**get_table_name**

get_table_name

**parse_ranges** .......................... 5

**parse_recursion** .......................... 6

%where% .................................. 6

---

Index 8

---

eval_recursion  \( \textit{Evaluate an entire dynprog recursion.} \)

**Description**

This function takes the ranges and recursions of a specification and evaluate the dynprog expression, returning a filled out dynamic programming table.

**Usage**

eval_recursion(ranges, recursions)

**Arguments**

ranges  \( \text{The ranges specification} \)

recursions  \( \text{The recursions specification} \)

**Value**

The filled out dynamic programming table

---

get_table_name  \( \textit{Extract the table name from a pattern.} \)

**Description**

We generally assume that patterns are on the form \( \text{table}[\text{exprs}] \) where \text{table} is the name of the dynamic programming table. This function extract that name.

**Usage**

get_table_name(patterns)

**Arguments**

patterns  \( \text{The patterns used in the recursion.} \)

**Value**

The table part of the pattern.
**make_condition_checks**  
*Translate condition expressions into calls that test them.*

**Description**
Takes the full dynprog expression and construct a list of condition tests for each component of the recursion.

**Usage**
```
make_condition_checks(ranges, patterns, conditions, recursions)
```

**Arguments**
- `ranges`  
The ranges specifications
- `patterns`  
The patterns specifications
- `conditions`  
The conditions specifications
- `recursions`  
The recursions specification

**Value**
A list of calls, one per recursion, for testing conditions.

**make_pattern_match**  
*Translate a pattern into a predicate that checks the pattern.*

**Description**
Takes a pattern from the DSL and make a comparison of the pattern specification against range variables.

**Usage**
```
make_pattern_match(pattern, range_vars)
```

**Arguments**
- `pattern`  
An expression on the form `table[index-list]`
- `range_vars`  
A list of the variables used in the ranges.

**Value**
An expression that tests `pattern` against `range_vars`. 
**make_pattern_tests**  
*Make pattern tests for all patterns.*

**Description**

This function calls `make_pattern_match()` for each pattern in `patterns` and return a list of all the pattern test expressions.

**Usage**

```python
make_pattern_tests(patterns, range_vars)
```

**Arguments**

- **patterns**  
  A list of the patterns used in a recursion.
- **range_vars**  
  The variables used in the ranges.

**Value**

A list of pattern check expressions.

---

**make_recursion_case**  
*Construct a test for a case in the recursion*

**Description**

This function creates an if-statement for testing if a case can be applied.

**Usage**

```python
make_recursion_case(test_expr, value_expr, continue)
```

**Arguments**

- **test_expr**  
  The expression that must be true for the case to be applied
- **value_expr**  
  The value to compute if the test is true
- **continue**  
  The next case to check if this one isn’t true

**Value**

An if-statement for checking and potentially evaluating one case.
make_update_expr

String together the case if-statements of a recursion.

**Description**

String together the case if-statements of a recursion.

**Usage**

make_update_expr(ranges, patterns, conditions, recursions)

**Arguments**

- **ranges**  
  The ranges specification
- **patterns**  
  The patterns specification
- **conditions**  
  The conditions specifications
- **recursions**  
  The recursions specification

**Value**

A series of if-else-statements for evaluating a recursion.

---

parse_ranges

Parser for the ranges part of a specification.

**Description**

Parses the ranges and return a list of index variables an the values they should iterate over. The ranges are returned as a list with the range variables as its names and the range values as the list components.

**Usage**

parse_ranges(ranges)

**Arguments**

- **ranges**  
  The quosure wrapping the input to the specification.

**Value**

A parsed specification for ranges.
parse_recursion  
*Parser for the recursion part of a specification.*

**Description**

Parse the recursion part of an expressions.

**Usage**

```
parse_recursion(recursion)
```

**Arguments**

- **recursion**
  The quosure wrapping the recursion of the specification.

**Details**

The parser return a list with the following components:

- **recursion_env**: The environment in which expressions should be evaluated.
- **patterns**: A list of patterns, one per recursion case.
- **conditions**: A list of conditions, one per recursion case.
- **recursions**: A list of expressions, one per recursion case.

**Value**

A parsed specification for recursions.

%where%

*Connects a recursion with sequences it should recurse over.*

**Description**

This function parses a dynamic programming recursion expression and evaluates it, returning the table that the recursions specify.

**Usage**

```
recursion %where% ranges
```

**Arguments**

- **recursion**
  Specification of the dynamic programming recursion.
- **ranges**
  Specification of the index-ranges the recursion should compute values over.
Value

A filled out dynamic programming table.

Examples

```r
# Fibonacci numbers
fib <- {
  F[n] <- 1 ? n <= 2
} %where% {
  n <- 1:10
}
fib

# Edit distance
x <- c("a", "b", "c")
y <- c("a", "b", "b", "c")
edit <- {
  E[1,j] <- j - 1
  E[i,1] <- i - 1
  E[i,j] <- min(
    E[i - 1,j] + 1,
    E[i,j - 1] + 1,
    E[i - 1,j - 1] + (x[i - 1] != y[j - 1])
  )
} %where% {
  i <- 1:(length(x) + 1)
  j <- 1:(length(y) + 1)
}
edit
```
Index

%where%, 6

eval_recursion, 2

get_table_name, 2

make_condition_checks, 3
make_pattern_match, 3
make_pattern_match(), 4
make_pattern_tests, 4
make_recursion_case, 4
make_update_expr, 5

parse_ranges, 5
parse_recursion, 6