hmrb Documentation

Release 1.2.1

Kristian Boda

Jan 25, 2022

CONTENTS

1 Introduction **3**
- 1.1 Features . . . 3
- 1.2 Rationale . . . 3
- 1.3 Release History . . . 3

2 Quick Start **5**
- 2.1 Installation . . . 5
- 2.2 Input . . . 5
- 2.3 Rules . . . 6
- 2.4 Callbacks . . . 7
- 2.5 A Complete Example . . . 7

3 Writing Rules **9**
- 3.1 Basic rule syntax . . . 9
- 3.2 Union (OR) . . . 11
- 3.3 Optionals and multiples . . . 12
- 3.4 Regular Expression . . . 13
- 3.5 Variables . . . 13
- 3.6 Callbacks and Labels . . . 15

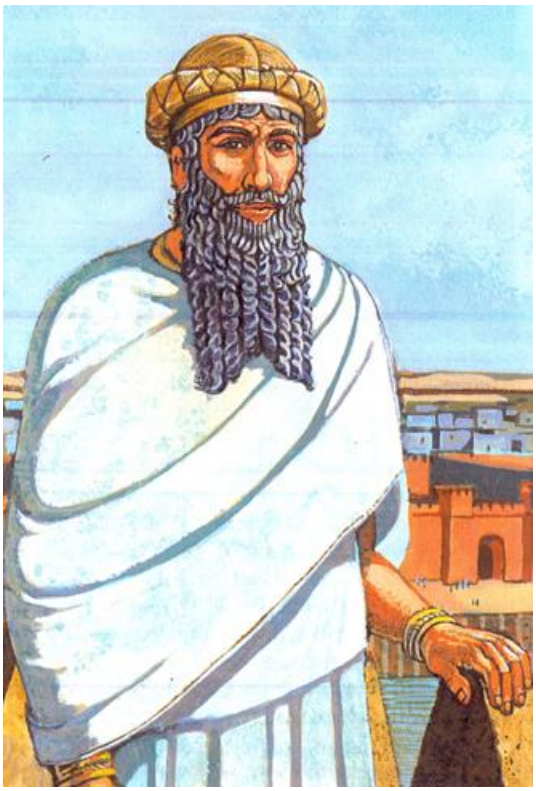
4 spaCy and callbacks **17**
- 4.1 Hammurabi in spaCy 2.X pipelines . . . 17
- 4.2 Hammurabi in spaCy 3.X pipelines . . . 17
- 4.3 Handling Callbacks . . . 18

5 hrmb package **21**
- 5.1 hmrb.core module . . . 21
- 5.2 hmrb.node module . . . 22
- 5.3 hmrb.lang module . . . 25
- 5.4 hmrb.protobuffer module . . . 29

6 Documentation for the Code **33**

Python Module Index **35**

Index **37**



CHAPTER

ONE

INTRODUCTION

Hammurabi[hmrb] is a system designed to efficiently execute rules on sequences of data. Its rule syntax is simple and human-readable but also very expressive.

As input, the system takes a sequence of hash tables (Python `dict`) and is capable of matching any combination of key-value pairs in order. It was designed as a task agnostic framework applicable to a variety of tasks, for instance, intent recognition, text annotation and log monitoring.

1.1 Features

- Attribute level rule definitions using key-values pairs
- Efficient matching of sequence using hash tables with no limit on length
- Support for nested boolean expressions and wildcard operators similar to regular expressions
- Variables can be side-loaded and reused throughout different rule sets
- User-defined rule-level callback functions triggered by a match
- Labels to tag and retrieve matched sequence segments

1.2 Rationale

Rules and heuristics are often used to kick start a project which has insufficient data for a machine learning solution. Hammurabi was built to abstract away these rules and heuristics and make them simple, reliable and explainable. This reduces the effort of building, testing and maintaining early-stage products.

1.3 Release History

Version `v1.2.1 (25.01.2022)`

`v1.2.0 (14.05.2021)`

`v1.1.1 (25.02.2021)`

`v1.1.0 (02.02.2021)`

`v1.0.0 (29.04.2020)`

CHAPTER

TWO

QUICK START

Hammurabi is a generic rule engine library that allows the user to match sequences of objects with an arbitrary set of attributes against a grammar of rules (for more details see *Writing Rules*).

2.1 Installation

To begin, simply install the package from a supported repository (PyPi, Gemfury, Artifactory):

```
$ pip install hmrb
```

2.2 Input

A great way to illustrate the use of Hammurabi is processing annotated text against a rule grammar. Here we will implement a toy relation extraction grammar that will look for people that love gorillas. Annotated text is a sequence of tokens with annotation attributes – this can be the input of the system. For example, we can run a few sentences through spaCy and serialise the output in JSON like this:

```
import json
import spacy

nlp = spacy.load('en_core_web_sm')
sentences = 'I love gorillas. Peter loves gorillas. Jane loves Tarzan.'
input_ = []
for sent in nlp(sentences).sents:
    sent_lst = []
    for token in sent:
        token_dict = {
            'text': token.orth_,
            'lemma': token.lemma_,
            'pos': token.pos_
        }
        sent_lst.append(token_dict)
    input_.append(sent_lst)
with open('my-input.json', 'w') as fh:
    json.dump(input_, fh, indent=2)
```

Content of `my-input.json`:

```
[
  [
    {"text": "I", "lemma": "-PRN-", "pos": "PRON"},
    {"text": "love", "lemma": "love", "pos": "VERB"},
    {"text": "gorillas", "lemma": "gorilla", "pos": "NOUN"},
    {"text": ".", "lemma": ".", "pos": "PUNCT"}
  ],
  [
    {"text": "Peter", "lemma": "Peter", "pos": "PROPN"},
    {"text": "loves", "lemma": "love", "pos": "VERB"},
    {"text": "gorillas", "lemma": "gorilla", "pos": "NOUN"},
    {"text": ".", "lemma": ".", "pos": "PUNCT"}
  ],
  [
    {"text": "Jane", "lemma": "Jane", "pos": "PROPN"},
    {"text": "loves", "lemma": "love", "pos": "VERB"},
    {"text": "Tarzan", "lemma": "Tarzan", "pos": "PROPN"},
    {"text": ".", "lemma": ".", "pos": "PUNCT"}
  ]
]
```

2.3 Rules

In order to capture the right sequence, we need to write a grammar with rules that would detect the sentences containing people that like gorillas. For more details on grammar see *Writing Rules*. Referencing the Babylonian king, rules in Hammurabi are denoted with the keyword **Law**. We wrote below a simple subject-verb-object rule that aims to detect all *people* that love gorillas:

```
Law:
(
  (pos: "PROPN")
  (text: "loves")
  (text: "gorillas")
)
```

It is a very specific rule that will match only one of our input sentences, so we may want to relax it a little bit. We can include pronouns as well as names for the subject and abstract the number of both subject and object by using *lemma* requirements instead of *text*:

```
Law:
- callback: "gorilla people"
(
  ((pos: "PROPN") or (pos: "PRON"))
  (lemma: "love")
  (lemma: "gorilla")
)
```

Now that we've relaxed our rule, we may want to detect other things in our input like say love interests. We can write another rule that identifies a person that loves another person but this time keep it specific:

```
Law:
- callback: "lover"
(
  (pos: "PROPN")
  (text: "loves")
  (pos: "PROPN")
)
```

2.4 Callbacks

Hammurabi supports passing a callback function using the reserved *callback* attribute. The name provided as value is looked up against a dictionary provided to the *callbacks* parameter of the *Core* constructor. The functions associated with matched rules are executed after the matching process is complete. They are passed three positional parameters which then need to handle: the original object sequence *seq*, the slice of *span* matched based on the sequence, and all the associated rule attributes from the grammar as *data*.

All rules (**Laws**) can take an arbitrary number of attributes that will be part of the data structure that is passed along with a matched span. This way the user can identify the rule that was fired and if necessary take action or access some specific data/information through this mechanism.

2.5 A Complete Example

```
import json
from hmrb.core import Core

with open("examples/my-input.json", "r") as fh:
    input_ = json.load(fh)


def conj_be(subj: str) -> str:
    if subj == "I":
        return "am"
    elif subj == "you":
        return "are"
    else:
        return "is"


def gorilla_clb(seq: list, span: slice, data: Dict) -> None:
    subj = seq[span.start]["text"]
    be = conj_be(subj)
    print(f"{subj} {be} a gorilla person.")


def lover_clb(seq: list, span: slice, data: Dict) -> None:
    print(
        f'{seq[span][-1]["text"]} is a love interest of'
        f'{seq[span.start]["text"]}.'
    )
```

(continues on next page)

(continued from previous page)

```
clbs = {"gorilla people": gorilla_clb, "lover": lover_clb}

grammar = """
Law:
- callback: "gorilla people"
(
((pos: "PROPN") or (pos: "PRON"))
(lemma: "love")
(lemma: "gorilla")
)
Law:
- callback: "lover"
(
(pos: "PROPN")
(text: "loves")
(pos: "PROPN")
)
"""

hmb_ext = Core(callbacks=clbs)
hmb_ext.load(grammar)

print("Loaded grammar...")

# process sentences one by one
for i, sent in enumerate(input_, start=1):
    hmb_ext(sent)

# Loaded grammar...
# Processing sent 1
# I am a gorilla person.
# Processing sent 2
# Peter is a gorilla person.
# Processing sent 3
# Tarzan is a love interest of Jane.
```

CHAPTER

THREE

WRITING RULES

Adding rules to Hammurabi is straightforward using a simple human readable syntax capable of defining complex rules.

This section walks you through steps to define rules for Hammurabi. Each subsection introduces a new feature that can be added to better express your rule. Naturally you can combine them as you see fit.

3.1 Basic rule syntax

The following code snippet shows the structural framework of a simple rule. You define a rule as a `Law` within Hammurabi.

```
Law <name>:
    - <return key>: "<return value>"
    - <return key>: "<return value>"
    - callback: "<callback name>"

(
    (attribute: "value")
    ...
    (attribute: "value")
)
```

- Following a `Law` keyword, you can **optionally** define a name for the rule. This allows the immediate re-use of the law in a subsequent rule.
- The initial head part of the `Law` lists key-value pairs that are returned if the rule is matched. The return key can be any string value except reserved keys `callback` and `_`.
- Finally, the body part contains definition of the token sequence which the rule is intended to match.

Attributes matched by the rule engine are defined as **key-value** pairs.

- **Keys** come from your problem setting's vocabulary. For instance, in time-series, this could be any attribute of a time-step, or in Natural Language Processing this could be meta-data on your word tokens.
- **Values** define the actual token needed the pass the rule. Types supported are `string`, `bool`, `int` and `float`. The values most importantly should align with your key vocabulary.

Note: **Escaping characters** is required in string values for special characters i.e. " should be entered as \" and \ as \\.

Listing 1: Example 1 - Single attribute

```
Law
    - package: "found number of icecreams needed"
(
    (written_number: True)  # could match a number like "one"
    (text: "icecream")
)
```

You can also define multiple attribute conditions that must be true for an element (we consider this as an and relationship between attributes). For example, in the below rule both icecream and yell attributes need to match the second sequence element.

Listing 2: Example 2 - Multiple attributes for a token

```
Law
    - package: "found angry person demanding lots of icecream"
(
    (text: "much")
    (text: "icecream", yell: True)
    (today: True)
)
```

Fig. 1: Photo by Alex Jones

3.2 Union (OR)

You can also define rules that require a union logic between tokens. Unions are defined by the `or` keyword. Note that unions must be wrapped in brackets (the indent is optional)

Listing 3: Example 3 - Simple union example

```
Law
    - package: "found person with small icecream appetite"
(
    (
       (text: "small")
       or
       (text: "little")
    )
    (text: "icecream")
)
```

Multiple unions can be nested in a simple structure allowing Hammurabi to define complex rules in a simple manner.

Listing 4: Example 4 - Nested union example

```
Law
    - package: "handling lots of icecream"
(
   (
      (
         (text: "much")
         or
         (text: "little")
      )
   or
      (
         (type: "vanilla")
         or
         (type: "chocolate")
         or
         (type: "strawberry")
      )
  )
  (text: "icecream")
)
```

Note: and syntax: by definition an intersection logic exists between sequential tokens. As mentioned earlier, an and logic exists between attribute key-value pairs.

3.3 Optionals and multiples

To allow compact rules, Hammurabi supports defining optionals and multiples. Each section or element can be marked with the number of times it should be matched. The table below summarises the available logical syntax.

Syntax	Min	Max
`optional`	0	1
`one or more`	1	inf
`zero or more`	0	inf
`X to Y`	X	Y
(default)	1	1

Listing 5: Example 5 - Optionals example

```
Law
    - package: "found person who might be willing to pay for icecream"
(
    optional (text: "free")
    (text: "icecream")
)
```

Listing 6: Example 6 - Complex optionals example

```
Law
    - package: "found person only looking for (very) big icecream"
(
    zero or more (text: "very")
    (text: "big")
    1 to 2 (text: "icecream")
)
```

Naturally, this functionality can be combined with any other syntax on any level.

Listing 7: Example 7 - Nested optionals example

```
Law
    - package: "found person only looking for bright icecream of any flavor"
(
    (text: "bright")
    optional (
         (type: "vanilla")
         or
         (type: "chocolate")
         or
         (type: "strawberry")
    )
    (text: "icecream")
)
```

3.4 Regular Expression

Hammurabi also supports defining attribute values as regular expressions (see Python RE library). The full syntax is as (attribute: regex("<regex expression>")) and can be used on any string value.

Listing 8: Example 8 - Regex example

```
Law
    - package: "found person only looking for some quantity of icecream"
(
    optional (text: "around")
    (text: regex("([0-9])\w+"))
    (text: "icecream")
)
```

Note: Escaping character inside Regex needs to be doubled for special characters i.e. `\.` should be entered as `\\.` and `\\` as `\\\`.

3.5 Variables

Variables allow the reuse of rules, which makes the grammar more readable as well as more efficient. There are two types of variabels supported in Hammurabi: `Var` and `named Law`.

Definitions:

- `Var <name>:` To reuse a sequence of token rules simply define it as a variable. The variable definition uses a similar syntax to defining laws with the addition of naming the variable. This allows us to refer to it in subsequent code. Note that variable definitions are not actually rules. They are elements to be used in Laws and will not be matched on their own. For this same reason, they consist solely of the body (i.e. no head part). To support functionality where you want to not only define a rule but also reuse it in other rules, we added named laws (see below).

- `Law <name>:` To reuse a Law as a variable add a name to its definition. You can refer to it in exactly the same way as a variable *$name*.

References:

- `$<name>` use references to add a sequence defined in a variable to your rule (or to another variable). A reference is defined as the name of a defined variable preceded by `$`. Variable references can be used in conjunction with other features of the language such as optionals and labels.

Listing 9: Example 9 - Variable example

```
Var flavored_icecream:
(
   (
      (type: "vanilla")
      or
      (type: "chocolate")
      or
      (type: "strawberry")
   )
   (text: "icecream")
)

Law
   - package: "found person only looking for some quantity of icecream"
(
   (text: "we")
   (text: "want")
   $flavored_icecream
)
```

When redefining the same as named law (as shown in the below example) you will receive matches for both sections.

Listing 10: Example 10 - Named law example

```
Law flavored_icecream:
(
    (
         (type: "vanilla")
         or
         (type: "chocolate")
         or
         (type: "strawberry")
    )
    (text: "icecream")
)

Law
    - package: "found person only looking for some quantity of icecream"
(
    (text: "we")
    (text: "want")
    $flavored_icecream
)
```

3.6 Callbacks and Labels

Hammurabi also makes it easy to work with the actual matches. We support both retrieval of data through labels and defining a custom action to be executed on match.

Definitions:

- `<label> ->` is the syntax that defines a label. It can be added to any element of the rule. Hammurabi will return the (start, end) offsets of the label within the original sequence in the match object.
- `- callback: "<callback_name>"` is the syntax used to attach a callback to a `Law`, `named Law` or `Var`. The `<callback_name>` string needs to match the key in the (key, function) dictionary that is passed in during the construction of the engine.

Listing 11: Example 11 - Labels and callbacks

```
Law flavored_icecream:
(
    flavour -> (
         (type: "vanilla")
         or
         (type: "chocolate")
         or
         (type: "strawberry")
    )
    (text: "icecream")
)

Law
    - package: "found person only looking for some quantity of icecream"
    - callback: "handle_icecream_van"
(
    (text: "we")
    (text: "want")
    $flavored_icecream
)
```

CHAPTER

FOUR

SPACY AND CALLBACKS

4.1 Hammurabi in spaCy 2.X pipelines

We provide native support for spaCy through the `SpacyCore` object. The `SpacyCore` object can simply be integrated into your existing spaCy 2.X pipelines.

```
from hmrb.core import SpacyCore
core = SpacyCore(callbacks=CALLBACKS,
                 map_doc=convert_to_json_fn,
                 sort_length=True)

core.load(rules)
nlp.add_pipe(core)
```

`SpacyCore` takes a *dict* of callbacks, an optional *function* that converts spaCy doc type (to_json) to a representation that corresponds to your rules and a *bool* whether to sort and execute in ascending order according to match length.

Once the object is instantiated, you can load rules using the `.load` method.

4.2 Hammurabi in spaCy 3.X pipelines

We also provide native support for spaCy 3.0+. You still have to import the *SpacyCore* object to run the component registration and the configuration syntax is slightly different versus 2.0.

We follow the new custom pipeline component API under `spacy.language` [Link]:

First, we have to register both our augmenter functions *map_doc* and any callback functions we would call in spaCy's registry.

Second, we have to create a configuration dictionary that contains the rules and references the callbacks and mapping functions as shown in the example below.

Finally, we can add the "hmrb" pipeline component using our configuration to the spaCy pipeline.

```
from hmrb.core import SpacyCore

@spacy.registry.augmenters("jsonify_span")
def jsonify_span(span):
  return [
     {"lemma": token.lemma_, "pos": token.pos_, "lower": token.lower_}
     for token in span
```

(continues on next page)

(continued from previous page)

```
 ]

@spacy.registry.callbacks("dummy_callback")
def dummy_callback(seq: list, span: slice, data: dict) -> None:
 print("OK")

conf = {
    "rules": GRAMMAR
    "callbacks": {"my_callback": "callbacks.dummy_callback"}
    "map_doc": "augmenters.jsonify_span"
}
nlp.add_pipe(SpacyCore.name, config=conf)
```

4.3 Handling Callbacks

Callbacks allow defining a custom action to be executed upon matching. There are no restrictions on how callbacks can be used, but we provide a few handy patterns below.

4.3.1 Validation

Callbacks can be used to validate likely matches and thereby programmatically extend your rule matching capacity beyond the limits of the grammar.

Listing 1: Example 1 - Validation with Callbacks

```
Var cardinal:
(
    (text: regex("^[1-9]+$"))
)

Var particle:
(
    (text: "st)
    (text: "nd")
    (text: "rd")
    (text: "th")
)

Law I_want_an_Nth_icecream:
- callback: "validate_Nth_icecream"
(
    (text: "I")
    (text: "want")
    (text: regex("an?"))
    cardinal -> $cardinal
    particle -> $particle
    (text: "icecream")
)
```

The above rule would successfully match *I want a 2nd icecream*. It will also incorrectly match *I want a 2th ice cream*

because we didn't spell out all valid English ordinal abbreviations explicitly. Instead of writing an exhaustive list, callbacks can be used to filter out false positives post-match. The following callback definition provides an example of post-match validation:

Listing 2: Example 2 - Callback example

```
ORDINALS = {
    '1': 'st',
    '2': 'nd',
    '3': 'rd'
}

def validate_Nth_icecream(doc, span_range, match_data):
    cardinal_offsets = match_data['_']['labels']['cardinal']
    particle_offsets = match_data['_']['labels']['particle']

    cardinal = doc[*cardinal_offsets].text
    particle = doc[*particle_offsets].text

    if ORDINALS.get(cardinal, 'th') != particle:
        print('No ice cream for you!')
    else:
        print(f'This is your {cardinal}{particle} ice cream!'
```

Note how the labels *cardinal* and *particle* are used to easily identify relevant tokens in the match.

4.3.2 Modularity

When working with large nested rule bases, callbacks can quickly start to become very complex. This can be prevented by applying a modular pattern within your rule base and your callback codebase:

Listing 3: Example 3 - Modularity with Callbacks

```
Var cardinal:
(
    (text: regex("^[1-9]+$"))
)

Var particle:
(
    (text: "st")
    (text: "nd")
    (text: "rd")
    (text: "th")
)

Law abbreviated_ordinal:
- callback: "validate_ordinal"
(
    $cardinal
    $particle
)
```

(continues on next page)

(continued from previous page)

```
Law Do_you_want_the_Nth_or_Nth_icecream:
- callback: "validate_Nth_or_Nth_icecream"
(
    (text: "Do")
    (text: "you")
    (text: "want")
    (text: "the")
    ordinal1 -> $abbreviated_ordinal
    (text: "or")
    ordinal2 -> $abbreviated_ordinal
    (text: "icecream")
)
```

This example shows how you can delegate validation complexity to a sub-rule. The ordinal validation behaviour is logically separated from the sentence validation behaviour. This allows to maintain a more readable grammar and have a cleaner 1-to-1 relationship between logical units, rules and callbacks:

Listing 4: Example 4 - Modularity with Callbacks

```
ORDINALS = {
    '1': 'st',
    '2': 'nd',
    '3': 'rd'
}


def validate_ordinal(doc, span_range, match_data):
    cardinal_offsets = match_data['_']['labels']['cardinal']
    particle_offsets = match_data['_']['labels']['particle']

    cardinal = doc[*cardinal_offsets].text
    particle = doc[*particle_offsets].text

    if ORDINALS.get(cardinal, 'th') == particle:
        doc[cardinal_offsets[0]:particle_offsets[1]]._.ordinal = cardinal + particle


def validate_Nth_or_Nth_icecream(doc, span_range, match_data):
    ordinal1_offsets = match_data['_']['labels']['ordinal1']
    ordinal2_offsets = match_data['_']['labels']['ordinal2']

    ordinal1 = doc[*ordinal1_offsets]._.ordinal
    ordinal2 = doc[*ordinal2_offsets]._.ordinal

    if ordinal1 and ordinal2 and ordinal1 == ordinal2:
        print('You mentioned the same ice cream twice! I want more choice!')
    else:
        print('These are both valid options! How can I choose?!')
```

Note that *validate_ordinal* is only responsible for validating the abbreviated ordinal. If successful, it persists its results in the *doc* object. These will be picked up by *validate_Nth_or_Nth_icecream*, which does not perform any additional validation of the ordinal syntax. Instead, it checks that the two compared ordinals are different. This example shows how frequent callback usage can be used to achieve better segregation of responsibility.

CHAPTER

FIVE

HRMB PACKAGE

5.1 hmrb.core module

class hmrb.core.**Core**(*callbacks: Optional[Dict] = None*, *sets: Optional[Dict] = None*, *sort_length: bool = False*)

Bases: object

Class handling the main functions surrounding the rule engine

Parameters

- **callbacks** (*dict*) – dictionary of callback functions to execute following a successfull call.
- **sort_length** (*bool*) – sort match results according to span length in ascending order (affects callback execution as well.)

Public methods: load : add list of rules to engine __call__ : match list of input dicts with internal rules

execute(*responses: Union[List[Tuple[Tuple[int, int], List[Dict]]], ItemsView[Tuple[int, int], List[Dict]]]*, *input: Any*) → None

_load(*rules: List[List[Dict]]*, *vars: List[List[Dict]]*) → None

Adds list of rules to the engine

Implementation: passes rules to the root BaseNode of the class sequentially

Parameters

- **rules** (*list*) – list of rules to add to root node
- **vars** (*list*) – list of shared varHandle objects to use

_match(*spans: List[Tuple[int, list]]*) → Union[List[Tuple[Tuple[int, int], List[Dict]]], ItemsView[Tuple[int, int], List[Dict]]]

Takes a list of spans and executes matching by passing each to the root node.

Parameters **spans** (*list*) – list of spans to match

Returns list of tuples containing match results

Return type (list)

static default_callback(*input_: list*, *span: slice*, *data: Dict*) → None

load(*inputs: str*) → None

Adds rules to the engine.

Parameters **inputs** (*list*) – list of rules in dialect

class hmrb.core.**SpacyCore**(*callbacks: typing.Optional[typing.Dict] = None*, *sets: typing.Optional[typing.Dict] = None*, *map_doc: typing.Callable = <function _default_map>*, *sort_length: bool = False*)

Bases: *hmrb.core.Core*

Class wrapping the *Core* object into a spaCy component.

Parameters

- **callbacks** (*dict*) – dictionary of callback functions to execute following a successfull call.
- **sort_length** (*bool*) – sort match results according to span length in ascending order (affects callback execution as well.)

Public methods: load : add list of rules in the engine __call__ : match a spaCy Document or Span against the rule set

name = 'hmrb'

hmrb.core.**_default_map**(*doc: Any*) → Any

5.2 hmrb.node module

class hmrb.node.**BaseNode**(*data: Optional[Dict] = None*)

Bases: object

Class for handling nodes

BaseNodes is an atomic element of our data structure. Each token is handled by a separate BaseNode (or one of its subclasses). The BaseNode is designed to build itself in a recursive manner through the consume method from a list of dict rules. It handles the matching of a list of incoming tokens through the BaseNode call method.

Parameters **data** (*dict*) – data associated with the node (optional: None)

Public methods: consume : handles the building of the data structure from a rule __call__ : handles the matching of incoming data

_build_child(*child_key: Tuple[frozenset, int]*, *child:* hmrb.node.BaseNode) → None

Adds new child to the children of BaseNode. Updates call order and attribute index with the new child.

Parameters

- **child_key** (*frozenset*) – a hashable identifier for new child
- **child** (BaseNode) – new child BaseNode object

_consume_child(*next_rule_token: Dict*, *rule: List[Dict]*, *vars: Dict*, *sets: Dict*) → None

_consume_regex(*next_rule_token: Dict*, *rule: List[Dict]*, *vars: Dict*, *sets: Dict*) → None

_consume_set(*next_rule_token: Dict*, *rule: List[Dict]*, *vars: Dict*, *sets: Dict*) → None

_consume_var(*next_rule_token: Dict*, *rule: List[Dict]*, *vars: Dict*, *sets: Dict*) → None

static _make_node_key(*token: Dict*) → Tuple[frozenset, int]

Creates a hashable dictionary key from a dict token

Parameters **token** (*dict*) – a token dictionary

Returns: (frozenset, int) created from the list of (key, value) tuples (sorted by default) and hash of data items

_match(*token: Dict*) → set
(private) Handles the matching of a single token dictionary with the current nodes children.

Implementation: TODO:

Notes: In the case of missing attribute att_name, att_value is None and all children with att_name are removed from the matches

Parameters **token** (*dict*) – dict of (attribute, values) of a single token

consume(*rule: List[Dict]*, *vars: Dict*, *sets: Dict*) → None
Builds internal representation from list of rules

Implementation: Recursively handles the construction of the internal tree structure. Passes token to the appropriate BaseNode class/subclass for handling. If an equivalent node already exists, the remaining tokens of the rules are passed to that node. If no such node exists a new node is added to the children of the current node.

Parameters

- **rule** (*List(dict)*) – list of token rule token dictionaries
- **vars** (*dict*) – dict of all varHandle objects created

static get_att(*token: Any*, *att_name: str*) → Any
Retrieves the value of a token attribute regardless of whether it is a dictionary or a normal object.

Parameters

- **token** (*Any*) – : target token
- **att_name** (*str*) – : attribute name

Returns Value of the target attribute

Return type response (Any)

optimise_call_order() → None

class hmrb.node.**FrozenMap**(**args: Any*, ***kwargs: Any*)
Bases: collections.abc.Mapping

based on https://github.com/pcattori/maps Creates a hashable from any object using frozensets

_abc_impl = <_abc_data object>

classmethod recurse(*obj: Any*) → Any

class hmrb.node.**RegexNode**(*token: Dict*)
Bases: *hmrb.node.BaseNode*

class hmrb.node.**SetNode**(*rule_set: Dict*, *data: Dict*)
Bases: *hmrb.node.BaseNode*

Class for Set nodes

SetNode is a subclass of BaseNode designed to efficiently handling the matching of sets.

Parameters

- **rule_set** (*dict*) – global dictionary of sets to check
- **data** (*dict*) – data object that is returned if the SetNode is matched.

Public methods:

__call__ [handles the matching of incoming list of tokens by] checking if the token is present in the rule_set.

class hmrb.node.**StarNode**(*data: Optional[Dict] = None*)
Bases: *hmrb.node.BaseNode*

hmrb.node.**_recurse**(*obj: Any*, *map_fn: Callable*) → Any
based on https://github.com/pcattori/maps Handles recursion within FrozenMap

hmrb.node.**make_key**(*obj: Any*) → int

Parameters **obj** (*any*) – any type of nested / unnested object

Returns: (int) created from the hash of the FrozenMap object

Notes: Python's hash() is inconsistent across processes/runs.

class hmrb.node.**varNode**(*var_handle:* hmrb.node.BaseNode, *data: Dict*, *min_length: int*, *min_run: int*, *max_run: int*)
Bases: *hmrb.node.BaseNode*

Class for var nodes

varNode is a subclass of BaseNode designed to efficiently handling the reuse of the same node structure (macros). The varNode wraps around a BaseNode object (varHandle) to support shared objects and the logical repitition of executions. The remaining parts are passed to its super BaseNode consume. In this way, we have a clear distinction between repeated/seperated section and sections that follow the repeated parts.

Matching is done in a similar two step process. First, the incoming pattern is passed to the varHandle structure var_handle returning depths of successfull matches. Depending on parameters of the varNode, it tries to match the varHandle multiple times. The remaining unmatched tokens are passed to the children "outer" structure for matching. In case, min_run is 0 it also passes the original input to the "outer" structure.

Parameters

- **var_handle** (BaseNode) – shared BaseNode object that becomes the "inner" structure of the varNode
- **data** (*dict*) – data object that is returned if the varNode is matched.
- **min_length** (*int*) – precomputed minimum length of the inner structure. Used to determine if enough input tokens are left to do another loop.
- **min_run** (*int*) – minimum runs of the inner structure. If set to 0 the inner structure is optional (default 1).
- **max_run** (*int*) – maximum runs of the inner structure (default 1).

Public methods:

__call__ [handles the matching of incoming list of tokens by] first recursing through the shared inner varHandle and then by recursing the remaining unmatched tokens through the super BaseNode object ("outer").

5.3 hmrb.lang module

class hmrb.lang.**Block**(*members: List*, *vars: Dict*, *neg: bool*, *min_: int*, *max_: int*, *label: Optional[str]*, *union: bool = False*, *is_body: bool = False*)

Bases: object

Represents a rule block that may be the body or part of the body of a Law or a Var.

Parameters

- **members** (*members [dict] -- Block*) –
- **negated** (*neg [bool] --*) –
- **matches** (*max [int] -- maximum number of*) –
- **matches** –

_add_var(*children: list*, *length: int*, *min_: int*, *max_: int*) → Any

_parse_block(*block:* hmrb.lang.Block) → None

_parse_labeled_element(*label: str*, *parent: list*) → None

_parse_ref(*ref:* hmrb.lang.Ref) → None

_parse_unit(*unit:* hmrb.lang.Unit) → None

_sequence_extend(*block:* hmrb.lang.Block) → None

_union_extend(*block:* hmrb.lang.Block) → None

parse() → None

class hmrb.lang.**BlockIterator**(*block_str: str*, *inf: int = 10000000000*, *start: int = 1*)

Bases: object

Provides an iterator that iterates over the top-level block segments. These segments could be blocks (see also Block), units (see also Unit), and variable references (see also Ref). These blocks are validated but parsed later on. This iterator will produce tuples of the following shape: (content_string, negated, min_match_num, max_match_num)

Parameters

- **string** (*block_str [str] -- block*) –
- **value** (*inf [int] -- infinity*) –
- **line** (*start [int] -- start*) –

_check_body_level() → None

_close_bracket(*ch: str*) → None

Handles a closing bracket.

Level 0: checks for an open variable reference and closes the block Level 1: checks type of segment (block/unit) and adds to iterable Other levels: adds character to the buffer

Parameters **ch** – – closing bracket character

_consume(*block: str*) → None

Consumes a block string a character at a time. Note that escaped characters are treated differently through the character iterator. The iterator acts on all brackets but it validates only variable references and operators from level 1 (the top content level).

Parameters **string** (*block [str] -- block*) –

_open_bracket(*ch: str*) → None
Handles opening bracket.

Level 0: checks for no operators and opens the block Level 1: parses operator into operator buffer and adds char to buffer Other levels: add to buffer

Parameters **ch** – – open bracket char

_parse_label() → None

_parse_operator() → Tuple
Assumes that there is an operator in the buffer and parses it. Spaces are important for all operators. They are matched as show below. Number placeholders can be replaced with any valid integer.

Examples

- not
- optional
- zero or more
- one or more
- at least {number}
- at most {number}
- {number} to {number}

Raises **ValueError -- when buffer doesn't contain a valid operator and is** – not empty

Returns [Tuple] – negated[bool], min # matches, max # matches

_parse_var() → None
Parses a variable reference name and adds it to the iterable.

property is_union: bool

class hmrb.lang.**Grammar**(*string: str*, *vars_: Dict*)
Bases: object

Represents a Babylonian grammar. It may consist of Var and Law segments.

Parameters **parsed** (*string [str] -- grammar string to be*) –

_build(*string: str*) → None

_deploy() → None

_map_segments(*type_: Any*) → Dict
Collects all segments of particular type and creates a mapping between their names and the objects themselves.

Returns [dict] – mapping between variable names and segments

static _parse_segment_type(*line: str*) → Optional[*hmrb.lang.Types*]
Determines the type of grammar segment: Law or Var.

Parameters **line** – – segment lines

Returns [Types] – segment type

_segment(*string: str*) → Generator
Segments the grammar into laws (Law) and variables (Var). Yields the type of the segment as well as all the lines.

Parameters **string** – – string representation of the segment

static end_var(*parent_end: Any*) → None

parser_map = {<Types.VAR: 'var'>: <class 'hmrb.lang.Var'>, <Types.LAW: 'law'>: <class 'hmrb.lang.Law'>}

class hmrb.lang.**Law**(*lines: List*, *vars: Dict*)
Bases: object

Represents a rule segment of a Babylonian grammar. It consists of an optional name, a list of attributes, and a compulsory body.

Parameters **lines** (*lines [list] -- segment*) –

_parse(*lines: List*) → None

static _parse_atts(*lines: List*) → Dict

static _parse_name(*first_line: str*, *start: int*) → str

static _segment_lines(*lines: List*) → Tuple[List, List]

class hmrb.lang.**Ref**(*ref: str*, *neg: bool*, *min_: int*, *max_: int*, *label: Optional[str]*)
Bases: object

Represents a reference to a variable.

Parameters

- **reference** (*ref [str] -- variable*) –
- **negated** (*neg [bool] --*) –
- **matches** (*max [int] -- maximum number of*) –
- **matches** –
- **label** (*label [str] -- reference*) –

class hmrb.lang.**Types**(*value*)
Bases: enum.Enum

Types of segments and items

BLOCK = 'block'

LAW = 'law'

UNIT = 'unit'

VAR = 'var'

VAR_REF = 'var_ref'

class hmrb.lang.**Unit**(*atts: Dict*, *neg: bool*, *min_: int*, *max_: int*, *label: Optional[str]*)
Bases: object

Represents a group of attribute constraints that form a rule unit. Units are typically members of a block.

Parameters

- **attributes** (*atts [dict] -- unit*) –
- **negated** (*neg [bool] --*) –

- **matches** (*max [int] -- maximum number of*) –
- **matches** –
- **label** (*label [str] -- unit*) –

class hmrb.lang.**Var**(*lines: List*, *vars: Dict*)
Bases: object

Represents a named rule variable segment of a Babylonian grammar.

Parameters **lines** (*lines [list] -- segment*) –

_parse(*lines: List*) → None

static _parse_name(*first_line: str*, *start: int*) → str

hmrb.lang.**char_iter**(*string: str*) → Generator
Iterate over the characters of a string while preserving escaped chars. The point is to allow escaped characters to be treated differently during char iteration (parsing) and then unescaped inside the final data structure.

Parameters **string** (*string [str] -- regex*) –

Returns

[Generator] – generator iterating over the characters of the string

hmrb.lang.**parse_block**(*string: str*, *vars: Dict*, *neg: bool = False*, *min_: int = 1*, *max_: int = 1*, *label: Optional[str] = None*, *start: int = - 1*) → *hmrb.lang.Block*
Parses a block string into a Block object. Takes quantifiers and negation modifier parameters. Recursively calls itself or *parse_unit* to parse nested blocks and units.

Parameters

- **string** (*string [str] -- block*) –
- **negated** (*neg [bool] -- True if*) –
- **matches** (*max [int] -- maximum number of*) –
- **matches** –
- **label** (*label [str] -- block*) –
- **number** (*start [int] -- block start line*) –

Returns [Block] – Block object representing the parsed string

hmrb.lang.**parse_unit**(*string: str*, *neg: bool = False*, *min_: int = 1*, *max_: int = 1*, *label: Optional[str] = None*) → *hmrb.lang.Unit*
Parses a unit string into a Unit object. Unit is a list of key-value pairs inside a pair of brackets. Key-value pairs are separated by a comma. There are colons between keys and values. Values are set inside double quotes, while keys are alphanumeric var-like names. Hyphens and underscores are allowed in the key names, but numbers and hyphens are not allowed in the beginning. An arbitrary amount of space separators is allowed between each of the components of the Unit (key, value, colon and comma).

Examples

- (att_name: “attribute value”, att-name2: “attribute value”)
- (att_name:”attribute value”,att-name2:”attribute value”)

Parameters

- **string** (*string [str] -- unit*) –
- **negated** (*neg [bool] -- True if*) –
- **matches** (*max [int] -- maximum number of*) –
- **matches** –

Returns [Unit] – Unit object representing the parsed string

hmrb.lang.**parse_value**(*string: str*) → Union[str, dict, bool, int, float]
Unescapes a Unit attribute value and determines whether it is a regular string or a Regex.

Parameters **value** (*string [str] -- attribute*) –

Returns [Union[str, Regex, bool, int, float]] – parsed value

hmrb.lang.**unescape**(*string: str*) → str
Unescaping escaped characters typically inside attribute values.

Parameters **unescaped** (*string [str] -- string to be*) –

Returns [str] – unescaped string

hmrb.lang.**unique**(*sequence: list*) → Iterator

5.4 hmrb.protobuffer module

class hmrb.protobuffer.**Labels**(*labels: set*, *depth: int*, *length: int = 1*)
Bases: object

Class wrapper handling Labels message protobuffer

Class for creating, holding and merging Labels type protobuffer messages with other defined types of messages. Initialization of the class creates a new Labels message. Addition of new Labels is handled through the += (__iadd__) magic method.

Protobuffer definition (proto3):

message Labels { map<string, Span> items = 1; }

message Span { string start = 1; string end = 2; }

Parameters

- **labels** (*list*) –
- **depth** (*int*) – of the Match span)
- **length** (*int*) – (defaults to 1)

Public methods: += – handles the addition of a new protobuffer to the object get_depth – returns the maximum depth reached

Notes

span start and end integers are stored as strings, since protobuffers are not able to distinguish between set 0 and unset (default) 0. See Google's protobuffer documentation on default values.

class hmrb.protobuffer.**Match**(*attributes: Dict*, *depth: int*)
Bases: object

Class wrapper handling Match message protobuffers

Class for creating, holding and merging Match type protobuffer messages with other defined types of messages. Initialization of the class creates a new Match message. The Match is considered Active if it contains any valid attributes. Inactive Matches are later ignored in merging objects. An inactive Match with depth_reached transfers its depth_reached to the new object. Addition of data is handled through the += (__iadd__) magic method.

Protobuffer definition (proto3):

message Match { Span span = 1; map<string, string> attributes = 2; map<string, google.protobuf.Any> underscore = 3; }

message Span { string start = 1; string end = 2; }

Parameters

- **attributes** (*dict*) – (except reserved attributes that are added to underscore)
- **depth** (*int*) – of the Match span)

Public methods: += – handles the addition of a new protobuffer to the object set_depth – sets depth reached get_depth – returns the maximum depth reached

Notes

span start and end integers are stored as strings, since protobuffers are not able to distinguish between set 0 and unset (default) 0. See Google's protobuffer documentation on default values.

get_depth() → int

set_depth(*depth: int*) → None

set_start(*start: int*) → None

class hmrb.protobuffer.**Responses**
Bases: object

Class wrapper handling Responses message protobuffers

Class for creating, holding and merging Response type protobuffer messages with other defined types of messages (see response.proto for protocol buffer definitions). Initializing the class creates an empty Responses protobuffer. Addition of data is handled through the += (__iadd__) magic method.

Protobuffer definition (proto3):

message Responses { repeated Match items = 1; }

Public methods: += – handles the addition of a new protobuffer to the object set_start – sets the start of all (not set) span messages get_depth – returns the maximum depth reached

format(*sort_length: bool = False*) → Union[List[Tuple[Tuple[int, int], List[Dict]]], ItemsView[Tuple[int, int], List[Dict]]]

get_depth() → int

set_depth(*depth: int*) → None

set_start(*start: int*) → None

hmrb.protobuffer.**mirror_depth**(*left: Union[*hmrb.protobuffer.Match, hmrb.protobuffer.Responses*]*, *right: Union[*hmrb.protobuffer.Match, hmrb.protobuffer.Responses*]*) → None

hmrb.protobuffer.**mirror_labels**(*left: Any*, *right: Any*) → None

CHAPTER

SIX

DOCUMENTATION FOR THE CODE

- modindex

PYTHON MODULE INDEX

h

hmrb, 33
hmrb.core, 21
hmrb.lang, 25
hmrb.node, 22
hmrb.protobuffer, 29

INDEX

Symbols

_abc_impl (*hmrb.node.FrozenMap attribute*), 23
_add_var() (*hmrb.lang.Block method*), 25
_build() (*hmrb.lang.Grammar method*), 26
_build_child() (*hmrb.node.BaseNode method*), 22
_check_body_level() (*hmrb.lang.BlockIterator method*), 25
_close_bracket() (*hmrb.lang.BlockIterator method*), 25
_consume() (*hmrb.lang.BlockIterator method*), 25
_consume_child() (*hmrb.node.BaseNode method*), 22
_consume_regex() (*hmrb.node.BaseNode method*), 22
_consume_set() (*hmrb.node.BaseNode method*), 22
_consume_var() (*hmrb.node.BaseNode method*), 22
_default_map() (*in module hmrb.core*), 22
_deploy() (*hmrb.lang.Grammar method*), 26
_execute() (*hmrb.core.Core method*), 21
_load() (*hmrb.core.Core method*), 21
_make_node_key() (*hmrb.node.BaseNode static method*), 22
_map_segments() (*hmrb.lang.Grammar method*), 26
_match() (*hmrb.core.Core method*), 21
_match() (*hmrb.node.BaseNode method*), 23
_open_bracket() (*hmrb.lang.BlockIterator method*), 25
_parse() (*hmrb.lang.Law method*), 27
_parse() (*hmrb.lang.Var method*), 28
_parse_atts() (*hmrb.lang.Law static method*), 27
_parse_block() (*hmrb.lang.Block method*), 25
_parse_label() (*hmrb.lang.BlockIterator method*), 26
_parse_labeled_element() (*hmrb.lang.Block method*), 25
_parse_name() (*hmrb.lang.Law static method*), 27
_parse_name() (*hmrb.lang.Var static method*), 28
_parse_operator() (*hmrb.lang.BlockIterator method*), 26
_parse_ref() (*hmrb.lang.Block method*), 25
_parse_segment_type() (*hmrb.lang.Grammar static method*), 26
_parse_unit() (*hmrb.lang.Block method*), 25
_parse_var() (*hmrb.lang.BlockIterator method*), 26
_recurse() (*in module hmrb.node*), 24
_segment() (*hmrb.lang.Grammar method*), 26
_segment_lines() (*hmrb.lang.Law static method*), 27
_sequence_extend() (*hmrb.lang.Block method*), 25
_union_extend() (*hmrb.lang.Block method*), 25

B

BaseNode (*class in hmrb.node*), 22
Block (*class in hmrb.lang*), 25
BLOCK (*hmrb.lang.Types attribute*), 27
BlockIterator (*class in hmrb.lang*), 25

C

char_iter() (*in module hmrb.lang*), 28
consume() (*hmrb.node.BaseNode method*), 23
Core (*class in hmrb.core*), 21

D

default_callback() (*hmrb.core.Core static method*), 21

E

end_var() (*hmrb.lang.Grammar static method*), 27

F

format() (*hmrb.protobuffer.Responses method*), 30
FrozenMap (*class in hmrb.node*), 23

G

get_att() (*hmrb.node.BaseNode static method*), 23
get_depth() (*hmrb.protobuffer.Match method*), 30
get_depth() (*hmrb.protobuffer.Responses method*), 30
Grammar (*class in hmrb.lang*), 26

H

hmrb
    module, 33
hmrb.core
    module, 21
hmrb.lang
    module, 25
hmrb.node
    module, 22

hmrb.protobuffer
    module, 29

I

is_union (*hmrb.lang.BlockIterator property*), 26

L

Labels (*class in hmrb.protobuffer*), 29
Law (*class in hmrb.lang*), 27
LAW (*hmrb.lang.Types attribute*), 27
load() (*hmrb.core.Core method*), 21

M

make_key() (*in module hmrb.node*), 24
Match (*class in hmrb.protobuffer*), 30
mirror_depth() (*in module hmrb.protobuffer*), 31
mirror_labels() (*in module hmrb.protobuffer*), 31
module
    hmrb, 33
    hmrb.core, 21
    hmrb.lang, 25
    hmrb.node, 22
    hmrb.protobuffer, 29

N

name (*hmrb.core.SpacyCore attribute*), 22

O

optimise_call_order() (*hmrb.node.BaseNode method*), 23

P

parse() (*hmrb.lang.Block method*), 25
parse_block() (*in module hmrb.lang*), 28
parse_unit() (*in module hmrb.lang*), 28
parse_value() (*in module hmrb.lang*), 29
parser_map (*hmrb.lang.Grammar attribute*), 27

R

recurse() (*hmrb.node.FrozenMap class method*), 23
Ref (*class in hmrb.lang*), 27
RegexNode (*class in hmrb.node*), 23
Responses (*class in hmrb.protobuffer*), 30

S

set_depth() (*hmrb.protobuffer.Match method*), 30
set_depth() (*hmrb.protobuffer.Responses method*), 30
set_start() (*hmrb.protobuffer.Match method*), 30
set_start() (*hmrb.protobuffer.Responses method*), 31
SetNode (*class in hmrb.node*), 23
SpacyCore (*class in hmrb.core*), 21
StarNode (*class in hmrb.node*), 24

T

Types (*class in hmrb.lang*), 27

U

unescape() (*in module hmrb.lang*), 29
unique() (*in module hmrb.lang*), 29
Unit (*class in hmrb.lang*), 27
UNIT (*hmrb.lang.Types attribute*), 27

V

Var (*class in hmrb.lang*), 28
VAR (*hmrb.lang.Types attribute*), 27
VAR_REF (*hmrb.lang.Types attribute*), 27
varNode (*class in hmrb.node*), 24