- "Pauses" the parent tree iterator.

As long as a parent tree iterator is *paused* `marpa_t_next()` will not succeed, and therefore the parent tree iterator cannot move on to a new parse tree. Many valuators can share the same parent parse tree. 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 hard failure, `NULL`.

22.6 Reference counting

`Marpa_Value marpa_v_ref (`*Marpa_Value v*`)` [Mutator function]

On success, increases the reference count by 1. Not needed by most applications.

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

`void marpa_v_unref (`*Marpa_Value v*`)` [Destructor function]

Decreases the reference count by 1, destroying *v* once the reference count reaches zero.

22.7 Stepping through the valuator

`Marpa_Step_Type marpa_v_step (` *Marpa_Value v*`)` [Mutator function]

Steps through the tree in depth-first, left-to-right order. On success, does the following:

- Takes the valuator to the next *step* in its life cycle.
- Sets and returns the *step type*, which is a non-negative integer. The C type for a step type is `Marpa_Step_Type`.

The step type tells the application how it expected to act on the step. See Section 22.8 [Valuator step types], page 67. Steps are often referred to along with their step type so that, for example, we say "a `MARPA_STEP_RULE` step" to refer to a step whose step type is `MARPA_STEP_RULE`.

When the iteration through the steps is finished, the step type is `MARPA_STEP_INACTIVE`. At this point, we say that the valuator is *inactive*. Once a valuator becomes inactive, it stays inactive.

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

22.8 Valuator step types

`Marpa_Step_Type MARPA_STEP_RULE` [Accessor macro]

MARPA_STEP_RULE is the step type for for a rule node. The application should perform its equivalent of rule execution.

- The "child values" for this step will be in the stack locations from `marpa_v_arg_0(v)` to `marpa_v_arg_n(v)`.
- The rule for this step will be `marpa_v_rule(v)`.
- The result of this step should be written to stack location `marpa_v_result(v)`. Typically, the result of this step is determined by executing the semantics for its rule on its child values.
- 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` [Accessor macro]

MARPA_STEP_TOKEN is the step type for a token node. The application's equivalent of the evaluation of the semantics of a non-null token should be performed.

- The application's value for the token whose ID is at stack location `marpa_v_token(v)`.
- Libmarpa will already have a"token value" for the token in this step, as was set by the `marpa_r_alternative()` method. See [marpa_r_alternative], page 48. Libmarpa's "token value" will be in stack location `marpa_v_token_value(v)`.
- The result of the applications's evaluation of the semantics of this token should be placed in stack location `marpa_v_result(v)`. Often, an application will simply copy Libmarpa's "token value" to stack location `marpa_v_result(v)`.

`Marpa_Step_Type MARPA_STEP_NULLING_SYMBOL` [Accessor macro]

MARPA_STEP_RULE is the step type for for a nulled node. The application's equivalent of the evaluation of the semantics of a nulling token should be performed.

- The ID of the nulling symbol is at stack location `marpa_v_symbol(v)`.
- The application's value for this nulling symbol instance should be placed in stack location `marpa_v_result(v)`. Often, an application will assign a fixed value to each nullable symbol, and will simply copy this fixed value to stack location `marpa_v_result(v)`.

The use of the word "nulling" in the step type name `MARPA_STEP_NULLING_SYMBOL` is problematic: While the node must be zero-length (nulled or nulling), the node's symbol need not be nulling: it may be nullable. See Section 28.1 [Nulling versus nulled], page 96.

`Marpa_Step_Type MARPA_STEP_INACTIVE` [Accessor macro]

When this is the step type, the valuator has gone through all of its steps and is now inactive. The value of the parse tree 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. Once a valuator becomes inactive, it stays inactive.

`Marpa_Step_Type MARPA_STEP_INITIAL` [Accessor macro]

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

`Marpa_Step_Type MARPA_STEP_INTERNAL1` [Accessor macro]
`Marpa_Step_Type MARPA_STEP_INTERNAL2` [Accessor macro]
`Marpa_Step_Type MARPA_STEP_TRACE` [Accessor macro]

These step types are reserved for internal purposes.

22.9 Basic step accessors

This section describes the accessors that are basic to stack manipulation.

`int marpa_v_arg_0` (*Marpa_Value v*) [Accessor macro]

Return value: For a `MARPA_STEP_RULE` step, the stack location where the value of first child can be found. For other step types, an unspecified value. Always succeeds.

`int marpa_v_arg_n` (*Marpa_Value v*) [Accessor macro]

Return value: For a `MARPA_STEP_RULE` step, the stack location where the value of the last child can be found. For other step types, an unspecified value. Always succeeds.

`int marpa_v_result` (*Marpa_Value v*) [Accessor macro]

Return value: For `MARPA_STEP_RULE`, `MARPA_STEP_TOKEN`, and `MARPA_STEP_NULLING_SYMBOL` steps, the stack location where the result of the semantics should be placed. For other step types, an unspecified value. Always succeeds.

`Marpa_Rule_ID marpa_v_rule` (*Marpa_Value v*) [Accessor macro]

Return value: For the `MARPA_STEP_RULE` step, the ID of the rule. For other step types, an unspecified value. Always succeeds.

`Marpa_Step_Type marpa_v_step_type` (*Marpa_Value v*) [Accessor macro]

This macro is usually not needed since its return value is the same as the value that `marpa_v_step()` returns on success.

Return value: The current step type: `MARPA_STEP_TOKEN`, `MARPA_STEP_RULE`, etc. Always succeeds.

`Marpa_Symbol_ID marpa_v_symbol` (*Marpa_Value v*) [Accessor macro]

Return value: For the `MARPA_STEP_NULLING_SYMBOL` step, the ID of the symbol. The value returned is the same as that returned by the `marpa_v_token()` macro. For other step types, an unspecified value. Always succeeds.

`Marpa_Symbol_ID marpa_v_token` (*Marpa_Value v*) [Accessor macro]

Return value: For the `MARPA_STEP_TOKEN` step, the ID of the token. The value returned is the same as that returned by the `marpa_v_symbol()` macro. For other step types, an unspecified value. Always succeeds.

`int marpa_v_token_value` (*Marpa_Value v*) [Accessor macro]

Return value: For the `MARPA_STEP_TOKEN` step, the "token value" that was assigned to the token by the `marpa_r_alternative()` method. See [marpa_r_alternative], page 48. For other step types, an unspecified value. Always succeeds.

22.10 Step location accessors

This section describes step accessors that are not basic to stack manipulation. They provide Earley set location information about the parse tree.

A step's location in terms of Earley sets is called its ES location. Every ES location is the ID of an Earley set. ES location is only relevant for steps that correspond to tree nodes.

Every tree node has both a start ES location and an end ES location. The start ES location is the first ES location of that parse node.

The end ES location of a leaf is the ES location where the next leaf symbol in the fringe of the current parse tree would start. Typically, this is the location where a leaf node actually starts but, toward the end of a parse, there may not be an actual next leaf node.

The start ES location of a MARPA_RULE_STEP is the start ES location of its first child node in the current parse tree. The end ES location of a MARPA_RULE_STEP is the end ES location of its last child node in the current parse tree.

`Marpa_Earley_Set_ID marpa_v_es_id` (*Marpa_Value v*) [Accessor macro]

Return value: If the current step type is MARPA_STEP_RULE, MARPA_STEP_TOKEN, or MARPA_STEP_NULLING_SYMBOL, the return value is the end ES location of the parse node. If the current step type is anything else, or if the valuator is inactive, the return value is unspecified.

`Marpa_Earley_Set_ID marpa_v_rule_start_es_id` (*Marpa_Value v*) [Accessor macro]

Return value: If the current step type is MARPA_STEP_RULE, the start ES location of the rule node. If the current step type is anything else, or if the valuator is inactive, the return value is unspecified.

`Marpa_Earley_Set_ID marpa_v_token_start_es_id` *(Marpa_Value v)* [Accessor macro]

Return value: If the current step type is MARPA_STEP_TOKEN or MARPA_STEP_NULLING_SYMBOL, the start ES location of the leaf node. If the current step type is anything else, or if the valuator is inactive, the return value is unspecified.

For every parse node of the current parse tree, the Earley set length (ES length) of the node is the end ES location, less the start ES location. The ES length of a nulled node is always 0.

If v is a valuator whose current step type is MARPA_STEP_NULLING_SYMBOL, it is always the case that

```
marpa_v_token_start_es_id(v) == marpa_v_es_id(v)
```

If v is a valuator whose current step type is MARPA_STEP_RULE or MARPA_STEP_TOKEN, it is always the case that

```
marpa_v_token_start_es_id(v) <= marpa_v_es_id(v)
```

For the following examples,

- let `Null` be a nulling symbol,
- let `Tok` be a non-nullable symbol, and
- let the notation `Sym@m-n` indicate that the symbol `Sym` has ES start location `m` and ES end location `n`.

Ordered from left to right, a possible fringe is

```
Null@0-0, Tok@0-1, Null@1-1, Tok@1-2, Null@2-2
```

Another example is

```
Null@0-0, Null@0-0, Tok@0-1, Null@1-1, Null@1-1, Tok@1-2,
Null@2-2, Null@2-2
```

In this second example note that when a nulled leaf immediately follows another nulled leaf, both leaves has the same start ES location and the same end ES location. This makes sense, because nulled symbol instances do not advance the current ES location, but it also implies that the ES locations and LHS symbol cannot be used to uniquely identify a parse node.

23 Events

23.1 Overview

This chapter discusses Libmarpa's events. It contains descriptions of both grammar and recognizer methods.

A method is *event-generating* iff it can add events to the event queue. The event-generating methods are `marpa_g_precompute()`, `marpa_r_earleme_complete()`, and `marpa_r_start_input()`. The event-generating methods always clear all previous events so that, on return from an event-generating method, the only events in the event queue will be the events generated by that method.

A Libmarpa method or macro is *event-safe* iff it does not change the events queue. All Libmarpa accessors are event-safe.

Regardless of the event-safety of the methods calls between event generation and event access, it is good practice to access events as soon as reasonable after the method that generated them. Note 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 are in the event queue, use the `marpa_g_event_count()` method.

To access specific events, use the `marpa_g_event()` and `marpa_g_event_value()` methods.

23.2 Basic event accessors

`Marpa_Event_Type marpa_g_event` (*Marpa_Grammar g*, *Marpa_Event** `event`, *int* `ix`) [Accessor 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 read using the `marpa_g_event_count()` method.

Hard fails if there is no *ix*'th event, or if *ix* is negative. On failure, the locations pointed to by *event* are not changed.

Return value: On success, the type of event *ix*, which is always non-negative. On hard failure, −2.

`int marpa_g_event_count` (*Marpa_Grammar g*) [Accessor function]

Return value: On success, the number of events, which is always non-negative. On hard failure, −2.

`int marpa_g_event_value` (*Marpa_Event** `event`) [Accessor macro]

Returns 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. See Section 23.7 [Event codes], page 78.

Return value: The "value" of the event. Always succeeds.

23.3 Completion events

Libmarpa can be set up to generate a `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 a completion event to be generated, the symbol must be *marked*, and the event must be *activated*.

To mark a symbol as a completion event symbol use the `marpa_g_symbol_is_completion_event_set()` method. The event will be activated by default.

To activate or deactivate a completion symbol event use the `marpa_r_completion_symbol_activate()` method.

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

Allows the user to deactivate and reactivate symbol completion events in the grammar. On success, does the following:

- If *reactivate* is zero, deactivates the event in the grammar.
- If *reactivate* is one, activates the event in the grammar.

The activation status of a completion event in the grammar becomes the initial activation status of a completion event in all of its child recognizers.

This method is seldom needed. When a symbol is marked as a completion event symbol in the grammar, it is activated by default. See [marpa_g_symbol_is_completion_event_set], page 73. And a completion event can be deactivated and reactivated in the recognizer using the `marpa_r_completion_symbol_activate` method. See [marpa_r_completion_symbol_activate], page 72.

Hard fails if the *sym_id* is not marked as a completion event symbol in the grammar, or if the grammar has not been precomputed.

Return value: On success, the value of *reactivate*, which is a boolean. On hard failure, −2.

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

Allows the user to deactivate and reactivate symbol completion events in the recognizer. On success, does the following:

- If *reactivate* is zero, deactivates the event in the recognizer.
- If *reactivate* is one, activates the event in the recognizer.

When a recognizer is created, the activation status of its symbol completion event for *sym_id* is initialized to the activation status of the symbol completion event for *sym_id* in the base grammar.

Hard fails if *sym_id* was not marked for completion events in the base grammar.

Return value: On success, the value of *reactivate*, which is a boolean. On hard failure, −2.

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

On success, returns a boolean which is 1 iff *sym_id* is marked as a completion event symbol in *g*. For more about completion events, see [marpa_g_symbol_is_completion_event_set], page 73.

On soft failure, *sym_id* is well-formed, but there is no such symbol.

Hard fails if *g* is precomputed.

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

`int marpa_g_symbol_is_completion_event_set (` *Marpa_Grammar* `g`*, Marpa_Symbol_ID* `sym_id`*, int* `value`) [Mutator 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 for *sym_id* to occur, *sym_id* must be marked as a completion event symbol, and the completion event for *sym_id* must be activated in the recognizer. Event activation also occurs in the grammar, and the recognizer event activation status for *sym_id* is initialized from the grammar event activation status for *sym_id*. See [marpa_g_completion_symbol_activate], page 72, and see [marpa_r_completion_symbol_activate], page 72.

On success, if *value* is 1,

- marks symbol *sym_id* as a completion event symbol,
- activates the completion event for *sym_id* in *g*, and
- returns 1.

On success, if *value* is 0,

- unmarks symbol *sym_id* as a completion event symbol,
- deactivates the completion event for *sym_id* in *g*, and
- returns 0.

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 that no completion event will ever be generated at earleme 0, the start of parsing.

If *sym_id* is well-formed, but there is no such symbol, soft fails.

Hards fails if the grammar is precomputed.

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

23.4 Symbol nulled events

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 a nulled event to be generated, the symbol must be *marked*, and the event must be *activated*.

To mark a symbol as a nulled event symbol use the `marpa_g_symbol_is_nulled_event_set()` method. The event will be activated by default.

To activate or deactivate a nulled symbol event use the `marpa_r_nulled_symbol_activate()` method.

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

Allows the user to deactivate and reactivate symbol nulled events in the grammar. On success, does the following:

- If *reactivate* is zero, deactivates the event in the grammar.
- If *reactivate* is one, activates the event in the grammar.

The activation status of a nulled event in the grammar becomes the initial activation status of a nulled event in all of its child recognizers.

This method is seldom needed. When a symbol is marked as a nulled event symbol in the grammar, it is activated by default. See [marpa_g_symbol_is_nulled_event_set], page 75. And a nulled event can be deactivated and reactivated in the recognizer using the `marpa_r_nulled_symbol_activate` method. See [marpa_r_nulled_symbol_activate], page 74.

Hard fails if the *sym_id* is not marked as a nulled event symbol in the grammar, or if the grammar has not been precomputed.

Return value: On success, the value of *reactivate*, which is a boolean. On hard failure, −2.

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

Allows the user to deactivate and reactivate symbol nulled events in the recognizer. On success, does the following:

- If *reactivate* is zero, deactivates the event in the recognizer.
- If *reactivate* is one, activates the event in the recognizer.

When a recognizer is created, the activation status of its symbol nulled event for *sym_id* is initialized to the activation status of the symbol nulled event for *sym_id* in the base grammar.

Hard fails if *sym_id* was not marked for nulled events in the base grammar.

Return value: On success, the value of *reactivate*, which is a boolean. On hard failure, −2.

`int marpa_g_symbol_is_nulled_event (` *Marpa_Grammar g*, *Marpa_Symbol_ID* `sym_id`) [Accessor function]

On success, returns a boolean which is 1 iff *sym_id* is marked as a nulled event symbol in *g*. For more about nulled events, see [marpa_g_symbol_is_nulled_event_set], page 75.

On soft failure, *sym_id* is well-formed, but there is no such symbol.

Hard fails if *g* is precomputed.

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

`int marpa_g_symbol_is_nulled_event_set (` *Marpa_Grammar* `g`, *Marpa_Symbol_ID* `sym_id`, *int* `value`) [Mutator 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 for *sym_id* to occur, *sym_id* must be marked as a nulled event symbol, and the nulled event for *sym_id* must be activated in the recognizer. Event activation also occurs in the grammar, and the recognizer event activation status for *sym_id* is initialized from the grammar event activation status for *sym_id*. See [marpa_g_nulled_symbol_activate], page 74, and see [marpa_r_nulled_symbol_activate], page 74.

On success, if *value* is 1,

- marks symbol *sym_id* as a nulled event symbol,
- activates the nulled event for *sym_id* in *g*, and
- returns 1.

On success, if *value* is 0,

- unmarks symbol *sym_id* as a nulled event symbol,
- deactivates the nulled event for *sym_id* in *g*, and
- returns 0.

A symbol instance can never generate both a nulled and a prediction event at the same location. Also, a symbol instance can never generate both a nulled and a completion event at the same location. (As a reminder, a symbol instance is a symbol starting at a specific location in the input, and with a specific length.) This is because the symbol instance for a nulled event must be zero length, and the symbol instance for prediction and completion events can never be zero length.

However, prediction and nulled events for the same symbol can trigger at the same location. This is because The same location can be the location of a nulled instance of a symbol, and the start of an non-nulled instance of the same symbol.

Also, completion and nulled events for the same symbol can trigger at the same location. This is because the same location can be the location of a nulled instance of a symbol, and the end of one or more non-nulled instances of the same symbol.

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 that 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.

If *sym_id* is well-formed, but there is no such symbol, soft fails.

Hards fails if the grammar is precomputed.

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

23.5 Prediction events

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 a prediction event to be generated, the symbol must be *marked*, and the event must be *activated*.

To mark a symbol as a prediction event symbol use the `marpa_g_symbol_is_prediction_event_set()` method. The event will be activated by default.

To activate or deactivate a prediction symbol event use the `marpa_r_prediction_symbol_activate()` method.

`int marpa_g_prediction_symbol_activate (` [Mutator function]
Marpa_Grammar `g`, *Marpa_Symbol_ID* `sym_id`, *int* `reactivate`)

Allows the user to deactivate and reactivate symbol prediction events in the grammar. On success, does the following:

- If *reactivate* is zero, deactivates the event in the grammar.
- If *reactivate* is one, activates the event in the grammar.

The activation status of a prediction event in the grammar becomes the initial activation status of a prediction event in all of its child recognizers.

This method is seldom needed. When a symbol is marked as a prediction event symbol in the grammar, it is activated by default. See [marpa_g_symbol_is_prediction_event_set], page 77. And a prediction event can be deactivated and reactivated in the recognizer using the `marpa_r_prediction_symbol_activate` method. See [marpa_r_prediction_symbol_activate], page 76.

Hard fails if the *sym_id* is not marked as a prediction event symbol in the grammar, or if the grammar has not been precomputed.

Return value: On success, the value of *reactivate*, which is a boolean. On hard failure, −2.

`int marpa_r_prediction_symbol_activate (` [Mutator function]
Marpa_Recognizer `r`, *Marpa_Symbol_ID* `sym_id`, *int* `boolean`)

Allows the user to deactivate and reactivate symbol prediction events in the recognizer. On success, does the following:

- If *reactivate* is zero, deactivates the event in the recognizer.
- If *reactivate* is one, activates the event in the recognizer.

When a recognizer is created, the activation status of its symbol prediction event for *sym_id* is initialized to the activation status of the symbol prediction event for *sym_id* in the base grammar.

Hard fails if *sym_id* was not marked for prediction events in the base grammar.

Return value: On success, the value of *reactivate*, which is a boolean. On hard failure, -2.

`int marpa_g_symbol_is_prediction_event (` [Accessor function]
Marpa_Grammar `g`*, Marpa_Symbol_ID* `sym_id`)

On success, returns a boolean which is 1 iff *sym_id* is marked as a prediction event symbol in *g*. For more about prediction events, see [marpa_g_symbol_is_prediction_event_set], page 77.

On soft failure, *sym_id* is well-formed, but there is no such symbol.

Hard fails if *g* is precomputed.

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

`int marpa_g_symbol_is_prediction_event_set (` [Mutator function]
Marpa_Grammar `g`*, Marpa_Symbol_ID* `sym_id`*, int* `value`)

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 prediction events for *sym_id* to occur, *sym_id* must be marked as a prediction event symbol, and the prediction event for *sym_id* must be activated in the recognizer. Event activation also occurs in the grammar, and the recognizer event activation status for *sym_id* is initialized from the grammar event activation status for *sym_id*. See [marpa_g_prediction_symbol_activate], page 76, and see [marpa_r_prediction_symbol_activate], page 76.

On success, if *value* is 1,

- marks symbol *sym_id* as a prediction event symbol,
- activates the prediction event for *sym_id* in *g*, and
- returns 1.

On success, if *value* is 0,

- unmarks symbol *sym_id* as a prediction event symbol,
- deactivates the prediction event for *sym_id* in *g*, and
- returns 0.

If *sym_id* is well-formed, but there is no such symbol, soft fails.

Hards fails if the grammar is precomputed.

Return value: On success, *value*, which is a boolean. On soft failure, -1. On hard failure, -2.

23.6 Symbol expected events

`int marpa_r_expected_symbol_event_set (` [Mutator function]
Marpa_Recognizer `r`, *Marpa_Symbol_ID* `symbol_id`, *int* `value`)

Libmarpa can be set up to generate an *expected symbol event* (`MARPA_EVENT_SYMBOL_EXPECTED`) when the symbol with ID *symbol_id* is acceptable as a terminal at the current earleme. Note that the symbol expected event is only generated if the symbol with ID *symbol_id* is acceptable as terminal. If the symbol with ID *symbol_id* is expected at the current earleme as a non-terminal, but is not acceptable as a terminal, an expected symbol event will not be triggered at the current earleme.

On success, if *value* is 1,

- activates the symbol expected event for the symbol with ID *symbol_id* in recognizer *r*; and
- returns 1.

On success, if *value* is 0,

- deactivates the symbol expected event for the symbol with ID *symbol_id* in recognizer *r*; and
- returns 0.

Hard fails if *value* is not a boolean. Hard fails if *value* is 1, and *symbol_id* is the ID of a nulling symbol, an inaccessible symbol, or an unproductive symbol. Hard fails if *symbol_id* is not the ID of a valid symbol.

Return value: On success, *value*, which will be a boolean. On hard failure, −2.

23.7 Event codes

`int MARPA_EVENT_NONE` [Accessor macro]

Applications should never see this event. Event value: Unspecified. Suggested message: "No event".

`int MARPA_EVENT_COUNTED_NULLABLE` [Accessor 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` [Accessor 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 53.

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

`int MARPA_EVENT_EXHAUSTED` [Accessor macro]

The parse is exhausted. Event value: Unspecified. Suggested message: "Recognizer is exhausted".

`int MARPA_EVENT_LOOP_RULES` [Accessor 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` [Accessor macro]

This event occurs only if LHS terminals feature is in use. The LHS terminals feature is deprecated. See Section 29.1 [LHS terminals], page 99. Event value: The ID of the symbol. Suggested message: "This symbol is a nulling terminal".

`int MARPA_EVENT_SYMBOL_COMPLETED` [Accessor 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` [Accessor 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 that 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` [Accessor 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` [Accessor 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".

24 Error methods, macros and codes

24.1 Error methods

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

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 current error code. Always succeeds.

`Marpa_Error_Code marpa_g_error_clear` (*Marpa_Grammar* `g`) [Mutator 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.

24.2 Error Macros

`int MARPA_ERRCODE_COUNT` [Accessor 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`.

24.3 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 24.4 [Internal error codes], page 88).

`int MARPA_ERR_NONE` [Accessor macro]

No error condition. The error code is initialized to this value. Methods that 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` [Accessor 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` [Accessor macro]

A tree iterator is positioned before the first tree, and the tree iterator was specified in a context where the tree iterator must be positioned at or after the first tree. 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` [Accessor 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` [Accessor macro]

This error indicates an attempt to add a BNF rule that 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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 45. Numeric value: 17. Suggested message: "Grammar has cycle".

`int` **`MARPA_ERR_HEADERS_DO_NOT_MATCH`** [Accessor macro]

This is an internal error, and indicates that Libmarpa was wrongly built. Libmarpa was compiled with headers that 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`** [Accessor 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`** [Accessor macro]

This error code indicates that the token symbol is an inaccessible symbol — one that 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`** [Accessor 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`** [Accessor 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`** [Accessor 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`** [Accessor macro]

A method was called with an invalid assertion ID. This is a assertion ID that 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`** [Accessor macro]

A method was called with an invalid rule ID. This is a rule ID that 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` [Accessor macro]
A method was called with an invalid symbol ID. This is a symbol ID that 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor macro]
The application attempted to create a bocage from a recognizer with no parse tree. Applications will often want to treat this as a soft error. Numeric value: 41. Suggested message: "No parse".

`int MARPA_ERR_NO_RULES` [Accessor macro]
A grammar that 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` [Accessor macro]
The grammar has no start symbol, and an attempt was made to perform an operation that 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` [Accessor macro]
A method was called with an assertion ID that 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` [Accessor macro]

A method was called with a rule ID that 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` [Accessor macro]

A method was called with a symbol ID that 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` [Accessor 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` [Accessor 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` [Accessor macro]

This error occurs only if LHS terminals feature is in use. The LHS terminals feature is deprecated. See Section 29.1 [LHS terminals], page 99. Numeric value: 49. Suggested message: "A symbol is both terminal and nulling".

`int MARPA_ERR_ORDER_FROZEN` [Accessor 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` [Accessor macro]

The parse is exhausted. Numeric value: 53. Suggested message: "The parse is exhausted".

`int MARPA_ERR_PARSE_TOO_LONG` [Accessor 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` [Accessor macro]
In a method that 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` [Accessor 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` [Accessor macro]
No recognizer progress report is currently active, and an action has been attempted that requires the progress report to be active. 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` [Accessor 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` [Accessor macro]
A symbol or rule rank was specified that 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` [Accessor macro]
A symbol or rule rank was specified that 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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 that 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` [Accessor 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` [Accessor 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` [Accessor macro]
A token length was specified that 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` [Accessor 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` [Accessor macro]
A Libmarpa parse tree iterator is “exhausted”, that is, it has no more parse trees. Numeric value: 79. Suggested message: "Tree iterator is exhausted".

`int MARPA_ERR_TREE_PAUSED` [Accessor macro]
A Libmarpa tree is “paused” and an operation was attempted that 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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` [Accessor 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".

24.4 Internal error codes

An internal error code may be one of two things: First, it can be an error code that 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 that 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` [Accessor macro]

Numeric value: 1.

`int MARPA_ERR_AHFA_IX_OOB` [Accessor macro]
Numeric value: 2.

`int MARPA_ERR_ANDID_NEGATIVE` [Accessor macro]
Numeric value: 3.

`int MARPA_ERR_ANDID_NOT_IN_OR` [Accessor macro]
Numeric value: 4.

`int MARPA_ERR_ANDIX_NEGATIVE` [Accessor macro]
Numeric value: 5.

`int MARPA_ERR_BOCAGE_ITERATION_EXHAUSTED` [Accessor macro]
Numeric value: 7.

`int MARPA_ERR_DEVELOPMENT` [Accessor 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 that describes the error. Numeric value: 9. Suggested message: "Development error, see string".

`int MARPA_ERR_DUPLICATE_AND_NODE` [Accessor macro]
Numeric value: 10.

`int MARPA_ERR_YIM_ID_INVALID` [Accessor macro]
Numeric value: 14.

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

`int MARPA_ERR_INVALID_AHFA_ID` [Accessor 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` [Accessor 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` [Accessor macro]
Numeric value: 23.

`int MARPA_ERR_INVALID_NSYID` [Accessor macro]
Numeric value: 24.

`int MARPA_ERR_NOOKID_NEGATIVE` [Accessor macro]
Numeric value: 33.

`int MARPA_ERR_NOT_TRACING_COMPLETION_LINKS` [Accessor macro]
Numeric value: 35.

`int MARPA_ERR_NOT_TRACING_LEO_LINKS` [Accessor macro]
Numeric value: 36.

`int MARPA_ERR_NOT_TRACING_TOKEN_LINKS` [Accessor macro]
Numeric value: 37.

`int MARPA_ERR_NO_AND_NODES` [Accessor macro]
Numeric value: 38.

`int MARPA_ERR_NO_OR_NODES` [Accessor macro]
Numeric value: 40.

`int MARPA_ERR_NO_TRACE_YS` [Accessor macro]
Numeric value: 46.

`int MARPA_ERR_NO_TRACE_PIM` [Accessor macro]
Numeric value: 47.

`int MARPA_ERR_NO_TRACE_YIM` [Accessor macro]
Numeric value: 45.

`int MARPA_ERR_NO_TRACE_SRCL` [Accessor macro]
Numeric value: 48.

`int MARPA_ERR_ORID_NEGATIVE` [Accessor macro]
Numeric value: 51.

`int MARPA_ERR_OR_ALREADY_ORDERED` [Accessor macro]
Numeric value: 52.

`int MARPA_ERR_PIM_IS_NOT_LIM` [Accessor macro]
Numeric value: 55.

`int MARPA_ERR_SOURCE_TYPE_IS_NONE` [Accessor macro]
Numeric value: 70.

`int MARPA_ERR_SOURCE_TYPE_IS_TOKEN` [Accessor macro]
Numeric value: 71.

`int MARPA_ERR_SOURCE_TYPE_IS_COMPLETION` [Accessor macro]
Numeric value: 68.

`int MARPA_ERR_SOURCE_TYPE_IS_LEO` [Accessor macro]
Numeric value: 69.

`int MARPA_ERR_SOURCE_TYPE_IS_AMBIGUOUS` [Accessor macro]
Numeric value: 67.

`int MARPA_ERR_SOURCE_TYPE_IS_UNKNOWN` [Accessor macro]
Numeric value: 72.

25 Technical notes

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

25.1 Elizabeth Scott's SPPFs

One of our most important data structures is what we call a "bocage". Prof. Scott's work preceded ours, and her SPPF structure is our bocage in all essential respects, so much so that her excellent writeup serves perfectly as documentation for the bocage: Scott, Elizabeth. "SPPF-style parsing from Earley recognisers." Electronic Notes in Theoretical Computer Science 203.2 (2008): 53-67, `https://dinhe.net/~aredridel/.notmine/PDFs/Parsing/SCOTT%2C%20Elizabeth%20-%20SPPF-Style%20Parsing%20From%20Earley%20Recognizers.pdf`.

25.2 Data types used by Libmarpa

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

25.3 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 that creates the bocage also deals with other issues that 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 parse trees 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 implementation, the valuation time object needs only to step through a sequence already determined by the tree iterator.

25.4 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.

25.5 Trap representations

In order to be C89 conformant, an application must initialize all locations that might be read. This is because C89 allows *trap representations*.

A trap representation is a byte pattern in memory that is not a valid value of some object type. When read, the trap representation causes undefined behavior according to the C89 standard, making the application that allowed the read non-conformant to the C89 standard. Trap representations are carefully defined and discussed in the C99 standard.

In real life, trap representations can occur when floating point values are used: Some byte patterns that can occur in memory are not valid floating point values, and can cause undefined behavior when read.

Pointers raise the same issue although, since it can be safely read as an integer, some insist that an invalid pointer is not, strictly speaking, a trap representation. But there is no portable c89-conformant way of testing the integer form of a pointer for validity, so that the only way to guarantee C89 conformance is to initialize the pointer, either to a valid pointer, or to a known and therefore testable value, such as `NULL`.

All this implies that, in order to claim c89-conformance, an application must initialize all locations that might be read to non-trap values. For a stack implementation, this means that, as a practical matter, all locations on the stack must be initialized.

26 Advanced input models

In an earlier chapter, we introduced Libmarpa's concept of input, and described its basic input models. See Chapter 6 [Input], page 15. 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.

26.1 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 6.2.1 [The standard model of input], page 16.

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.

26.2 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 to state 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.

27 Support

The “updates” (`https://github.com/jeffreykegler/libmarpa/blob/updated/UPDATES.md`). document contains instructions for reporting bugs, getting answers to questions, and other support.

28 Futures

This chapter is not about the current interface. Instead, it discusses changes or additions that might be made to this document or to the external interface in the future.

28.1 Nulling versus nulled

Currently we call a zero-length instance (aka tree node) either a nulling instance or a nulled instance. The use of "nulling" is for historic reasons and arguably is confusing. The symbol of a nulling instance is not necessarily a nulling symbol — it might be a nullable symbol. Usage of the term "nulled" is less confusing. At this time, we continue to allow zero-length instances to be called nulling instances because that terminology is embedded in a lot of code and documents.

28.2 Document pre-conditions more formally

A more formal approach to documenting preconditions of the methods is possible, and may be helpful enough to repay any cost in verbosity or complexity. Dave Abrahams recommended I look at `https://www.boost.org/sgi/stl/` for one approach.

28.3 Simpler events interface

Some of the events interfaces are unnecessarily complex. Activation in the grammar is unnecessary, as is the ability to "unmark" an event for a symbol before precomputation. See Section 23.3 [Completion events], page 72, see Section 23.4 [Symbol nulled events], page 74, and see Section 23.5 [Prediction events], page 76.

28.4 Better defined ambiguity metric

With experience, we are now in a position to define an ambiguity metric that can be cheaply calculated, and that might be of real use. Preliminary notes are in the CWeb code.

28.5 Report item traverser should be a time object

Right now, a report item traverser is a kind of "subobject" of a recognizer. It should be made into a full-fledged time object. This will allow multiple report item traversers to be in use at once, allowing more aggressive use of this facility.

28.6 Orthogonal treatment of soft failures

The treatment of soft failure evolved along with this interface, leaving traces of that evolution in the interface. For example, soft failures should not set the error code, but soft failure in `marpa_r_progress_item()` sets the error code to `MARPA_ERR_PROGRESS_REPORT_EXHAUSTED`. See [marpa_r_progress_item], page 56. Similar, soft failure `marpa_t_next()` sets the error code to `MARPA_ERR_TREE_EXHAUSTED`. These non-orthogonalities should be fixed someday.

28.7 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 49, and [marpa_r_start_input], page 47.

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

28.8 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. The only important potential failure is calling `marpa_r_furthest_earleme` when the furthest earleme is an indeterminate value. We eliminate this potential cause of failure by regarding furthest earleme as having 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.

28.9 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 that 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.

28.10 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.

28.10.1 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]

28.10.2 Methods for revising parses

Marpa allows an application to "change its mind" about a parse, rejecting rules 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]

29 Deprecated techniques and methods

29.1 LHS terminals

29.1.1 Overview of LHS terminals

The user creates LHS terminals with the `marpa_g_symbol_is_terminal_set()` method. See [marpa_g_symbol_is_terminal_set], page 99. If the `marpa_g_symbol_is_terminal_set()` method is never called for a grammar, then LHS terminals are not *in use* for any time object with that grammar as its base grammar.

29.1.2 Motivation of LHS terminals

Recall that a terminal symbol is a symbol that may appear in the input. Traditionally, all LHS symbols, as well as the start symbol, must be non-terminals. By default, this is Marpa's behavior.

In a departure from tradition, Marpa had a feature that allowed the user to eliminate the distinction between terminals and non-terminals. This feature is now deprecated.

When LHS terminals are in use, a terminal can appear on the LHS of one or more rules, and can be be the start symbol. Note however, that terminals can never be zero length.

The basis of the LHS terminals feature was that, while sharp division between terminals and non-terminals was a useful simplification for proving theorems, it was not essential in practice. In the UNIX "toolkit" tradition, the practice has been to include even awkward, dangerous tools with no known use, in the toolkit. The philosophy was that empowering the user who discovers new techniques is more important than playing nanny to the toolkit's users.

LHS symbols could be used to bypass, or "short circuit", the rules on whose LHS they occur. Short circuiting rules, it was thought, might prove helpful in debugging, or have other applications.

But, a decade after the release of Libmarpa, no uses for LHS symbols have emerged. And they do introduce many new corner cases into the code and complicate the API documentation.

29.1.3 LHS terminal methods

The *terminal status* of a symbol is a boolean, which is true iff the symbol is a terminal. The terminal status of a symbol is *locked* iff the terminal status of that symbol cannot be changed.

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

On success, does the following:

- Sets the terminal status of the symbol in grammar *g* with symbol ID *sym_id* to *value*. To be used as an input symbol in the `marpa_r_alternative()` method, a symbol must be a terminal.
- Locks the terminal status of symbol *sym_id*.
- Returns *value*.

Hard fails with error code `MARPA_ERR_TERMINAL_IS_LOCKED` if the symbol with *sym_id* is locked, and the terminal status of the symbol with *sym_id* is not equal to *value*. Also hard fails if *value* is not a boolean or if *g* is precomputed.

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

29.1.4 Precomputation and LHS terminals

On success, `marpa_g_precompute()` will sets and locks the terminal status of every symbol. More precisely, let the symbol be *x*, let the terminal status of *x* when `marpa_g_precompute()` was called be *v_before*, and let the terminal status of *x* when `marpa_g_precompute()` returns success be *v_after*. The effect of the successful call of `marpa_g_precompute()` will be as follows:

- If terminal status of *x* is locked when `marpa_g_precompute()` was called, then `v_after = v_before`.
- If terminal status of *x* is not locked, and *x* appears on the LHS of some rule then *v_after* is false.
- If terminal status of *x* is not locked, and *x* does not appear on the LHS of any rule then *v_after* is true.

The terminal status of all symbols is locked after a successful call to `marpa_g_precompute()`. See [marpa_g_precompute], page 45.

29.1.5 Nulling terminals

When LHS terminals are not in use, nulling terminals cannot occur, and applications need not take them in account. This is because, in order to be nullable, a symbol must appear on the LHS of a nullable rule. Without LHS terminals, therefore, no terminals can ever be either nullable or nulling.

Things become more complicated if LHS terminals are allowed. In that case nulling terminals can be created, and Libmarpa must take measures to prevent a recognizer from being created for a grammar with nulling terminals. Libmarpa will not allow a recognizer to be created from a grammar with nulling terminals because they are a logical contradiction. A terminal is (by definition) a symbol which can appear in the input, and a nulling symbol, by definition, cannot appear in the input.

Libmarpa's `marpa_g_precompute` method fails with the error code `MARPA_ERR_NULLING_TERMINAL` if it detects nulling terminals during precomputation. The error code `MARPA_ERR_NULLING_TERMINAL` is library-recoverable. See [marpa_g_precompute], page 45.

Libmarpa's `marpa_g_precompute` method also triggers one `MARPA_EVENT_NULLING_TERMINAL` event for every nulling terminal in the grammar. This implies that one or more `MARPA_EVENT_NULLING_TERMINAL` events occur iff `marpa_g_precompute` fails with error code `MARPA_ERR_NULLING_TERMINAL`.

29.2 Valued and unvalued symbols

29.2.1 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.

Unvalued symbols may be used in combination with another deprecated feature, LHS terminals. See Section 29.1 [LHS terminals], page 99. 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.

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 valued marking. LHS terminals are disabled by default.

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.

29.2.2 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.

29.2.3 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`.

30 History of the Marpa algorithm

This chapter is a quick summary of the most important events in Marpa's development. My "timeline" of the major events in parsing theory has a much broader scope, and also includes more detail about Marpa's development. See Section 31.8 [Timeline], page 105.

- **1970**: Jay Earley invents the algorithm that now bears his name See [Bibliography-Earley-1970], page 104.
- **1991**: Joop Leo describes a way to modify Earley's algorithm so that it runs in O(n) time for all LR-regular grammars. See [Bibliography-Leo-1991], page 105. LR-regular is a vast class of grammars, including all the LR(k) grammars, all grammars parseable with recursive descent, and regular expressions. LR-regular can safely be thought of as including all grammars in practical use today, and then some.
- **2002**: Aycock and Horspool describe a way to do LR(0) precomputation See [Bibliography-Aycock-and-Horspool-2002], page 104. for Earley's algorithm. Their method makes Earley's faster in most practical situations, but not all. In particular, right-recursion remains quadratic in the Aycock and Horspool algorithm. Worst case is no better than Earley's. Leo is unaware of Aycock and Horspool's work and Aycock and Horspool seem unaware of Leo.
- **2010**: Marpa combines the Leo and Aycock-Horspool algorithms in the process making significant changes to both of them. See [Bibliography-Kegler-2022], page 105. The result preserves the best features of both. Marpa also tackles the many remaining implementation issues.

31 Annotated bibliography

31.1 Aho and Ullman 1972

The Theory of Parsing, Translation and Compiling, Volume I: Parsing by Alfred Aho and Jeffrey Ullman (Prentice-Hall: Englewood Cliffs, New Jersey, 1972). I think this was the standard source for Earley's algorithm for decades. It certainly was **my** standard source. The account of Earley's algorithm is on pages 320-330.

31.2 Aycock and Horspool 2002

Marpa is based on ideas from John Aycock and R. Nigel Horspool's "Practical Earley Parsing", *The Computer Journal*, Vol. 45, No. 6, 2002, pp. 620-630. The idea of doing LR(0) precomputation for Earley's general parsing algorithm (see [Bibliography-Earley-1970], page 104), and Marpa's approach to handling nullable symbols and rules, both came from this article.

The Aycock and Horspool paper summarizes Earley's very nicely and is available on the web: `http://www.cs.uvic.ca/~nigelh/Publications/PracticalEarleyParsing.pdf`. Unlike Earley's 1970 paper (see [Bibliography-Earley-1970], page 104), Aycock and Horspool 2002 is **not** easy reading. I have been following this particular topic on and off for years and nonetheless found this paper very heavy going.

31.3 Dominus 2005

Although my approach to parsing is not influenced by Mark Jason Dominus's *Higher Order Perl*, Mark's treatment of parsing is an excellent introduction to parsing, especially in a Perl context. His focus on just about every other technique **except** general BNF parsing is pretty much standard, and will help a beginner understand how unconventional Marpa's approach is.

Both Mark's Perl and his English are examples of good writing, and the book is dense with insights. Mark's discussion on memoization in Chapter 3 is the best I've seen. I wish I'd bought his book earlier in my coding.

Mark's book is available on-line. You can download chapter-by-chapter or the whole thing at once, and you can take your pick of his original sources or PDF, at `http://hop.perl.plover.com/book/`. A PDF of the parsing chapter is at `http://hop.perl.plover.com/book/pdf/08Parsing.pdf`.

31.4 Earley 1970

Of Jay Earley's papers on his general parsing algorithm, the most readily available is "An efficient context-free parsing algorithm", *Communications of the Association for Computing Machinery*, 13:2:94-102, 1970.

Ordinarily, I'd not bother pointing out 35-year old nits in a brilliant and historically important article. But more than a few people treat this article as not just the first word in Earley parsing, but the last as well. Many implementations of Earley's algorithm come, directly and unaltered, from his paper. These implementers and their users need to be aware of two issues.

First, the recognition engine itself, as described, has a serious bug. There's an easy fix, but one that greatly slows down an algorithm whose main problem, in its original form, was speed. This issue is well laid out by Aycock and Horspool in their article. See [Bibliography-Aycock-and-Horspool-2002], page 104.

Second, according to Tomita there is a mistake in the parse tree representation. See page 153 of [Bibliography-Grune-and-Jacobs-1990], page 105, page 210 of [Bibliography-Grune-and-Jacobs-2008], page 105, and the bibliography entry for Earley 1970 in [Bibliography-Grune-and-Jacobs-2008], page 105. In the printed edition of the 2008 bibliography, the entry is on page 578, and on the web (`ftp://ftp.cs.vu.nl/pub/dick/PTAPG_2nd_Edition/CompleteList.pdf`), it's on pp. 583-584. My methods for producing parse results from Earley sets do not come from Earley 1970, so I am taking Tomita's word on this one.

31.5 Grune and Jacobs 1990

Parsing Techniques: A Practical Guide, by Dick Grune and Ceriel Jacobs, (Ellis Horwood Limited: Chichester, West Sussex, England, 1990). This book is available on the Web: `http://dickgrune.com/Books/PTAPG_1st_Edition/`

31.6 Grune and Jacobs 2008

Parsing Techniques: A Practical Guide, by Dick Grune and Ceriel Jacobs, 2nd Edition. (Springer: New York NY, 2008). This is the most authoritative and comprehensive introduction to parsing I know of. In theory it requires no mathematics, only a programming background, but even so it is moderately difficult reading.

This is [Bibliography-Grune-and-Jacobs-1990], page 105, updated. The bibliography for this book is available in enlarged form on the web: `ftp://ftp.cs.vu.nl/pub/dick/PTAPG_2nd_Edition/CompleteList.pdf`.

31.7 Kegler 2022

My writeup of the theory behind Marpa, with proofs of correctness and of my complexity claims, was first made public in 2013. It was updated in 2022, and can be found on `arxiv.org` (`https://arxiv.org/abs/1910.08129`).

31.8 Timeline

Far more popular than my Marpa theory paper is my *Parsing: a timeline*. This is a detailed history of parsing theory, and is available online: `https://jeffreykegler.github.io/personal/timeline_v3`.

31.9 Leo 1991

Marpa's handling of right-recursion uses the ideas in Joop M.I.M. Leo's "A General Context-Free Parsing Algorithm Running in Linear Time on Every LR(k) Grammar Without Using Lookahead", *Theoretical Computer Science*, Vol. 82, No. 1, 1991, pp 165-176. This is a difficult paper. It is available online at `http://www.sciencedirect.com/science/article/pii/030439759190180A`, click the PDF icon at the top left.

31.10 Wikipedia

Wikipedia's article on Backus-Naur form is `http://en.wikipedia.org/wiki/Backus-Naur_form`. It's a great place to start if you don't know the basics of grammars and parsing. As Wikipedia points out, BNF might better be called Panini-Backus Form. The grammarian Panini gave a precise description of Sanskrit more than 23 centuries earlier in India using a similar notation.

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 that are in some way defining.