jertl

Release 0.1.3

Ray Pelletier

Nov 14, 2022

CONTENTS

Where developers declaratively define and execute common operations on complex data structures.

Operations are specified using a mini-language in which target structures are visually similar to their textual representation.

CHAPTER

ONE

EXAMPLES

1.1 Matching

Jertl can be used to verify the structure of data, select nested values, or both.

```
>>> movie = {"title": "Pinocchio", "MPAA rating": "PG"}
>>>
>>> match = jertl.match('{"title": title, "MPAA rating": "PG"}', movie)
>>>
>>> if match is not None:
...     print(match.bindings['title'])
...
Pinocchio
```

1.2 Filling

Jertl can also be used as a template library.

```
>>> jertl.fill('{"name": name, "age": age, "status": status}',
...            name="Ray",
...            age=66,
...            status='employed')
{'name': 'Ray', 'age': 66, 'status': 'employed'}
```

1.3 Transforming

Data transformations are defined using representations of the source and target data.

```
>>> retire = '{"status": "employed", **the_rest} --> {"status": "retired", **the_rest}'
>>>
>>> ray = {'name': 'Ray', 'age': 66, 'status': 'employed'}
>>>
>>> transformation = jertl.transform(retire, ray)
>>> transformation.filled
{'status': 'retired', 'name': 'Ray', 'age': 66}
```

1.4 Collating

You can use Jertl to verify relationships between data structures.

```
>>> supervises = '''
...     supervisor ~ {"underlings": [*_, name, *_]}
...     employee   ~ {"name": name}
...     '''
>>>
>>> jeremy = {'name': 'Jeremy'}
>>> jeff   = {'name': 'Jeff', 'underlings': ['Jimmy', 'Johnny', 'Jeremy', 'Joe']}
>>>
>>> collation = jertl.collate(supervises, supervisor=jeff, employee=jeremy)
>>> collation is not None
True
```

1.5 Inferring

Combining all these operations gives you an inference engine.

```
>>> rule = '''
...    //
...    // Create a list of movies with their ratings explained
...    //
...    movies       ~ [*_, {"title": title, "MPAA rating": rating},        *_]
...    MPAA_ratings ~ [*_, {"rating": rating, "explanation": explanation}, *_]
... -->
...    movie       := {"title": title, "contents": explanation}
...'''
...
>>> movies = [{'title': 'Toy Story',                          'MPAA rating': 'G'},
...           {'title': 'South Park: Bigger, Longer & Uncut', 'MPAA rating': 'NC-17'}]
...
>>> MPAA_ratings = [{'rating':      'G',
...                  'summary':     'GENERAL AUDIENCES',
...                  'explanation': 'Nothing to offend parents for viewing by children.'}
↪,
...                 {'rating':      'PG',
...                  'summary':     'PARENTAL GUIDANCE SUGGESTED',
...                  'explanation': 'May contain some material parents might not like␣
↪for their young children'},
...                 {'rating':      'PG-13',
...                  'summary':     'PARENTS STRONGLY CAUTIONED',
...                  'explanation': 'Some material may be inappropriate for pre-teens.'},
...                 {'rating':      'R',
...                  'summary':     'RESTRICTED',
...                  'explanation': 'Contains some adult material.'},
...                 {'rating':      'NC-17',
...                  'summary':     'NO ONE 17 AND UNDER ADMITTED',
...                  'explanation': 'Clearly for adults only.'}]
...
```

(continues on next page)

(continued from previous page)

```
>>> for inference in jertl.infer_all(rule, movies=movies, MPAA_ratings=MPAA_ratings):
...     print(inference.fills['movie'])
...
{'title': 'Toy Story', 'contents': 'Nothing to offend parents for viewing by children.'}
{'title': 'South Park: Bigger, Longer & Uncut', 'contents': 'Clearly for adults only.'}
```

CHAPTER

TWO

INSTALLATION

```
pip install jertl
```

2.1 The Mini-language

The mini-language used by jertl contains elements which could be called "JSON adjacent". The syntax for data structures will be recognizable by developers who have worked with JSON. This is more so for Python developers given there are elements of the syntax which are identical to that of the structural matching used in Python's *match* statement.

2.1.1 Literals

The syntax for literals follows that of JSON.

```
>>> jertl.match('true', True) is not None
True
```

```
>>> jertl.match('false', False) is not None
True
```

```
>>> jertl.match('null', None) is not None
True
```

For numbers to be considered a match they *must* be of the same type.

```
>>> jertl.match('4', 4.0) is not None
False
```

whereas

```
>>> jertl.match('4', 4) is not None
True
```

Strings are specified using double quotes

```
>>> jertl.match('"a string"', 'a string') is not None
True
```

Single quoted strings are not allowed.

```
>>> jertl.match("'a string'", 'a string') is not None
Traceback (most recent call last):
...
jertlParseError: line 1: 0 token recognition error at: '''
```

2.1.2 Variables

In a matching context unbound variables match *the current focus*.

```
>>> jertl.match('x', True).bindings
{'x': True}
```

If bound, the varialble binding must match the current focus.

```
>>> jertl.match('[x, x]', [False, False]).bindings
{'x': False}
```

if not, the match fails,

```
>>> jertl.match('[x, x]', [4, 4.0]).bindings
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'bindings'
```

In a fill context unbound variables are not allowed.

```
>>> jertl.fill('[x]')
Traceback (most recent call last):
...
jertlFillException: 'x' not bound
```

Anonymous variables are indicated using an underscore '_'. Like regular variables these match anything but do not result in a binding. When used multiple times in a pattern they do not have to refer to the same value.

```
>>> jertl.match('[_, _, _]', [1, 2, 3]).bindings
{}
```

2.1.3 Structures

Arrays

Arrays are represented by a comma seperated list of expressions surrounded by brackets. Each expression must match item in the corresponding position of the data being matched.

```
>>> jertl.match('[1, 2.0, true, null, "string"]', [1, 2.0, True, None, 'string']) is not␣
↪None
True
```

Splat expressions indicate a variable which is to be bound to a slice of the list being processed. These can only occur within a list expression.

```
>>> jertl.match('[*before, 3, *after]', [1, 2, 3, 4]).bindings
{'before': [1, 2], 'after': [4]}
```

As with non-splatted variables, once it is bound it must match to the same value wherever it occurs in the pattern.

```
>>> jertl.match('[*x, y, *x]', [1,2,3,4,1,2,3]).bindings
{'x': [1, 2, 3], 'y': 4}
```

Splatted variables does not need to be splatted each time.

```
>>> jertl.match('[*x, x]', [1,2,3,[1,2,3]]).bindings
{'x': [1, 2, 3]}
```

Patterns containing splatted variables can result in multiple matches.

```
>>> for match in jertl.match_all('[*before, x, *after]', [1, 2, 3, 4]):
...     print(match.bindings)
...
{'before': [], 'x': 1, 'after': [2, 3, 4]}
{'before': [1], 'x': 2, 'after': [3, 4]}
{'before': [1, 2], 'x': 3, 'after': [4]}
{'before': [1, 2, 3], 'x': 4, 'after': []}
```

Anonymous variables may also be splatted.

```
>>> for match in jertl.match_all('[*_, x, *_]', [1, 2, 3, 4]):
...     print(match.bindings)
...
{'x': 1}
{'x': 2}
{'x': 3}
{'x': 4}
```

Objects

The syntax of objects is a superset of that of JSON. Key/value pairs are seperated by colons. Pairs are surrounded by curly braces "{}". Keys *must* be string literals. Values can be any expression. In addition the last item in an object pattern can be a double splatted variable ("**variable").

```
>>> jertl.match('{"integer": 1, "boolean": true, "anything": anything, "list": [*list]}',
...             {'integer': 1, 'boolean': True, 'anything': {'inner': 'object'}, 'list': [
↪'a', 'list']}).bindings
{'anything': {'inner': 'object'}, 'list': ['a', 'list']}
```

Double splatted variables are bound to the key/value pairs of the focus which were not referenced in the object pattern.

```
>>> jertl.match('{"x": x, "y": y, **double_splat}',
...             {'x': 1, 'y': 2, 'z': 3, 'name': 'Harry'}).bindings
{'x': 1, 'y': 2, 'double_splat': {'z': 3, 'name': 'Harry'}}
```

Once a double splatted is bound it must match the current focus

```
>>> jertl.match('[{"x": x, **double_splat}, {"y": y, **double_splat}]',
...             [{'x': 1, 'z': 3, 'name': 'Harry'}, {'y': 2, 'z': 3, 'name': 'Harry'}]).
↪bindings
{'x': 1, 'double_splat': {'z': 3, 'name': 'Harry'}, 'y': 2}
```

Anonymous variables may be double splatted but this will not do anything useful.

2.1.4 Operations

Simple transforms

The pattern for simple transforms is two structure patterns one each side of the IMPLICATION token

```
<structure> --> <structure>
```

For example

```
'{"name": name, "status": "employed"} --> {"name": name, "status": "retired"}'
```

Targeted matches

Conjoins and rules, which can match to multiple data structures, explicitly identify which structure to examine.

```
<variable> ~ <structure>
```

For example

```
'employee ~ {"name": name, "status": "employed"}'
```

The variable, in this case `employee`, *must* be bound.

Targeted fills

Similarly, rules can perform multiple fill operations. The targeted fill pattern specifies a variable to be bound to a filled structure.

```
<variable> := <structure>
```

For example

```
'retiree :- {"name": name, "status": "retired"}'
```

The variable, in this case `retiree`, *must not* be bound.

Collations

The syntax for collations are a sequence of targeted matches seperated by whitespace

```
supervisor ~ {"underlings": [*_, name, *_]}
employee   ~ {"name": name}
```

Rules

rules are a sequence of targeted matches seperated by whitespace followed by IMPLIES, then a sequence of targetted fills seperated by whitespace.

```
 movies       ~ [*_, {"title": title, "MPAA rating": rating},        *_]
 MPAA_ratings ~ [*_, {"rating": rating, "explanation": explanation}, *_]
-->
 movie       := {"title": title, "contents": explanation}
```

2.1.5 Comments

Everything following a double slash ('//') is ignored.

2.2 API

2.2.1 Toplevel functions

Matching

jertl.**match**(*structure*, *data*)
Matches a structure to data.

Parameters

- **structure** (*str*) – A pattern defining the structure to compare to data.
- **data** (*(Sequence | Mapping | Number)*) – Python data to compare structure against

Returns A description of the match

Return type Optional(*Match*)

jertl.**match_all**(*structure*, *data*)
Generate all Matches of a structure to data.

Parameters

- **structure** (*str*) – A pattern defining the structure to compare to the target data.
- **data** (*(Sequence | Mapping | Number)*) – Python data to match

Yields *Match* – All possible matches

jertl.**compile_match**(*structure*)
Compiles a structure pattern for later use.

Parameters **structure** (*str*) – A pattern defining the structure to compare to the target data.

Returns An object which can be used to match template to objects

Return type *Matcher*

Filling

jertl.**fill**(*structure*, ***bindings*)
Populate a structure.

Parameters

- **structure** (*str*) – A pattern defining the structure to compare to the target data.
- ****bindings** (*Dict[str, list | dict | str | Number]*) – values for free variables referenced by template

Returns A filled template

Return type Dict | Sequence | Number

jertl.**compile_fill**(*structure*)
Compiles a structure for later filling.

Parameters **structure** (*str*) – A pattern defining the structure to compare to the target data.

Returns A filler

Return type *Filler*

Transforming

jertl.**transform**(*transform*, *data*)
Matches a structure to data and fills a target structure.

Parameters

- **transform** (*str*) – A pattern describing structures for matching and filling.
- **data** (*Dict | Sequence | Number*) – Data to match against.

Returns Optional(*Transformer*):: A Transformation

jertl.**transform_all**(*transform*, *data*)
Generate all Transforms of data given Transformation specification.

Parameters

- **transform** (*str*) – A pattern describing structures for matching and filling.
- **data** (*KwArgs*) – Data to match against.

Yields *Transformation* – All possible transforms of the sources

jertl.**compile_transform**(*transform*)
Compiles a transform pattern for later reuse.

Parameters **transform** (*str*) – Pattern describing a transform.

Returns A transformer

Return type *Transformer*

Collating

jertl.**collate**(*collation*, ***bindings*)
Match all matchers to targets

Parameters

- **collation** (*str*) – Specification of collation.
- ****bindings** (*Dict[str, list | dict | str | Number]*) – Data to match against.

Returns A Collation

Return type Optional(Collation)

jertl.**collate_all**(*collation*, ***sources*)
Yields all matches to targets

Parameters

- **collation** (*str*) – Specification of collation.
- ****sources** (*Dict[str, list | dict | str | Number]*) – Data to match against.

Yields *Collation* – All possible collations

jertl.**compile_collate**(*collation*)
Compiles a template for later reuse.

Parameters

- **collation** (*str*) – Specification of collation.
- **sources** (*Dict[str, list | dict | str | Number]*) – Data to match against.

Returns A collator

Return type *Collator*

Inferring

jertl.**infer**(*rule*, ***sources*)
Applies a production rule agains sources and generates new data according to output structures

Parameters

- **rule** (*str*) – Pattern describing a production rule.
- ****sources** (*Dict[str, list | dict | str | Number]*) – Data to match against.

Returns An inference

Return type Optional(*Inference*)

jertl.**infer_all**(*rule*, ***sources*)
Generate all application of rule.

Parameters

- **rule** (*str*) – Pattern describing a production rule.
- ****sources** (*Dict[str, list | dict | str | Number]*) – Data to match against.

Yields *Inference* – All possible inferences

jertl.**compile_rule**(*rule*)
Compiles a inference rule for later reuse.

Parameters **rule** (*str*) – Pattern describing a production rule.

Returns A rule

Return type *Rule*

2.2.2 Processors

class jertl.processors.**Matcher**(*structure*)
Class which compares data to a structure and returns all Matches

__init__(*structure*)

Parameters **structure** (*str*) – pattern describing structure to be identified

match_all(*data*)
match_all Generate all Matches of a structure to data.

Parameters **data** (*(Sequence | Mapping | Number)*) – Python data structure to examine

Yields *Match* – All possible Matches

match(*data*)
match Matches a structure to data.

Parameters **data** (*(Sequence | Mapping | Number)*) – Python data structure to match

Returns (Sequence | Mapping | Number)

Return type Optional(*Match*)

class jertl.processors.**Filler**(*structure*)
Class which performs fills

__init__(*structure*)

Parameters **structure** (*str*) – pattern describing structure to be used for creating new data

fill(***bindings*)
fill Fills structure and returns data with structure variables replaced with their bindings

Parameters ****bindings** (*Dict[str, list | dict | str | Number]*) – Initial variable bindings

Returns data

Return type (list | dict | str | Number)

class jertl.processors.**Transformer**(*transform*)
A transformer with compile structure

__init__(*transform*)

Parameters **transform** (*str*) – pattern describing a transform

transform_all(*data*)
transform_all finds all possible transforms of data

Parameters **data** (*(list | dict | str | Number)*) – Data structure to be transformed

Yields *Transformation* – All possible transforms of the sources

transform(*data*)
transform Finds a transform of data

Parameters **data** (*(list | dict | str | Number)*) – Data structure to be transformed

Returns Optional(*Transformer*):: A Transformation if one was found

class jertl.processors.**Collator**(*collation*)
Class which performs collations

__init__(*collation*)

Parameters **collation** (*str*) – pattern describing a collation

collate_all(***bindings*)
collate_all Yield all possible collations of data

Parameters ****bindings** (*Dict[str, list | dict | str | Number]*) – Initial variable bindings

Yields *Collation* – All possible collations

collate(***bindings*)
collate Find a collation if any

Parameters ****bindings** (*Dict[str, list | dict | str | Number]*) – Initial variable bindings

Returns A Collation

Return type Optional(Collation)

class jertl.processors.**Rule**(*rule*)
Class which finds inferences of a production rule

__init__(*rule*)

Parameters **rule** (*str*) – pattern describing an inference rule

infer_all(***bindings*)
infer_all Yield all possible inferences of rule applied to data

Parameters ****bindings** (*Dict[str, list | dict | str | Number]*) – Initial variable bindings

Yields *Inference* – All possible inferences

infer(***bindings*)
infer Finds an inference of rule applied to data (if any)

Parameters ****bindings** (*Dict[str, list | dict | str | Number]*) – Initial variable bindings

Returns An inference

Return type Optional(*Inference*)

2.2.3 Result Objects

class jertl.results.**Bindings**(*bindings*)
Base class for jertl processing results

property **bindings**
bindings A dictionary mapping variable identifiers to their bindings

Returns

dictionary containing a set of variable bindings

Return type Dict[str, list | dict | str | Number]

class jertl.results.**Match**(*matched*, *bindings*)
Output of a matcher

property **matched**
matched The result of filling the structure pattern with the variables bound during the match process

Returns Result of a fill using the structure pattern and variable bindings

Return type list | dict | str | Number

class jertl.results.**Transformation**(*matched*, *filled*, *bindings*)
Class describing a tranform result

property **matched**
matched matching structure with variables replace with their bindings

Returns Result of a fill using the matching structure pattern and variable bindings

Return type list | dict | str | Number

property **filled**
filled transformed structure

Returns Result of a fill using the target structure pattern and variable bindings

Return type list | dict | str | Number

class jertl.results.**Collation**(*matchers*, *bindings*)
Output of a collator

property **matches**
matches The result of filling the targetted match structures with the variables bound during the match process

Returns Mapping of target identifiers to filled matching structures

Return type Dict[str, list | dict | str | Number]

class jertl.results.**Inference**(*matchers*, *fillers*, *bindings*)
Result yielded by an Inferencer

property **matches**
result The result of filling the targetted match structures with the variables bound during the match process

Returns Mapping of variable identifiers to structures

Return type Dict[str, list | dict | str | Number]

property **fills**
result The result of filling the targetting fill structures with the variables bound during the match process

Returns Mapping of variable identifiers to structures

Return type Dict[str, list | dict | str | Number]

2.3 Future Directions

2.3.1 New Features

Evaluation

In a matching context the result of an evaluation expression is interpreted as a guard whose truthyness determines whether or not the matching process should continue.

```
company ~ {"employees": [*_, {"age": <<@ > 65>>}, *_]}
```

The '@' character in the example above refers to the current focus (the value of "age").

In a filling context the result of the evaluation expression is the value inserted into the structure

```
//
// Do a bit of math
//

{"numerator": x, "denominator": y} --> {"quotient": <<x/y>>}
```

```
>> jertl.transform('{"numerator": x, "denominator": y} --> {"quotient": <<x/y>>}', x=1.0,
↪ y=2.0).result
{'quotient': 0.5}
```

Inline Filters

```
//
// Find employees eligible for graduation
//

company ~ {"employees": [*_, employee<<{"age": <<@ > 65>>}>>, *_]}
```

The @ character in the example above refers to the current focus.

Aggregation

```
//
// Get list of employee names
//

{"employees": employees} --> <<map({"name": name} --> name, employees)>>
```

This will appear in the same release having Evaluation. The functions *map*, *filter*, and *reduce*, will be in the set of predefined functions.

Iteration

The jertl mini-language is 'functional-first' so iteration will not be available.

Disjunction

Eventually.

2.3.2 Ecosystem Improvements

- Semantic analysis to identify potential issues beforehand.
- Informative Exceptions (location in pattern where exception happened).
- Editor support for .jertl source files.
- Structure matching optimization.
- "Eat our own dogfood" in AST generator and OpCode emitter.
- Match debugger

2.3.3 Taking it to the Next Level

Compilation

```
>>> cat find_highest_paid_male_employee.jertl

module highest_paid_male
//
// Create a Python module with functions is_male, salary, and highest_paid_male_employee
//
matcher is_male:
    {"gender": "male"}

transform salary:
    {"salary": salary} --> salary

collate highest_paid_male_employee [company]:       \\ Input to this function is a list␣
↪containing a `company` data structure
    company         ~  {"employees": employees}
    highest_salary :=  <<max(map(age, filter(is_male, employees)))>>
    employees       ~  [*_, employee<<{"salary": highest_salary}>>, *_]

>>> jertl find_highest_paid_male_employee.jertl -o generated_sources
```

Rule Sets and Chaining (Forward Inference)

Where we consider multiple inference rules simultaneously and can require there to be sequences of inferences in order for the rule to apply.

```
//
// Your classic ancestors problem
//
ruleset ancestors
    rule find_ancestors [person]:
        person ~ {"parents": [mother, father]}
      -->
        0=0=0 ancestors [person, mother]    // `0=0=0`: chain to ancestors ruleset
        0=0=0 ancestors [person, father]

    rule note_ancestry_and_look_deeper [person, ancestor]:
        person ~ {"name": person_name},
        ancestor ~ {"name": ancestor_name, "parents": [ancestors_mother, ancestors_
↪father]}
      -->
        ancestry := {"person": person_name, "ancestor": ancestor_name}
        0=0=0 ancestors [person, ancestors_mother]
        0=0=0 ancestors [person, ancestors_father]

    rule no_more_birth_records [person, parent]:
        person ~ {"name": person_name},
        ancestor ~ {"name": ancestor_name, "parents": null}
      -->
        ancestry := {"person": person_name, "ancestor": ancestor_name}
```

Working Memory

Where working memory is a key/value store.

```
rule supervises [supervisor, employee]
    supervisor ~ {"name": supervisor_name, "underlings": [\*_, employee, \*_]}
    employee@  ~ {"name": underling_name}      // <-- `employee` is bound to string␣
↪which points to data in working memory.
                                               //     The data is retrieved and the␣
↪matching process continued.
    -->
    supervises := [supervisor_name, underling_name]
```

2.3.4 Moonshots

Data Stores

Where data is external to Python.

```
rule is_supervisor [supervisor, employee]
    supervisor@sql_employee_table ~ {"name": supervisor_name, "underlings": [\*_,␣
↪employee, \*_]}
    employee@sql_employee_table   ~ {"name": underling_name}      // employee is key to␣
↪data stored in a SQL table
    -->
    supervises := [supervisor_name, underling_name]
```

Mutation

Where we mutate a data structure using overlays. What is an overlay you ask? Good question! Overlays and how they work need to be formally defined.

```
rule record_change_of_supervisor [employee_id, previous_supervisor, new_supervisor]
  -->
    previous_supervisor :- {"underlings": [*_, employee_id, *_]}  // <-- remove portion␣
↪of data structure matching overlay
    new_supervisor      :+ {"underlings": [*_, employee_id]}      // <-- add data␣
↪described by overlay
```

2.4 Under the Hood

2.4.1 Glossary

pattern A string describing elements of the jertl mini-language for processing data structures. Elements include structures, transforms, collations, and rules.

structures Used both for matching and as template for filling.

binding A mapping from identifiers to data.

variable A reference to data. Variables are represented in mini-language via identifiers.

identifier string used to represent a variable in the mini-language.

rule sequence of conditions which when satisfied implies a set of actions to be taken.

inference describes conditions and results of the application of a rule to data.

vararg Mini-language construct indicating that a variable should be bound to a slice of an array.

kwargs Construct indicating that a variable could be bound to an object containing key value pairs not referenced in the enclosing object structure.

matcher Construct defining a condition which is satisfied when a variable matches a structure

setter Construct defining an action which results in a variable being bound to a filled template

focus The data being matched. (see *The jertl Virtual Machine*)

2.4.2 Mini-language Grammar

Derived from json.g4 in the repository antlr / grammars-g4

```
grammar Jertl;

options {
   language = Python3;
}

toplevel_structure
   : structure EOF
   ;

transform
   : structure IMPLIES structure EOF
   ;

collation
   : matcher* EOF
   ;

rule_
   : matcher* IMPLIES setter* EOF
   ;

matcher
   : variable MATCHES structure
   ;

setter
   : variable ASSIGNED structure
   ;

IMPLIES
   : '-->'
   ;

// Structures

structure
   : obj
   | array
   | atom
   | variable
   ;

obj
   : '{' key_values+=key_value (',' key_values+=key_value)* (',' kwargs)?'}'
   | '{' '}'
   ;

key_value
```

(continues on next page)

(continued from previous page)

```
   : STRING ':' structure
   ;

kwargs
   : '**' variable
   ;

array
   : '[' elements+=element (',' elements+=element)* ']'
   | '[' ']'
   ;

element
   : structure
   | varargs
   ;

varargs
   : '*' variable
   ;

atom
   : NULL
   | TRUE
   | FALSE
   | INTEGER
   | FLOAT
   | STRING
   ;

variable
   : IDENTIFIER
   ;

NULL
   : 'null'
   ;
TRUE
   : 'true'
   ;

FALSE
   : 'false'
   ;

STRING
   : '"' (ESC | SAFECODEPOINT)* '"'
   ;
MATCHES
   : '~'
   ;
```

(continues on next page)

(continued from previous page)

```
ASSIGNED
   : ':='
   ;

IDENTIFIER
   : VALID_ID_START VALID_ID_CHAR*
   ;

fragment VALID_ID_START
   : ('a' .. 'z') | ('A' .. 'Z') | '_'
   ;

fragment VALID_ID_CHAR
   : VALID_ID_START | ('0' .. '9')
   ;

fragment ESC
   : '\\' (["\\/bfnrt] | UNICODE)
   ;

fragment UNICODE
   : 'u' HEX HEX HEX HEX
   ;

fragment HEX
   : [0-9a-fA-F]
   ;

fragment SAFECODEPOINT
   : ~ ["\\\u0000-\u001F]
   ;

INTEGER
   : '-'? INT
   ;

FLOAT
   : '-'? INT ('.' [0-9] +)? EXP?
   ;

fragment INT
   : '0' | [1-9] [0-9]*
   ;

// no leading zeros

fragment EXP
   : [Ee] [+\-]? INT
   ;

// \- since - means "range" inside [...]
```

(continues on next page)

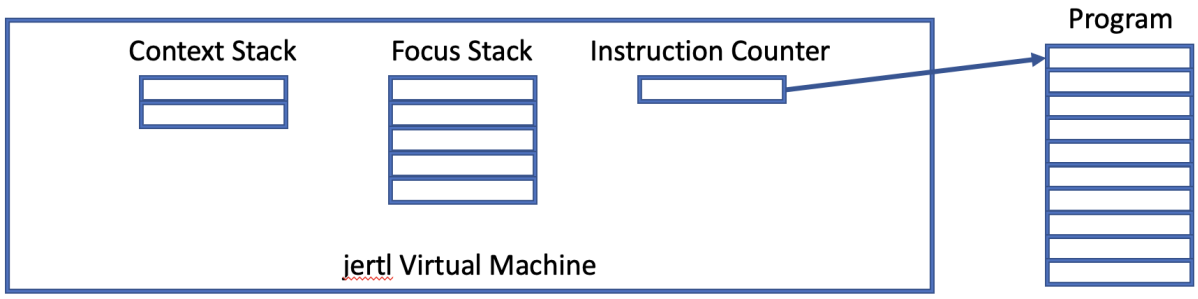
(continued from previous page)

```
WS
   : [ \t\n\r] + -> skip
   ;

LINE_COMMENT
    : '//' ~[\r\n]* -> skip
    ;
```

2.4.3 The jertl Virtual Machine

Architecture



Context Stack

When binding the variable of a splat expression the vm progressively includes larger slices of the list being examined. This is implemented using backtracking. When first matching an unbound binding splat expression to a list, the vm takes a snapshot of the vm's state and pushes it onto the context stack. The variable of the splat expression is then bound to an empty list.

On backtrack the vm uses the snapshot at the top of the context stack to restore state, widens the splat variable's binding to include the next element of the list, then continues execution. If there are no more elements in the list the snapshot is popped off the context stack and the vm again backtracks. The vm halts if the context stack is empty.

Focus Stack

The focus stack holds the structured data being processed. For the match and transform operations the data argument is the first item pushed onto the stack. This stack grows as the vm examines deeper items of a data structure or switches to different data via a targetted match.

Masking

Masks are thin wrappers used by the VM to mark off items in lists and dicts which have been matched. The intent is to avoid destructive modification of input data, excessive copying, and to make the code a bit more readable.

Opcodes

OpCode	Argument	Action
MATCH_VALUE	A value to compre to focus	Compares value to focus, backtrack if no match.
BIND_VARIABLE	Identifier	Bind identifier to focus.
MATCH_VARIABLE	Identifier	Compares binding of identifier to focus, backtrack if no match.
BIND_VARARGS	Identifier	Push snapshot of virtual machine and onto context stack. Sets initial binding of identifier to an empty list.
MATCH_VARARGS	Identifier	Check if focus starts with binding of identifier, backtrack if no match.
MASK_IF_LIST	<none>	If focus is a list mask it, otherwise backtrack.
MASK_IF_DICT	<none>	If focus is a dict mask it, otherwise backtrack.
FOCUS_ON_HEAD	<none>	Push first item of list onto focus stack, backtrack if list is empty.
FOCUS_ON_KEY	Key	Push value of dict key onto focus stack, backtrack if key not present.
FOCUS_ON_BINDING	Identifier	Push variable binding onto focus stack.
POP_FOCUS	<none>	Pop focus stack.
YIELD_BINDINGS	<none>	Yield copy of current bindings then backtrack.

CHAPTER

THREE

CREDITS

3.1 Credits

This module uses Open Source components. You can find the source code of their projects along with license information below. We acknowledge and are grateful to these developers for their contributions.

3.1.1 Antlr

Home Page: antlr.org
License: The BSD License

3.1.2 JSON Grammar

Used as base of jertl mini-language

File: json.g4
Repository: antlr / grammars-g4

3.1.3 dataclasses

Backport of dataclasses required to use jertl in Python 3.6

Home Page: github
License: Apache Software License

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

j

INDEX

Symbols

B

C

F

I

J

M

R

T