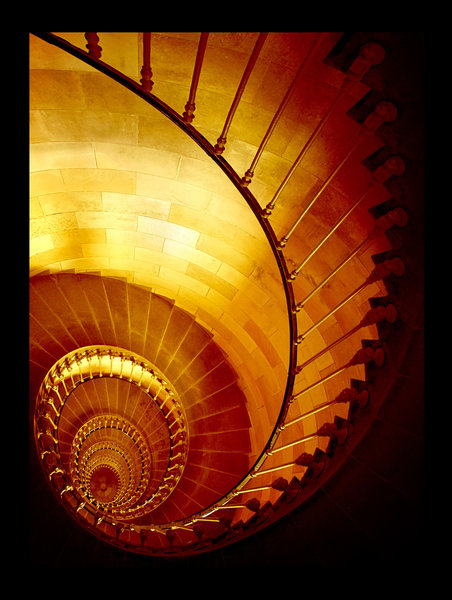
**Recursive Descent Parser**

****

**Module Design**

Author: Raymond de Lacaze

Document Created: 6/20/11

Last Updated: 3/11/21

Contents

[Introduction 4](#_Toc318684047)

[Overview 4](#_Toc318684048)

[Layer 1: A Simple Recursive Descent Parser 5](#_Toc318684049)

[Defining Grammars 5](#_Toc318684050)

[Productions 5](#_Toc318684051)

[Terminal Symbols 5](#_Toc318684052)

[Example 6](#_Toc318684053)

[Generating Tokens 6](#_Toc318684054)

[Example 6](#_Toc318684055)

[The Simple Parser 7](#_Toc318684056)

[Layer 2: The Generic Parser 8](#_Toc318684057)

[Defining Context Grammars 8](#_Toc318684058)

[Example 1 9](#_Toc318684059)

[Example 2 9](#_Toc318684060)

[The Domain Language Analyzer 11](#_Toc318684061)

[Invoking the Analyzer 11](#_Toc318684062)

[Layer 3: Domain Specific Parsers 12](#_Toc318684063)

[English Phrase Parser 12](#_Toc318684064)

[Example 13](#_Toc318684065)

[Measurement Phrase Parser 14](#_Toc318684066)

[Example 14](#_Toc318684067)

[Packaging Phrase Parser 14](#_Toc318684068)

[Example 1. 14](#_Toc318684069)

[Example 2 15](#_Toc318684070)

[Layer 4: An Item Description Parser 16](#_Toc318684071)

[Example 16](#_Toc318684072)

[Future Direction 17](#_Toc318684073)

[Appendix A: Module Organization & Dependencies 18](#_Toc318684074)

[Appendix B: A Micro Grammar 19](#_Toc318684075)

[Appendix C: A Backtracking Example 20](#_Toc318684076)

[Appendix C: Thoughts on the usage of simultaneous multiple Grammars 28](#_Toc318684077)



Recursive Descent Parser

# Introduction

This document describes a new structured and layered API for what was previously referred to as the Item Description Parser. This latter has quite a history, and was originally designed several years ago to parse size and packaging expressions contained inside of healthcare material item descriptions. It was later extended to handle sutures and solutions. During the BSG inception it was again extended to support English noun and prepositional phrases. As BSG evolved and the need for domain knowledge in order to correctly parse item descriptions became crucial, the ITEM DESCRIPTION-Parser was divided into three pieces a Scanner, a language Analyzer and a language Parser. Furthermore the need to parse item descriptions in context became clear, and the Parser was again enhanced to handle noun and lexicon (dictionary) contexts.

This resulted in a reasonably adequate parsing paradigm that included among other features automatic substitution of misspellings and abbreviations, token expansions and phrase transformations, the ability to support multiple grammars and the support of cross-grammar references. However, this ultimately resulted in a somewhat monolithic system that contained an incredible amount of embedded domain knowledge and was hence dependent on and susceptible to structural domain knowledge changes. Furthermore this also resulted in creating intertwined code and module co-dependencies.

The old parser had been completely restructured, rewritten, re-modularized and layered. In addition a PARSER Common Lisp package has been introduced which has helped reveal the dependencies that the old parser had on the Concept Space, the Generic Belief System, the dictionaries and lexicons, the taxonomies and the attribution graph.

It is the belief of the author that a basic parser module and layer should exist that does not depend on anything except perhaps basic system utilities such as various string manipulation functions. It is also the belief of the author that the parser should not procedurally embed domain knowledge and domain. Instead an infrastructure and protocol should be provided by which specialized parsers can be implemented without changing the basic parsing code, but rather by extending the fundamental object classes manipulated by the parser such as the token, grammar & parse trees classes and by specializing the documented generic functions.

This document describes such an effort. This parser redesign and modularization effort consists of four layers shown in the diagram of Appendix A. These are the Simple Parser layer, the Generic Parser Layer, the Domain Parser Layer and the Item Description Parser layer. Each layer has its own module. The API and fundamental concepts underlying each layer are described in detail in the sections that follow.

# Overview

This document describes the Parser API. Unless otherwise noted all documented entities are in the common lisp PARSER package. First the basic parser layer 1 API is described with examples. Then we describe the infrastructure and protocols of layer 2 which allow the users of the parser module to generate specialized domain parsers that leverage domain knowledge without the parser explicitly needing to know anything about the domain. Then three domain specific parsers are described along with their API. Finally we describe an item description parser which leverages the domain specific parsers of layer 3 in order to produce a parser than can operate in a heterogeneous domain environment, essentially parsing descriptions that contain a mixture of languages, specifically English description, measurement descriptions and packaging descriptions.



# Layer 1: A Simple Recursive Descent Parser

Parsers typically take a set of input tokens and a grammar and attempt to find a path through the grammar that accommodates and consumes all the tokens. Below we document a simple macro for defining grammars and a simple function for generating tokens from an input string. Then we document a simple recursive descent parser that takes these tokens and a grammar as input and produces and set of clauses, a set of productions and a parse-tree.

## Defining Grammars

(DEFINE-GRAMMAR <name> <productions><terminal symbols> &key <class>)

This macro returns a grammar object. The keyword argument <class> defaults to the class *grammar* and if specified should be a CLOS class that inherits from the class *grammar*. The arguments <productions> and <terminal-symbols> should be lists that respectively describe the grammar productions for non-terminal-symbols and the grammars terminal symbol predicates for the terminal symbols.

### Productions

These are the non-terminal symbol definitions. Each production consists of a left-hand-side (LHS) naming the non-terminal symbol and a right-hand side (RHS) consisting of a disjunction of conjunctions of terminal symbols and non-terminal symbols. In BNF-like notation a single production would be described as follows:

(<NTS-Name> ([<NTS> | <TS>]\*)\*)

Examples will be given below.

### Terminal Symbols

Each terminal symbol specification consists of a list of the terminal-symbol name and a terminal symbol predicate. The predicate should accept one argument, namely the terminal symbol itself. The terminal symbol object will in fact be an object that inherits from the class *token* as created by the scanner. Terminal symbol predicate functions can access the value of the token by invoking the generic function *object-value.*

### Example

Here is an example of a very simple grammar that only accommodates the input string “a b c”.

**(define-grammar “Micro Grammar 1"**

**'((P (a NTSB c))**

**(NTSB (b)))**

**'((a (lambda (x context)(equalp (object-value x) "a")))**

**(b (lambda (x context)(equalp (object-value x) "b")))**

**(c (lambda (x context)(equalp (object-value x) "c")))))**

* #<GRAMMAR "Micro Grammar 1">

## Generating Tokens

(SCAN-TOKENS <string> <grammar-or-nil>)

This generic function takes a required <string> argument and returns a list of token of objects. The second required argument can be nil or a grammar object as returned by the macro *define-grammar*. The purpose of the effectively optional required <grammar-or-nil> argument will be become clear in subsequent sections.

The objects in the returned list are objects all inherit from the parser class *token*. The actual class of the token object will be one of seven classes depending on the character composition of the token itself.

**Note:** The actual seven classes are *alphabetic-token, numeric-token, special-token, alpha-numeric-token, alpha-special-token, numeric-special-token and alpha-numeric-special-token*. These classes reflect a natural partitioning of tokens based on character composition and are considered domain independent knowledge.

### Example

PARSER(13): **(scan-tokens "a b c" nil)**

🡺(#<Alphabetic Token: a> #<Alphabetic Token: b> #<Alphabetic Token: c>)

(**OBJECT-VALUE** <token>)

This returns the value of the token object specified by <token>. Token objects are created by invoking *scan-token* or *scan-tokens*. Typically the value will be a string for non-numeric type tokens, or a number for numeric type tokens.

## The Simple Parser

The *simple parser* implements a simple recursive descent parser. It returns four values: A list of parsed clauses, a list of unconsumed unknown tokens, list of productions corresponding to the parsed clauses and a parse tree object. It essentially performs a depth-first traversal of the grammar and consumes tokens as the terminal symbol predicate functions succeed on consecutive input tokens. The parser backtracks as terminal symbol predicates fail. A detailed example of this process is provided in Appendices B & C. Appendix B displays the grammar as a fully expanded tree and highlights a failure path and a successful parse path for a simple input string. Appendix C shows a full trace of the recursive descent parser that includes failure points and subsequent backtracking.

(SIMPLE-PARSER <tokens> <grammar>)

This function is used to invoke the simple parser. The required <tokens> argument should be a list of tokens as returned by the function *scan-tokens* described earlier. The required argument <grammar> should be a grammar object as returned by the macro *define-grammar* macro also described earlier.

This function returns as multiple values four distinct values. There are a list of parsed clauses, a list of unconsumed tokens, a list of productions corresponding to the parsed clauses and a parse tree object.

**Note:** For convenience this function can also be called with a to text-string argument in which case the scanner is first invoked to produce a list of token objects.

#### Example

The following example invoke the simple parser with the string “a b c” using the grammar \*micro-grammar-1\*defined in the previous section.

PARSER(12):**(simple-parser "a b c" \*micro-grammar-1\*)**

* ((#<Alphabetic Token: a> #<Alphabetic Token: b> #<Alphabetic Token: c>))
* NIL
* ((A B C))
* #<PARSE-TREE: a b c>



# Layer 2: The Generic Parser

The purpose of the Generic Parser layer is to provide the necessary infrastructure to support the use of *domain* *knowledge* during the parsing process. This layer introduces a *context-grammar* class that provides a place holder in the grammar object for domain knowledge. It then defines a new *apply-terminal-symbol-predicate* method on this class which now invokes the terminal symbol predicates with two arguments, both the terminal-symbol itself and the value of the context slot that is sitting in the grammar. This a simple paradigm by which the parser can be given contextual domain and apply that knowledge without explicitly knowing anything about that knowledge.

This layer also introduces two new API entities. These are *define-context-grammar* and *with-context-grammar*.

## Defining Context Grammars

(DEFINE-CONTEXT-GRAMMAR <name> <productions><terminal symbols> <context> &key <class>)

This macro returns a context grammar object. The keyword argument <class> defaults to the class *context- grammar* and if specified should be a CLOS class that inherits from the class *context-grammar*. The arguments <productions> and <terminal-symbols> should be lists that respectively describe the grammar productions for non-terminal-symbols and the grammars terminal symbol predicates for the terminal symbols. Finally the context object can be any common Lisp entity. It is completely up to the user. Naturally during the parsing process when the terminal symbol predicates are invoked with this additional argument it will up to the supplied procedural domain knowledge contained in the predicate definitions to properly manipulate this context object. The parser need know nothing about it.

(SET- GRAMMAR-CONTEXT <context-grammar> <context>)

This function sets the context slot of the grammar specified by <context-grammar> to be the context specified by <context>. The context can be any Common Lisp entity. It is up to the user of the grammar to ensure that the specified context is compatible with the terminal symbol predicate functions specified by the user when defining the grammar.

(WITH- GRAMMAR-CONTEXT (<context-grammar> <context>) &body <body>)

This macro temporarily sets the context slot of the grammar specified by <context-grammar> to be the context specified by <context> for the duration of the execution of <body>.

### Example 1

PARSER(71):  (WITH-GRAMMAR-CONTEXT (G "English")

                  (SIMPLE-PARSER "Hello Everybody" G))

🡺 ((#<Alphabetic Token: Hello> #<Alphabetic Token: Everybody>))

### Example 2

Here is an example of how to define a simple context grammar.

(defparameter **\*CONTEXT-GRAMMAR-1\***

   (**define-context-grammar** "Hello Grammar"

       ;; These are the productions. Note the recursive <WORD-PHRASE> production.

       `((S (WORD-PHRASE))

         (WORD-PHRASE (SINGLE-WORD WORD-PHRASE) (SINGLE-WORD))

         (SINGLE-WORD (word)))

`     ;; These are the terminal symbol predicates . We only have one type of terminal symbol

;; 'word'. The terminal symbol predicate below does one of two tings depending on the

;; CONTEXT. If the context is English then it sees if <word> is a member of a

;; predefined collection of English words. If the context is French it checks to see

;; if <word> is in a predefined collection of French words,

      `((word (lambda (x context)

            (cond ((pq::same-value-p context "English")

                   (member (object-value x)

                         '("Hello" "Every" "Body" "Everybody")

                         :test #'equalp))

                  ((pq::same-value-p context "French")

                   (member (object-value x)

                         '("Bonjour" "Tout" "Le" "Monde")

                         :test #'equalp))

                  (t nil)))))

      ;; Initial context

      "English"))

Note that clearly any sequence of the four English words or the three French words would be accepted by this silly grammar. The purpose here of course is to illustrate the use of context.

Now we illustrate the use of both the example “Hello Grammar” context grammar defined above and the use of the *with-grammar-context* macro. The core parser functionality and logic is still that of the simple parser layer.

**Example 1: We parse an English sentence in the ENGLISH Context:**

PARSER(71):  (WITH-GRAMMAR-CONTEXT **(\*context-grammar-1\*** "English")

                  (SIMPLE-PARSER "Hello Everybody" \*context-grammar-1\*))

* Clauses:      ((#<Alphabetic Token: Hello> #<Alphabetic Token: Everybody>))
* Unknown:      NIL
* Productions:  ((WORD WORD))

**Example 2: We parse a French sentence in the ENGLSH Context:**

PARSER(72):  (WITH-GRAMMAR-CONTEXT **(\*context-grammar-1\*** "English")

                 (SIMPLE-PARSER "Bonjour tout le monde" \*context-grammar-1\*))

* Clauses:      NIL
* Unknown:     (#<Alphabetic Token: Bonjour> #<Alphabetic Token: tout>

               #<Alphabetic Token: le> #<Alphabetic Token: monde>)

* Productions:  NIL

**Example 3: We parse a FRENCH sentence in the FRENCH Context:**

PARSER(73):  (WITH-GRAMMAR-CONTEXT **(\*context-grammar-1\*** "French")

                  (SIMPLE-PARSER "Bonjour tout le monde" \*context-grammar-1\*))

* Clauses:      ((#<Alphabetic Token: Bonjour> #<Alphabetic Token: tout>

                #<Alphabetic Token: le> #<Alphabetic Token: monde>))

* Unknown:      NIL
* Productions:  ((WORD WORD WORD WORD))

**Example 4**: We parse an ENGLISH sentence in the FRENCH Context:

PARSER(74):  (WITH-GRAMMAR-CONTEXT **(\*context-grammar-1\*** "French")

                  (SIMPLE-PARSER "Hello Everybody" \*context-grammar-1\*))

* Clauses:      NIL
* Unknown:     (#<Alphabetic Token: Hello> #<Alphabetic Token: Everybody>)
* Productions:  NIL

## The Domain Language Analyzer

The Generic Parser layer also provides an infrastructure that facilitates the mapping of parser token objects to domain objects so that the terminal symbol predicate functions can directly manipulate domain objects again without the parser needing to know anything about the actual domain details.

This infrastructure is provided by ensuring that, as its last order of business, the *scan-tokens* method on grammars of class *context-grammar* will invoke a generic function called *analyze-tokens* on the token objects returned by *scan-tokens*. The idea is that domain specific methods on domain specific grammars can provide a method for *analyze-tokens* that will perform the desired mapping of token objects to domain objects and return a list of domain objects. The only functions in the parser that manipulate these objects directly are the terminal symbol predicates and these are actually part of the grammar specification and thus fall into the category of domain knowledge. The parser will then be invoked with these objects and then the terminal symbol predicates supplied with the domain grammar can expect to be invoked with these domain objects as well as the value of the grammar’s context slot. Hence the use of domain objects rather than parser token objects becomes completely transparent to the parser itself.

## Invoking the Analyzer

(ANALYZE-TOKENS <tokens> <context-grammar>)

Normally, this generic function will be invoked automatically with the token objects returned by *scan-tokens* so long as the specified grammar in the invocation of *scan-tokens* (or *parse-tokens*) is a subtype of the class *context-grammar*.

The default method on the class *context-grammar* does nothing and simply returns the token objects created by *scan-tokens* as this layer is not intended to implement any particular domain, but rather provide mechanisms that easily allow the creation of domain specific grammars and parsers.

In the next section on domain specific parsers we will see the application of these mechanisms and protocols..

(ANALYZE-TOKEN <token> <context-grammar>)

Again, this generic function is automatically called by the system on a single token during the token analysis process and therefore users typically need not call this directly. The primary method on the class *token* simply returns the token. The domain specific parsers in the following section provide secondary methods (:around methods) on *alphabetic-token*, *numeric-token* and *special-token* that actually perform the work of mapping the token objects to domain specific objects, which in our case will be an object that inherits from the class *language-word-mixin*. This generic function will prove particularly useful in providing customized behavior to further specialized token classes and mixin classes. It is currently beyond the scope this section to provide examples. Time and space permitting, examples of this will be provided in an advanced programmatic API usage section.



# Layer 3: Domain Specific Parsers

This section describes three domain specific parsers: an *English Phrase parser*, a *Measurement Phrase parser* and a *Packaging Phrase parser*. These are constructed using the mechanisms and protocols described in the previous section. Specialized grammar classes are defined for each specific parser and the appropriate methods which specialize the generic functions of the previous section are added for those classes.

Before describing each parser individually, we describe some common aspects of these parsers. The grammar class *domain-grammar* is defined as a common class that the individual grammar classes of each parser inherit from. A common grammar context protocol is used across all three parsers. The *domain-grammar* class therefore comes with the assumption that the context of the grammar will consist of a principal noun and a source reference object. The source reference object provides access to all the lexicons for a particular domain. The principle noun and the source reference object can be accessed from the domain-grammar object with the accessor functions *context-noun* & *context-reference*.

We proceed by first describing a very high usage of these individual parsers. In each case in is assumes that the domain knowledge comes from the external dictionaries and the domain specific lexicons and taxonomies. In later sections we will describe how these parsers can be used in a more customized fashion.

Note: This layer and module can be loaded with the REPL command *:domain-parser*.

(INITIALIZE-DOMAN-PARSERS)

This function initializes the domain knowledge used by the three domain specific parsers mentioned above. This includes loading the external dictionaries ands well as restoring and initializing the domain Lexicons and Taxonomies.

## English Phrase Parser

\*ENGLISH-GRAMMAR\*

This is the English Phrase grammar object. It is an object of class english-grammar and inherits from domain-grammar and a number of mixin-classes which will be described much later. It is the grammar used by the English phrase parser describe below.

(PARSE-ENGLISH-PHRASE <phrase> &key <reference><noun>)

This is the English Phrase parser. The required <phrase> argument should be a Common Lisp string. The optional keyword argument <reference> should be a source reference object and defaults to the source reference object returned by the function KB-source-reference. See the KBS documentation for more information on this function. Finally the optional keyword argument <noun> defaults to the common lisp keyword *:any*.

This function returns as multiple values four distinct values. There are a list of parsed clauses, a list of unconsumed tokens, a list of productions corresponding to the parsed clauses and a parse tree object.

### Example

The example below illustrates the use the *English Phrase Parser*. Notice the auto-loading and auto-initialization of the dictionaries and the KBS lexicons & taxonomies. Furthermore, because the english-grammar is a *context-grammar* this means that the *analyze-tokens* generic function is invoked automatically and the method provided on the grammar class *domain-grammar* is run which causes the parsers token objects to be mapped to domain objects which in this case are english language words with identified parts of speech. This is completely transparent to the parser and is a result of the infrastructure provided by the *generic parser layer*.

PARSER(3): **(parse-english-phrase "big blue bald bird")**

*Loading basic dictionary with 10k medical terms...*

*Loading WordNet dictionary into memory...*

*Loading KBS DICTIONARY from KB-Dictionary.*

*Loading KBS ABBREVIATIONS from KB-Abbreviations.*

*Loading KBS PLURALS from KB-Plurals.*

*Loading KBS ADJECTIVES from KB-Adjectives.*

*Loading KBS ACRONYMS from KB-Acronyms.*

*Loading KBS SYNONYMS from KB-Synonyms.*

*Loading KBS AKAS from KB-Akas.*

*Loading KBS CODES from KB-Codes.*

*Loading KBS MISSPELLINGS from KB-Misspellings.*

*Loading KBS composites from KB-Composites ...*

*Loading Attribute Taxonomy from: KB-Attribute-Taxonomy...*

*Loading Unit Taxonomy from: KB-Unit-Taxonomy...*

*Loading Value Taxonomy from: KB-Value-Taxonomy...*

*Loading Packaging Taxonomy from: KB-Packaging-Taxonomy...*

*Loading Attribute/Unit bridges from: KB-Attribute-Unit-Bridges...*

*Loading Value/Attribute bridges from: KB-Value-Attribute-Bridges...*

* Clauses:((#<ADJECTIVE: BIG> #<NOUN: BLUE> #<ADJECTIVE: BALD> #<NOUN: BIRD>))
* Unconsumed:NIL
* Productions:((ADJECTIVE NOUN ADJECTIVE NOUN))
* Parse Tree: #<PARSE-TREE: BIG BLUE BALD BIRD>

## Measurement Phrase Parser

(PARSE-MEASUREMENT-PHRASE <measurement-phrase>)

This function parses the specified <measurement-phrase> string. It returns as multiple values four distinct values. These are a list of parsed clauses, a list of unconsumed tokens, a list of productions corresponding to the parsed clauses and a parse tree object.

### Example

PARSER(3): (parse-measurement-phrase "length 3 cm")

* ((#<DIMENSION: LENGTH> #<NUMBER: 3> #<UNIT: CENTIMETER>))
* NIL
* ((SIZE-DIMENSION NUMBER SIZE-UNIT))
* #<PARSE-TREE: #<DIMENSION: LENGTH> #<NUMBER: 3> #<UNIT: CENTIMETER>>

## Packaging Phrase Parser

(PARSE-PACKAGING-PHRASE <packaging-phrase>)

This function parses the specified < packaging-phrase > string. It returns as multiple values four distinct values. These are a list of parsed clauses, a list of unconsumed tokens, a list of productions corresponding to the parsed clauses and a parse tree object.

### Example 1.

PARSER(5): **(parse-packaging-phrase "10 - 12 boxes per cs")**

* ((#<NUMBER: 10> #<CHARACTER-TOKEN: - > #<NUMBER: 12> #<UNIT: BOX>

#<CONNECTOR: PER> #<UNIT: CASE>))

* NIL
* ((NUMBER TO NUMBER PACKAGING-UNIT / PACKAGING-UNIT))

Notice how *boxes* and *cs* were mapped to their singular centric concept equivalents as a result of the automatic application of domain knowledge from the domain lexicons and taxonomies without the Parser needing to know this.

## Example 2

This example demonstrates how the generic function is specialized on PACKAGING-GRAMMAR to supplement the mapping of tokens to language word classes. An :around method is provided rather than a primary method so that any mapping common to the domain-grammar classes is not circumvented but rather leveraged by the call-next-method

(defmethod **ANALYZE-TOKEN** :around ((token ALPHABETIC-TOKEN-MIXIN)

(grammar PACKAGING-GRAMMAR))

(cond ((KBS::find-packaging-unit (object-value TOKEN))

(make-instance 'PACKAGING-UNIT-WORD :token token))

((KBS::find-packaging-connector (object-value TOKEN))

(make-instance 'PACKAGING-CONNECTOR-WORD :token token))

(t

(call-next-method))))

**Note:** This type of mapping to language domain classes facilitates that the job of the grammar’s terminal symbol predicates which can leverage the domain language object class.



# Layer 4: An Item Description Parser

The *Item description Parser* as its name implies is intended to parse healthcare material item descriptions. It builds upon the domain parsers and domain knowledge of the previous section. The current implementation has remained faithful to the old implementation mainly for integration purposes. In other words the entry function parse-item-description takes explicit noun and source-reference arguments. This function has also been extended to better handle multiple grammars, in fact all grammars are tried and the one that consumes the most tokens is applied. This is one simple approach to selecting a grammar when several grammars could apply. Appendix D briefly describes a number of different approaches to this problem.

Unlike the domain specific parser of the previous section, the item description parser does not currently require a grammar a required argument. Instead it takes an optional grammars keyword argument which defaults to a list of the three domain specific grammars previously described.

(**PARSE-ITEM-DESCRIPTION** <description> &key <grammars> <reference><noun>)

**Arguments**

<description> : A common lisp string denoting a healthcare item material description.

<grammars>: A list of context grammars. The context of each will be set to <noun> & <reference>

<reference>: A source reference object, i.e. a collection of domain dictionaries & lexicons

<noun>: A common listp string denoting a registered domain noun category, or :all, or :any.

This function parses the specified <description> string. It returns as multiple values four distinct values. There are a list of parsed clauses, a list of unconsumed tokens, a list of productions corresponding to the parsed clauses and a parse tree object. A source reference object is a collection of lexicons including dictionary, misspellings, acronyms, etc..

### Example

PARSER(27): **(PARSE-ITEM-DESCRIPTION "bone screw 3 mm 12 per box" :reference sr)**

* ((#<NOUN: BONE> #<NOUN: SCREW>)

(#<NUMBER: 3> #<NOUN: MILLIMETER>)

(#<NUMBER: 12> #<PREPOSITION: PER> #<NOUN: BOX>))

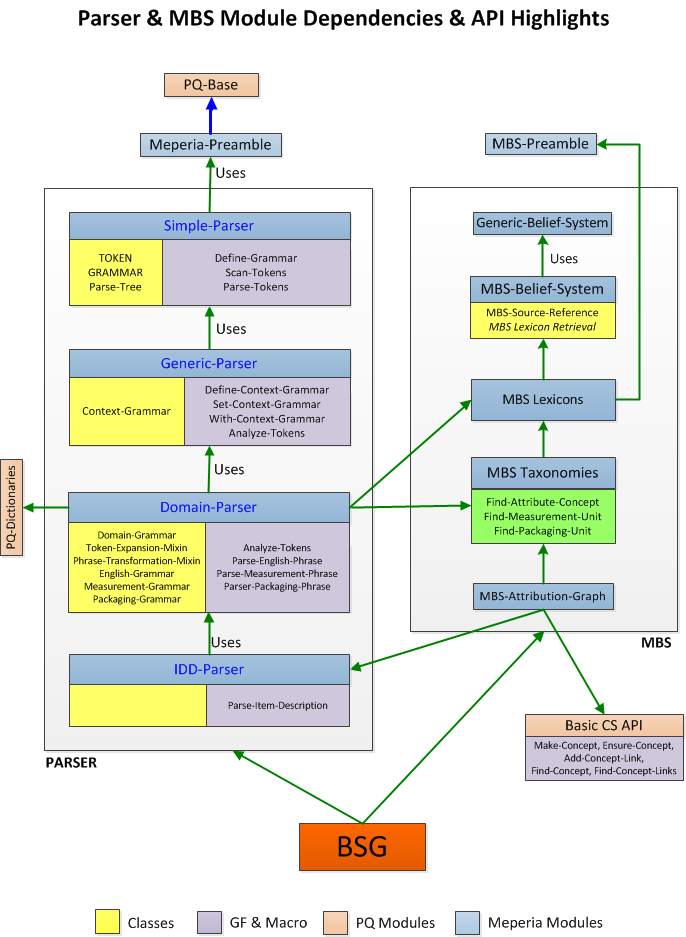
* NIL
* ((NOUN NOUN) (NUMBER SIZE-UNIT) (NUMBER / PACKAGING-UNIT))

Notice that each clause in the returned clauses was parsed by a different grammar.

# Future Direction

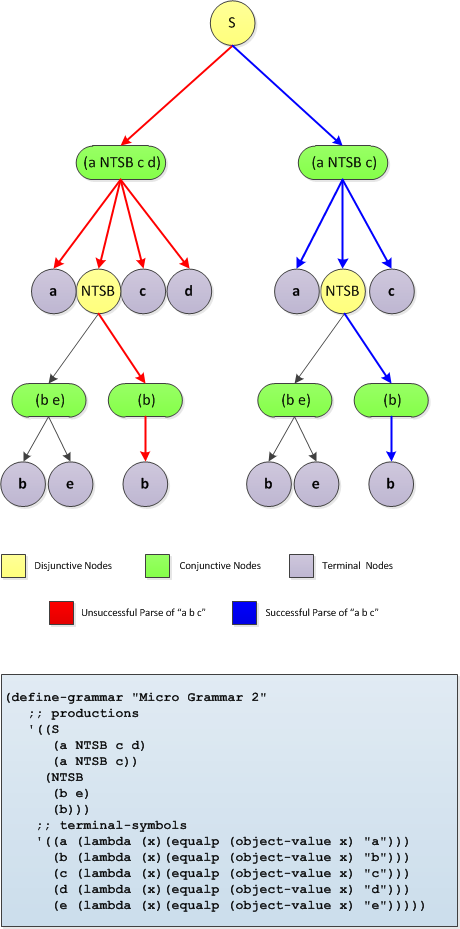


# Appendix A: Module Organization & Dependencies



# Appendix B: A Micro Grammar

The following diagram displays the entire “Micro Grammar 2” as a fully expanded tree, illustrating the disjunctive nodes, the conjunctive nodes and the terminal nodes of the grammar. A successful parse path of the string “a b c” is highlighted in blue. This diagram will prove useful in working through and understanding the detailed backtracking example in Appendix B. In a conjunctive node, all children must be satisfied.



# Appendix C: A Backtracking Example

In the pages that follow we trace through a complete recursive parse of a simple string and a simple grammar. This trace exemplifies the backtracking aspect of the recursive descent parser. The section in red illustrates the first point of failure. The section in blue shows the unwinding of the parse stack and the section green shows the beginning of the actual successful parse path.

PARSER(56): **(SIMPLE-PARSER "a b c" G)**

0[4]: (%SIMPLE-PARSER

(#<Alphabetic Token: a> #<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-TREE: a b c>)

1[4]: (%SIMPLE-PARSER-NTS-OR

(#<Alphabetic Token: a> #<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-TREE: a b c>

:PATTERNS S :RESULT NIL :PRODUCTIONS NIL

:DEPTH 1)

2[4]: (%SIMPLE-PARSER

(#<Alphabetic Token: a> #<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: (A NTSB C D)>

PATTERNS (A NTSB C D) :RESULT NIL :PRODUCTIONS NIL

:DEPTH 1)

3[4]: (%SIMPLE-PARSER-NTS-AND

(#<Alphabetic Token: a> #<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: (A NTSB C D)>

:PATTERNS (A NTSB C D) :RESULT NIL :PRODUCTIONS NIL

:DEPTH 1)

4[4]: (%SIMPLE-PARSER

(#<Alphabetic Token: a> #<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: A>

:PATTERNS A :RESULT NIL :PRODUCTIONS NIL

:DEPTH 2)

5[4]: (%SIMPLE-PARSER-TS

(#<Alphabetic Token: a> #<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: A>

:REFERENCE NIL

:PATTERNS A

:RESULT NIL

:PRODUCTIONS NIL)

5[4]: returned (#<Alphabetic Token: a>)

(#<Alphabetic Token: b> #<Alphabetic Token: c>) (A) T

4[4]: returned (#<Alphabetic Token: a>)

(#<Alphabetic Token: b> #<Alphabetic Token: c>) (A) T

4[4]: (%SIMPLE-PARSER

(#<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: NTSB>

:PATTERNS NTSB :RESULT (#<Alphabetic Token: a>) :PRODUCTIONS (A)

:DEPTH 2)

5[4]: (%SIMPLE-PARSER-NTS-OR

(#<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: NTSB>

:PATTERNS NTSB :RESULT (#<Alphabetic Token: a>) :PRODUCTIONS (A)

:DEPTH 3)

6[4]: (%SIMPLE-PARSER

(#<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: (B E F)>

:PATTERNS (B E F) :RESULT NIL :PRODUCTIONS NIL

:DEPTH 3)

7[4]: (%SIMPLE-PARSER-NTS-AND

(#<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: (B E F)>

:PATTERNS (B E F) :RESULT NIL :PRODUCTIONS NIL

:DEPTH 3)

8[4]: (%SIMPLE-PARSER

(#<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: B>

PATTERNS B :RESULT NIL :PRODUCTIONS NIL

:DEPTH 4)

9[4]: (%SIMPLE-PARSER-TS

(#<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: B>

:PATTERNS B :RESULT NIL :PRODUCTIONS NIL)

9[4]: returned (#<Alphabetic Token: b>)

(#<Alphabetic Token: c>) (B) T

8[4]: returned (#<Alphabetic Token: b>)

(#<Alphabetic Token: c>) (B) T

8[4]: (%SIMPLE-PARSER (#<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: E>

:PATTERNS E :RESULT (#<Alphabetic Token: b>) :PRODUCTIONS (B)

:DEPTH 4)

9[4]: (%SIMPLE-PARSER-TS (#<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: E>

:PATTERNS E :RESULT (#<Alphabetic Token: b>) :PRODUCTIONS (B))

9[4]: returned (#<Alphabetic Token: b>)

(#<Alphabetic Token: c>) (B) NIL

8[4]: returned (#<Alphabetic Token: b>)

(#<Alphabetic Token: c>) (B) NIL

8[4]: (%SIMPLE-PARSER (#<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: F>

:PATTERNS F :RESULT (#<Alphabetic Token: b>) :PRODUCTIONS (B)

:DEPTH 4)

9[4]: (%SIMPLE-PARSER-TS (#<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: F>

:PATTERNS F :RESULT (#<Alphabetic Token: b>) :PRODUCTIONS (B))

9[4]: returned (#<Alphabetic Token: b>)

(#<Alphabetic Token: c>) (B) NIL

8[4]: returned (#<Alphabetic Token: b>)

(#<Alphabetic Token: c>) (B) NIL

7[4]: returned (#<Alphabetic Token: b>)

(#<Alphabetic Token: c>) (B) T

6[4]: returned (#<Alphabetic Token: b>)

(#<Alphabetic Token: c>) (B) T

5[4]: returned (((#<Alphabetic Token: b> #<Alphabetic Token: a>)

(#<Alphabetic Token: c>) (B A) T))

4[4]: returned (#<Alphabetic Token: b> #<Alphabetic Token: a>)

(#<Alphabetic Token: c>) (B A) T

4[4]: (%SIMPLE-PARSER (#<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: C>

:PATTERNS C

:RESULT (#<Alphabetic Token: b> #<Alphabetic Token: a>)

:PRODUCTIONS (B A)

:DEPTH 2)

5[4]: (%SIMPLE-PARSER-TS (#<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: C>

:PATTERNS C

:RESULT (#<Alphabetic Token: b> #<Alphabetic Token: a>)

:PRODUCTIONS (B A))

5[4]: returned (#<Alphabetic Token: c> #<Alphabetic Token: b>

#<Alphabetic Token: a>)

NIL (C B A) T

4[4]: returned (#<Alphabetic Token: c> #<Alphabetic Token: b>

#<Alphabetic Token: a>)

NIL (C B A) T

4[4]: (%SIMPLE-PARSER NIL #<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: D>

:PATTERNS D

:RESULT (#<Alphabetic Token: c> #<Alphabetic Token: b>

#<Alphabetic Token: a>)

:PRODUCTIONS (C B A)

:DEPTH 2)

4[4]: returned NIL NIL NIL NIL

3[4]: returned NIL

(#<Alphabetic Token: a> #<Alphabetic Token: b>

#<Alphabetic Token: c>)

NIL NIL

2[4]: returned NIL

(#<Alphabetic Token: a> #<Alphabetic Token: b>

#<Alphabetic Token: c>)

NIL NIL

2[4]: (%SIMPLE-PARSER

(#<Alphabetic Token: a> #<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: (A NTSB C)>

:PATTERNS (A NTSB C) :RESULT NIL :PRODUCTIONS NIL

:DEPTH 1)

3[4]: (%SIMPLE-PARSER-NTS-AND

(#<Alphabetic Token: a> #<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: (A NTSB C)>

:PATTERNS (A NTSB C) :RESULT NIL :PRODUCTIONS NIL

:DEPTH 1)

4[4]: (%SIMPLE-PARSER

(#<Alphabetic Token: a> #<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: A>

:PATTERNS A :RESULT NIL :PRODUCTIONS NIL

:DEPTH 2)

5[4]: (%SIMPLE-PARSER-TS

(#<Alphabetic Token: a> #<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: A>

:PATTERNS A :RESULT NIL :PRODUCTIONS NIL)

5[4]: returned (#<Alphabetic Token: a>)

(#<Alphabetic Token: b> #<Alphabetic Token: c>) (A) T

4[4]: returned (#<Alphabetic Token: a>)

(#<Alphabetic Token: b> #<Alphabetic Token: c>) (A) T

4[4]: (%SIMPLE-PARSER

(#<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: NTSB>

:PATTERNS NTSB :RESULT (#<Alphabetic Token: a>) :PRODUCTIONS (A)

:DEPTH 2)

5[4]: (%SIMPLE-PARSER-NTS-OR

(#<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: NTSB>

:PATTERNS NTSB :RESULT (#<Alphabetic Token: a>) :PRODUCTIONS (A)

:DEPTH 3)

6[4]: (%SIMPLE-PARSER

(#<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: (B E F)>

:PATTERNS (B E F) :RESULT NIL :PRODUCTIONS NIL

:DEPTH 3)

7[4]: (%SIMPLE-PARSER-NTS-AND

(#<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: (B E F)>

:PATTERNS (B E F) :RESULT NIL :PRODUCTIONS NIL

:DEPTH 3)

8[4]: (%SIMPLE-PARSER

(#<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: B>

:PATTERNS B :RESULT NIL :PRODUCTIONS NIL

:DEPTH 4)

8\* #<STANDARD-METHOD %SIMPLE-PARSER (LIST GRAMMAR)>

9[4]: (%SIMPLE-PARSER-TS

(#<Alphabetic Token: b> #<Alphabetic Token: c>)

#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: B>

:PATTERNS B :RESULT NIL :PRODUCTIONS NIL)

9[4]: returned (#<Alphabetic Token: b>)(#<Alphabetic Token: c>) (B) T

8[4]: returned (#<Alphabetic Token: b>)(#<Alphabetic Token: c>) (B) T

8[4]: (%SIMPLE-PARSER (#<Alphabetic Token: c>)#<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: E>

:PATTERNS E :RESULT (#<Alphabetic Token: b>) :PRODUCTIONS (B)

:DEPTH 4)

9[4]: (%SIMPLE-PARSER-TS (#<Alphabetic Token: c>) #<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: E>

:PATTERNS E :RESULT (#<Alphabetic Token: b>) :PRODUCTIONS (B))

9[4]: returned (#<Alphabetic Token: b>)(#<Alphabetic Token: c>) (B) NIL

8[4]: returned (#<Alphabetic Token: b>)(#<Alphabetic Token: c>) (B) NIL

8[4]: (%SIMPLE-PARSER (#<Alphabetic Token: c>) #<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: F>

:PATTERNS F :RESULT (#<Alphabetic Token: b>) :PRODUCTIONS (B)

:DEPTH 4)

9[4]: (%SIMPLE-PARSER-TS (#<Alphabetic Token: c>) #<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: F>

:PATTERNS F :RESULT (#<Alphabetic Token: b>) :PRODUCTIONS (B))

9[4]: returned (#<Alphabetic Token: b>)(#<Alphabetic Token: c>) (B) NIL

8[4]: returned (#<Alphabetic Token: b>)(#<Alphabetic Token: c>) (B) NIL

7[4]: returned (#<Alphabetic Token: b>)(#<Alphabetic Token: c>) (B) T

6[4]: returned (#<Alphabetic Token: b>)(#<Alphabetic Token: c>) (B) T

5[4]: returned (((#<Alphabetic Token: b> #<Alphabetic Token: a>)(#<Alphabetic Token: c>)

(B A) T))

4[4]: returned (#<Alphabetic Token: b> #<Alphabetic Token: a>)(#<Alphabetic Token: c>)

(B A) T

4[4]: (%SIMPLE-PARSER (#<Alphabetic Token: c>) #<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: C>

:PATTERNS C :RESULT (#<Alphabetic Token: b> #<Alphabetic Token: a>)

:PRODUCTIONS (B A) :DEPTH 2)

5[4]: (%SIMPLE-PARSER-TS (#<Alphabetic Token: c>) #<GRAMMAR "Micro Grammar 2">

:PARSE-NODE #<PARSE-NODE: C>

:PATTERNS C

:RESULT (#<Alphabetic Token: b> #<Alphabetic Token: a>)

:PRODUCTIONS (B A))

5[4]: returned (#<Alphabetic Token: c> #<Alphabetic Token: b> #<Alphabetic Token: a>)

NIL

(C B A) T

4[4]: returned (#<Alphabetic Token: c> #<Alphabetic Token: b> #<Alphabetic Token: a>)

NIL

(C B A) T

3[4]: returned (#<Alphabetic Token: c> #<Alphabetic Token: b> #<Alphabetic Token: a>)

NIL

(C B A) T

2[4]: returned (#<Alphabetic Token: c> #<Alphabetic Token: b> #<Alphabetic Token: a>)

NIL

(C B A) T

1[4]: returned (((#<Alphabetic Token: c> #<Alphabetic Token: b> #<Alphabetic Token: a>)

NIL

(C B A) T))

0[4]: returned (#<Alphabetic Token: c> #<Alphabetic Token: b> #<Alphabetic Token: a>)

NIL (C B A) T

These are the final 4 returned values

**Clauses:** ((#<Alphabetic Token: a> #<Alphabetic Token: b> #<Alphabetic Token: c>))

**Unknown:** NIL

**Productions:** ((A B C))

**Parse Tree:** #<PARSE-TREE: a b c>

# Appendix C: Thoughts on the usage of simultaneous multiple Grammars

Approach 1: Best-first parsing using A\*

Approach 2: Parallel Overlay Parser (cool name to boot!)

Approach 3: Maximum token consumption

