Loop Patterns: Extension of Kleene Star Operator for More Powerful Pattern Matching against Arbitrary Data Structures

SATOSHI EGI, Rakuten Institute of Technology, Japan

The Kleene star operator is an important pattern construct for representing a pattern that repeats multiple times. Due to its simplicity and usefulness, it is imported into various pattern-matching systems other than regular expressions. For example, Mathematica has a similar pattern construct called the repeated pattern. However, they have the following limitations: (i) We cannot change the pattern repeated depending on the current repeat count, and (ii) we cannot apply them to arbitrary data structures such as trees and graphs other than lists. This paper proposes the *loop patterns* that overcome these limitations. This paper presents numerous working examples and formal semantics of the loop patterns. The examples in this paper are coded in the Egison programming language, which features the customizable non-linear pattern-matching facility for non-free data types.

CCS Concepts: • **Software and its engineering** → **General programming languages**;

Additional Key Words and Phrases: loop pattern, repeated pattern, pattern matching, non-linear pattern, backtracking

1 INTRODUCTION

The Kleene star operator [15], sometimes called the repeated pattern [4], is an important pattern construct for representing a pattern that repeats multiple times. It allows us to represent patterns such as `[a,a,...]` (repetition of a), `[a,b,a,b,...]` (repetition of `[a,b]`), and `[a,b,a,a,b,...]` (repetition of a or b) for lists. Due to its simplicity and usefulness, it is implemented in various pattern-matching systems. For example, Mathematica [4] and Racket [19] have the repeated pattern. Domain specific languages such as Parsing expression grammars [14] and graph query languages such as Cypher [3] and Gremlin [17] also have the Kleene star like operator.

However, the repeated patterns have the following two limitations.

(1) We cannot change the content of the pattern repeated depending on the repeat count.
(2) We can apply the repeated pattern only for lists.

The first limitation becomes a problem, for example, when we consider the pattern that matches with lists of lists that the n-th list contains n elements such as `[[a],[a,a]]` and `[[a],[a,a],[a,a,a]]`. The repeated patterns cannot represent such a pattern because they do not provide users a method for referring to the current repeat count in the pattern. The repeated patterns also does not allow us to refer to the values bound to the patterns repeated by $i - 1$ times when we pattern-match the n-th repeat of the pattern repeated. Consequently, we cannot describe a pattern that matches lists whose elements are in sequence such as `[1,2,3]` and `[2,3,4]` with the repeated patterns.

Author's address: Satoshi Egi, Rakuten Institute of Technology, Japan.

The second limitation is crucial for the general-purpose programming languages. The reason for the second limitation is that a pattern repeated is always appended to the tail of the pattern. For that reason, the repeated patterns can be applied only to collection data such as lists and multisets. [1] As a result, for example, we cannot use the repeated patterns to traverse the ancestors or descendants of a tree node.

This paper presents the loop patterns that overcome the above two limitations. We show the numerous sample programs of the loop patterns coded in the Egison programming language, which features the user-customizable non-linear pattern matching with backtracking for arbitrary user-defined data types [1, 11].

2 RELATED WORK

There are tremendous amount of research on pattern-matching extensions [2]. This section compares these studies with our proposal.

Some of these studies allow users to define pattern-matching methods for each pattern constructor [5, 12, 20, 21]. It enables users to customize a pattern-matching method for arbitrary data types by themselves. However, these proposals do not discuss on the repeated patterns. For example, active patterns [12, 18] applies its pattern-matching facility to graphs [13]. However, they use recursive functions for traversing graphs. These proposals that provide a customizable pattern-matching facility do not support non-linear pattern matching with multiple results except for Egison [11]. The repeated patterns are powerful especially when it is combined with non-linear pattern matching with backtracking. It may be the reason that these proposals does not discuss the extension of the repeated patterns.

There is another approach, regular-tree expressions that extends regular expressions for handling trees. Recursively defined patterns proposed in the *trx* regular-tree expression language [6] overcomes the second limitation mentioned in Section 1, and enables us to apply repeated patterns to other than lists. However, they still suffer from the first limitation. The reason is because the recursively defined patterns do not provide a method for managing the repeat count. The advantage of the loop patterns over the recursively defined patterns is its ability to refer to the repeat count in patterns.

3 PATTERN MATCHING OF EGISON

Before starting the discussion on the loop patterns, this section gives a brief overview of our language focusing on its pattern-matching facility. Our language features the user-customizable efficient non-linear pattern matching with backtracking. The loop pattern is even more powerful when combined with this feature.

Figure 1 shows the formal grammar of our language. Our language is a Scheme-based language with lazy evaluation strategy. The `match-all` and `match` expressions are syntax for pattern matching. This section explains the usage of these expressions. In Figure 1, *ident* stands for an identifier that begins with an lowercase letter, whereas *Ident* stands for an identifier that begins with an uppercase letter.

Our language uses four kinds of parenthesis in addition to "(" and ")", which denote function applications. "<" and ">" are used to apply pattern and data constructors. In our language, the name of a data constructor starts with uppercase, whereas the name of a pattern constructor starts with lowercase. "[" and "]" are used to build a tuple. "{" and "}" are used to denote a collection. "{|" and "|}" are used to denote a hash table.

The following program is a sample of the `match-all` expression for handling pattern matching with multiple results. In this paper, we show the evaluation result of a program in the comment that follows the program. ";" is the inline comment delimiter of our language.

```
(match-all {1 2 3} (list integer) [<join $xs $ys> [xs ys]])
; {[{} {1 2 3}] [{1} {2 3}] [{1 2} {3}] [{1 2 3} {}]}
```

---

[1]Strings, which are the target of regular expressions, are a list of characters.

⟨*expr*⟩ ::= ⟨*bool*⟩   (boolean)
 | ⟨*number*⟩   (number)
 | ⟨*ident*⟩['_'⟨*expr*⟩]*   (variable)
 | '<' ⟨*Ident*⟩ ⟨*expr*⟩* '>'   (algebraic data)
 | '[' ⟨*expr*⟩* ']'   (tuple)
 | '{' ⟨*expr*⟩* '}'   (collection)
 | '{|' ⟨*key-value*⟩* '|}'   (hash)
 | '(lambda [' '$'⟨*ident*⟩* ']' ⟨*expr*⟩ ')'   (function)
 | '(' ⟨*expr*⟩ ⟨*expr*⟩* ')'   (function application)
 | '(if' ⟨*expr*⟩ ⟨*expr*⟩ ⟨*expr*⟩ ')'   (if)
 | '(letrec {' ⟨*binding*⟩* '}' ⟨*expr*⟩ ')'   (letrec)
 | '(match-all' ⟨*expr*⟩ ⟨*expr*⟩ '[' ⟨*pattern*⟩ ⟨*expr*⟩ '])'   (match-all)
 | 'something'   (something build-in matcher)
 | '(matcher {' ⟨*matcher-clause*⟩* '})'   (matcher)

⟨*key-value*⟩ :: '[' ⟨*number*⟩ ⟨*expr*⟩ ']'   (hash key and value)

⟨*binding*⟩ ::= '[' '$'⟨*ident*⟩ ⟨*expr*⟩ ']'   (binding)

⟨*pattern*⟩ ::= '_'   (wildcard)
 | '$'⟨*ident*⟩['_'⟨*expr*⟩]*   (pattern variable)
 | ','⟨*expr*⟩   (value pattern)
 | '<' ⟨*ident*⟩ ⟨*pattern*⟩* '>'   (inductive pattern)
 | '(|' ⟨*pattern*⟩* ')'   (or-pattern)
 | '(&' ⟨*pattern*⟩* ')'   (and-pattern)
 | '!' ⟨*pattern*⟩   (not-pattern)
 | '(loop' '$'⟨*ident*⟩ ⟨*index-range*⟩ ⟨*pattern*⟩ ⟨*pattern*⟩')'   (loop pattern)

⟨*index-range*⟩ ::= '[' ⟨*expr*⟩ ⟨*expr*⟩ ⟨*pattern*⟩ ']'   (index range)

Fig. 1. Formal grammar of Egison

match-all is composed of an expression called *target*, *matcher*, and *match clause*, which consists of a *pattern* and *body expression*. The match-all expression evaluates the body of the match clause for each pattern-matching result and returns a collection that contains all results. In the above code, we pattern-match the target {1 2 3} as a list of integers using the pattern <join $xs $ys>. (list integer) is a matcher to pattern-match the pattern and target as a list of integer. The pattern is constructed using the join pattern constructor. $xs and $ys are called *pattern variables*. We can use the result of pattern matching referring to them. In the sample program, given a join pattern, (list integer) tries to divide the target collection into two collections. The collection {1 2 3} is thus divided into two collections by four ways.

Our language can handle non-linear patterns. In our language, a pattern is examined from left to right in order, and the binding to a pattern variable can be referred to in its right side of the pattern. In the following example, the pattern variable $p is bound to one of prime numbers. After that, the pattern ",(+ p 2)" is examined. A pattern that begins with "," is called a *value pattern*. The expression following "," can be any kind of expressions. The value patterns match with the target data if the target is equal to the content of the pattern. Therefore, after successful pattern matching, $p is bound to the first element of a prime twin. Also note that, match-all can handle pattern matching that may yield infinitely many results thanks to the lazy evaluation strategy.

```
(define $twin-primes
  (match-all primes (list integer)
    [<join _ <cons $p <cons ,(+ p 2) _>>> [p (+ p 2)]]]))

(take 6 twin-primes) ; {[3 5] [5 7] [11 13] [17 19] [29 31] [41 43]}
```

Egison supports non-linear patterns instead of pattern guards because of its efficiency. Implementation of efficient backtracking for non-linear patterns is easier than that for pattern guards that enumerate all matches before filtering the results using conditions described using pattern guards. For example, the time complexity of the following pattern matching is $O(n^2)$. This pattern matching checks weather the target collection contains an identical triple or not. between is a function that return a collection consisting of the integers between the first and second arguments. Therefore, the target collection contains no identical triple in this case.

```
(match-all (between 1 n) (multiset integer)
  [<cons $x <cons ,x <cons ,x _>>> x])
; returns {} in O(n^2)
```

On the other hand, the time complexity of the following Curry program is $O(n^3)$.

```
insert x [] = [x]
insert x (y:ys) = x:y:ys ? y:(insert x ys)

tri (insert x (insert x (insert x _))) = "Matched"
tri _ = "Not matched"

tri [1,2,3,4,5,6,7,8,9,10]  -- returns "Not matched" in O(n^3) time
```

The reason is because Curry transforms non-linear patterns into pattern guards as follows. It enumerates all $\binom{n}{3}$ candidates before filtering them.

```
tri' (insert x (insert y (insert z _))) | x == y && x == z = "Matched"
tri' _ = "Not matched"
```

There is another primitive syntax called match expression. The major difference from match-all is that it can take multiple match clauses. It tries pattern matching starting from the head of the match clauses, and tries the next clause if it fails. Therefore, it is useful when we want to write conditional branching. Another difference is that match only evaluates the first result of the results of the pattern matching, while match-all returns a collection of all results.

The matcher expression is a syntax to define a matcher. Users can define the pattern-matching method for each data type using the matcher expression. The usage of the matcher expression is explained in [11]. Extensibility of pattern matching is the core feature of our language.

All the matchers that appears in this paper including integer, string, list, multiset, and set are defined in our language using the matcher expression. However, in reading this paper, we can consider them as built-in matchers.

4 LOOP PATTERNS

This section presents the loop patterns that overcome the limitations of the repeated patterns.

4.1 Usage of Loop Patterns

This section explains the grammar and usage of the loop patterns.

First, let us consider pattern matching for enumerating all combinations of two elements from a target collection. It can be written using match-all as follows.

```
(define $comb2
  (lambda [$xs]
```

```
  (match-all xs (list integer)
    [<join _ <cons $x_1 <join _ <cons $x_2 _>>>>
     {x_1 x_2}])))
```

```
(comb2 {1 2 3 4}) ; {{1 2} {1 3} {2 3} {1 4} {2 4} {3 4}}
```

Our language allows users to append indices to a pattern variable as $x_1 and $x_2 in the above sample. They are called *indexed variables* and represent x_1 and x_2 in mathematical expressions. The expression after '_' must be evaluated to an integer and is called an *index*. We can append as many indices as we want. When a value is bound to an indexed pattern variable $x_i, the system initiates a hash if x is not bound to a hash, and bind it to x. If x is already bound to a hash, the new key and value is added to the hash.

Next, let us consider pattern matching for enumerating all combinations of *three* elements from the target collection. It can be written by only modifying the above program a bit as follows.

```
(define $comb3
  (lambda [$xs]
    (match-all xs (list integer)
      [<join _ <cons $x_1 <join _ <cons $x_2 <join _ <cons $x_3 _>>>>>>
       {x_1 x_2 x_3}])))
```

```
(comb3 {1 2 3 4}) ; {{1 2 3} {1 2 4} {1 3 4} {2 3 4}}
```

Now, let us generalize comb2 and comb3. The loop patterns can be used for that purpose.[2]

```
(define $comb
  (lambda [$n $xs]
    (match-all xs (list integer)
      [(loop $i [1 {n} _]
         <join _ <cons $x_i ...>>
         _)
       (map (lambda [$i] x_i) (between 1 n))])))
```

```
(comb 2 {1 2 3 4}) ; {{1 2} {1 3} {2 3} {1 4} {2 4} {3 4}}
(comb 3 {1 2 3 4}) ; {{1 2 3} {1 2 4} {1 3 4} {2 3 4}}
```

As shown in Figure 1, the loop pattern takes an *index variable*, *index range*, *repeat pattern*, and *end pattern* as arguments. An index variable is a variable to hold the current repeat count. An index range specifies the range where the index variable moves. A repeat pattern is a pattern repeated when the index variable is in the index range. An end pattern is a pattern expanded when the index variable get out of the index range.

Inside loop patterns, we can use the *ellipsis pattern* (...). The repeat pattern or the end pattern is expanded at the location of the ellipsis pattern. When the repeat pattern is expanded replacing the ellipsis pattern, the index variable is incremented. When the end pattern is expanded replacing the ellipsis pattern, the end number pattern is pattern-matched with the value bound to the index variable at that time.

The index range is composed of a *start number*, *end numbers*, and *end number pattern*. A start number specifies the initial value of the index variable. End numbers are a sorted list of integers. When the current value of index variable is not in the end numbers, the ellipsis pattern is replaced only with the repeat pattern. When the index variable reaches one of the end numbers, the ellipsis pattern is replaced with both of the end pattern and the repeat pattern. When the index variable reaches the last end number, the ellipsis pattern is replaced only with the end pattern.

In the above sample, the start number is 1 and the end numbers are {n}. In this case, the end number can be only n. Therefore, the index variable moves from 1 to n.

---

[2]This generalization can be done by also the repeated patterns. We show pattern matching that can be done only by the loop patterns in the next section.

If we use a loop pattern we can redefine `comb2` as follows. In this sample, the index range is [1 {2} _]. The start number is 1, which means the index variable i starts from 1. The end numbers are {2}, which means the index variable i moves to 2 and the pattern is repeated twice. The end number pattern is the wildcard _. In this case, the end number can be only 2, therefore, we do not need to bind it to a pattern variable.

```
(define $comb2
  (lambda [$xs]
    (match-all xs (list integer)
      [(loop $i [1 {2} _]
         <join _ <cons $x_i ...>>
         _)
       {x_1 x_2}])))
```

```
(comb2 {1 2 3 4})
; {{1 2} {1 3} {2 3} {1 4} {2 4} {3 4}}
```

The following program is an example that utilize both of end numbers and an end number pattern. The `comb2or3` function returns all combinations of two or three elements from the argument collection. 2 or 3 is bound to $n because the end number pattern $n and the index variable that reached the end numbers 2 or 3 are pattern-matched.

```
(define $comb2or3
  (lambda [$xs]
    (match-all xs (list integer)
      [(loop $i [1 {2 3} $n]
         <join _ <cons $x_i ...>>
         _)
       (map (lambda [$i] x_i) (between 1 n))])))
```

```
(comb2or3 {1 2 3 4})
; {{1 2} {1 3} {2 3} {1 4} {2 4} {3 4} {1 2 3} {1 2 4} {1 3 4} {2 3 4}}
```

We prepared several syntax sugar for the index range as follows. It allows us to omit end numbers, an end number pattern, or both.

```
[ <start-num> ].              ;=> [ <start-num> (from <start-num>) _ ]
[ <start-num> <end-numbers> ] ;=> [ <start-num> <end-numbers> _ ]
[ <start-num> <end-number> ]  ;=> [ <start-num> {<end-number>} _ ]
[ <start-num> <end-pattern> ] ;=> [ <start-num> (from start-num) <end-pattern> ]
```

A start number cannot be omitted. When end numbers are omitted, (from <start number>) is complemented. `from` is a function that returns the infinite sequence of integers that starts from its argument. If end numbers are not a collection but a single number, it is converted to the collection that contains only that end number. When an end number pattern is omitted, the wildcard _ is complemented.

If we use these syntax sugar, we can rewrite the index range of `comb2` to [1 2] as follows.

```
(define $comb2
  (lambda [$xs]
    (match-all xs (list integer)
      [(loop $i [1 2]
         <join _ <cons $x_i ...>>
         _)
       {x_1 x_2}])))
```

```
(comb2 {1 2 3 4})
; {{1 2} {1 3} {2 3} {1 4} {2 4} {3 4}}
```

4.2 Advantages of Loop Patterns

The first characteristic of the loop patterns is that we explicitly name an index variable ($i and $j in the following
example). We can parameterize repeat patterns by a current repeat count by referring to the index variable.

The following program describes a pattern that matches with lists of lists whose i-th list contains i elements.
Note that we can nest the loop patterns. We refers to the index variable i of the first loop pattern in the end
number of the second loop pattern.

```
(match {{1} {2 2} {3 3 3} {4 4 4 4}} (list (list integer))
  {[(loop $i [1 $n]
     <cons (loop $j [2 i]
            <cons _ ...>
            <nil>)
          ...>
       <nil>)
    #t]
    [_ #f]})
; #t
```

The pattern in the following program matches if the elements of a target list are in sequence. We are referring
to the value bound in the pattern previously repeated as ,(+ 1 x_(- i 1)). Note that index variables play
the crucial roles in this pattern. In Mathematica [4] and Racket [19], the values bound to a repeated pattern are
stored as elements of a collection. However, they are stored in a hash table in our language. It allows us to easily
access the value bound to the pattern variables in the loop pattern. In this example, the and-pattern is used as
an as-pattern.

```
(match {1 2 3 4 5} (list integer)
  {[<cons $x_1
     (loop $i [1 $n]
       <cons (& ,(+ 1 x_(- i 1)) $x_i)
         ...>
     <nil>)>
    #t]
    [_ #f]})
; #t
```

In Section 5.1 and 5.3, we show more interesting samples utilizing this characteristic of the loop patterns.

The second characteristic of the loop patterns is that we can place a ellipsis pattern in a place where we like.
This feature allows us to use the loop patterns for arbitrary user-defined data structures, not only for collections.
For example, we can apply the loop patterns for trees (Section 5.2).

5 EXAMPLES

This section demonstrates working examples of the loop patterns.

5.1 N-queens Solver

Figure 2 shows the four queens solver. n-queen problem is the problem of the placing n chess queens on an $n \times n$
board such that no queen is able to attack any of the others. The chess queen can attack other chess pieces on
the same row, column, and diagonal.

In Figure 2, we represent the positions of the four queens with a list. The number of the n-th element represents
the row number of the queen of the n-th line. The solution must be a rearrangement of the list {1 2 3 4} because
no two queens can be in the same line or row. Therefore, we pattern-match a collection {1 2 3 4} as a multiset of
integers. The requirement that all two queens must not share the same diagonal is represented with conditions

```
(define $four-queens
  (match-all {1 2 3 4} (multiset integer)
    [<cons $a_1
      <cons (& !,(- a_1 1) !,(+ a_1 1) $a_2)
       <cons (& !,(- a_1 2) !,(+ a_1 2) !,(- a_2 1) !,(+ a_2 1) $a_3)
        <cons (& !,(- a_1 3) !,(+ a_1 3) !,(- a_2 2) !,(+ a_2 2) !,(- a_3 1) !,(+ a_3 1) $a_4)
         <nil>>>>>
    {a_1 a_2 a_3 a_4}]))

four-queens ; {{2 4 1 3} {3 1 4 2}}
```

Fig. 2. Four queens solver

```
(define $n-queens
  (lambda [$n]
    (match-all (between 1 n) (multiset integer)
      [<cons $a_1
        (loop $i [2 n]
              <cons (loop $j [1 (- i 1)]
                          (& !,(- a_j (- i j)) !,(+ a_j (- i j)) ...)
                          $a_i)
                    ...>
              <nil>)>
      a]))))

(n-queens 4) ; {{|[1 2] [2 4] [3 1] [4 3]|} {|[1 3] [2 1] [3 4] [4 2]|}}
```

Fig. 3. n-queens solver

$a_1 \pm 1 \neq a_2$, $a_1 \pm 2 \neq a_3$, $a_2 \pm 1 \neq a_3$, $a_1 \pm 3 \neq a_4$, $a_2 \pm 2 \neq a_4$, and $a_3 \pm 1 \neq a_4$. *Not-patterns* are used to represent $\neq$. A not-pattern matches with a object if the object does not matches the pattern following after '!'.

Next, let us consider defining five-queens, six-queens, seven-queens, eight-queens solvers, and so on. Patterns for these solvers have a similar form and it would be better to define the general n-queens solver at once than to define each of them. The loop patterns can be used for that.

We can generalize four-queens to n-queens solver using loop patterns as Figure 3. A double loop pattern is used to express the pattern for n-queens solver.

If we use loop patterns in a pattern, the numbers of the pattern variables in the pattern can change by the parameter. It is the reason why we introduced indexed variables.

This pattern is an example that can be described only by the loop patterns because

(1) it refers to the index variable i (the current repeat count) of the first loop pattern in the end number of the second loop pattern, and
(2) it refers to the value bound in the previously repeated pattern as ",a_j".

5.2 Pattern Matching for Trees

Figure 4 demonstrates pattern matching for trees. A matcher for trees is defined using the algebraic-data-matcher expression [9]. Trees have two data constructors, leaf and node. leaf obtains one argument and it is pattern-matched as a string. node obtains two arguments, and the first and second argument are pattern-matched as a string and a multiset of trees, respectively.

```
(define $tree (algebraic-data-matcher {<leaf string> <node string (multiset tree)>}))

(define $tree-data
  <Node "Programming language"
    {<Node "Pattern-matching-oriented"
       {<Leaf "Egison">}>
     <Node "Functional language"
       {<Node "Strictly typed"
          {<Leaf "OCaml"> <Leaf "Haskell"> <Leaf "Curry"> <Leaf "Coq">}>
        <Node "Dynamically typed"
          {<Leaf "Egison"> <Leaf "Lisp"> <Leaf "Scheme"> <Leaf "Racket"> <Leaf "Clojure">}>
       }>
     <Node "Logic programming"
       {<Leaf "Prolog"> <Leaf "LiLFeS"> <Leaf "Curry">}>
     <Node "Object oriented"
       {<Leaf "C++"> <Leaf "Java"> <Leaf "Ruby"> <Leaf "Python"> <Leaf "OCaml">}>
    }>)

;; All categories that Egison belongs.
(match-all tree-data tree
  [(loop $i [1 $n]
          <node $c_i <cons ... _>>
          <leaf ,"Egison">)
   c]))
;{{|[1 "Programming language"] [2 "Pattern-matching-oriented"]|}
  {|[1 "Programming language"] [2 "Functional language"] [3 "Dynamically typed"]|}}
```

Fig. 4. Pattern matching for trees

In Figure 4, `tree-data` defines a category tree of programming languages. For example, `"Egison"` belongs to the `"Pattern-matching-oriented"` category, and the `"Dynamically typed"` sub-category of the `"Functional programming"` category.

The `match-all` expression in Figure 4 enumerates all categories to which `"Egison"` belongs. A loop pattern is necessary to describe a pattern for this purpose because leaves can be appear in arbitrary depth.

This pattern matching is an illustrative example to show the importance of the ellipsis pattern. The characteristic that we can choose the position where the repeat pattern is expanded allows us to apply the loop patterns to trees.

XML path language [7] is a domain specific language for pattern matching against XML trees. In XML path language, we use the built-in `ancestor` command to enumerates all ancestors of a node. However, in our language, we can do the same thing by just combining the loop patterns and a few simple pattern constructors.

5.3 Pattern Matching for Graphs

In Figure 5, a graph is pattern-matched as a set of edges. This pattern is an example that can be described only by the loop pattern. The reason is because it refers to the value bound in the previously repeated pattern as ",x_(- i 1)".

We can use the `let` expression inside a pattern. A `let` expression inside a pattern is called a *let pattern*. In this pattern, the let pattern is effectively used to bind s to $x_1. Thanks to this let pattern, we do not need the special treatment for $x_1.

In Figure 6, a graph is pattern-matched as an adjacency list. `graph-data` in Figure 6 represents the railway network in Tokyo. `"Tokyo"`, `"Shibuya"`, `"Mitaka"`, `"Kinshicho"`, and `"Kitasenju"` are names of the train stations in Tokyo. The integers in `graph-data` are train fees.

```
(define $graph (set edge))
(define $edge (algebraic-data-matcher {<edge integer integer>}))
(define $graph-data {<Edge 1 4> <Edge 2 1> <Edge 3 1> <Edge 3 2> <Edge 4 3> <Edge 5 1> <Edge 5 4>})

;; The shortest path paths from 's' to 'e'.
(car (let {[$s 1] [$e 2]}
       (match-all graph-data graph
         [(let {[$x_1 s]}
            (loop $i [2 $n]
              <cons <edge ,x_(- i 1) $x_i> ...>
              <cons <edge ,x_(- n 1) (& ,e $x_n)> _>))
          (map (lambda [$i] x_i) (between 1 n))])))
; {1 4 3 2}
```

Fig. 5. Pattern matching for graphs as a set of edges

```
(define $station string)
(define $price   integer)
(define $graph   (multiset [station (multiset [station price])]))

(define $graph-data
  {
  ["Tokyo"     {              ["Shinjuku" 200] ["Shibuya" 200] ["Mitaka" 390] ["Kinshicho" 160] ["Kitasenju" 220]}]
  ["Shinjuku"  {["Tokyo" 200]                  ["Shibuya" 160] ["Mitaka" 220] ["Kinshicho" 220] ["Kitasenju" 310]}]
  ["Shibuya"   {["Tokyo" 200] ["Shinjuku" 160]                 ["Mitaka" 310] ["Kinshicho" 220] ["Kitasenju" 310]}]
  ["Mitaka"    {["Tokyo" 390] ["Shinjuku" 220] ["Shibuya" 310]                ["Kinshicho" 470] ["Kitasenju" 550]}]
  ["Kinshicho" {["Tokyo" 160] ["Shinjuku" 220] ["Shibuya" 220] ["Mitaka" 470]                   ["Kitasenju" 220]}]
  ["Kitasenju" {["Tokyo" 220] ["Shinjuku" 310] ["Shibuya" 310] ["Mitaka" 550] ["Kinshicho" 220]                 }]
  })

;; List all routes that visit all cities exactly once and return to Tokyo.
(define $trips
  (match-all graph-data graph
    [<cons [,"Tokyo" <cons [$s_1 $p_1] _>]
           (loop $i [2 5]
             <cons [,s_(- i 1) <cons [$s_i $p_i] _>]
                   ...>
             <cons [,s_5 <cons [(& ,"Tokyo" $s_6) $p_6] _>]
                   _>)>
     [(sum (map (lambda [$i] p_i) (between 1 6)))
      s]]))
```

Fig. 6. Pattern matching for graphs as an adjacency list

The pattern in Figure 6 lists all routes that visit all cities exactly once and return to Tokyo. This pattern can be used for solving the traveling salesman problem. It is also an example that can be described only by the loop patterns because it refers to the value bound in the previously repeated pattern as ",s_(- i 1)".

There are several graph database query languages [3, 16, 17]. The advantages of our language against these query languages are as follows.

Our language does not focus on pattern matching for graphs. However, we can represent various patterns by combining the loop patterns and a small number of simple pattern constructors.

5.4 take and drop

Pattern matching with backtracking allows us to define the basic functions for list programming more concisely than the traditional functional definitions. The reason is because we can hide recursion to traverse the elements of lists by pattern matching. For example, the map function can be defined as follows [10]. something in the following code is the only built-in matcher of our pattern-matching system. something can handle only wildcards and pattern variables, and is used to bind a value to a pattern variable.

```
(define $map
 (lambda [$f $xs]
  (match-all xs (list something)
   [<join _ <cons $x _>> (f x)])))
```

The loop patterns are useful to simplify some basic functions with pattern matching. For example, the take and drop functions are implemented using loop patterns as follows.

```
(define $take
 (lambda [$n $xs]
  (match xs (list something)
   {[(loop $i [1 n] <cons $x_i ...> _) (map (lambda [$i] x_i) (between 1 n))]
    [_ xs]})))

(define $drop
 (lambda [$n $xs]
  (match xs (list something)
   {[(loop $i [1 n] <cons _ ...> $ys) ys]
    [_ {}]})))
```

6 DISCUSSION

This section discusses the reasonableness of the design of the loop patterns.

6.1 Necessity of End Numbers

If we write [1 ,2] instead of [1 2], and [1 (& (| ,2 ,3) $n)] instead of [1 {2 3} $n], we can remove the end numbers from the index range. It makes the syntax of the loop patterns simpler. However, cases in which the loop pattern with this syntax cannot be executed correctly exist.

For example, the program for the n-queens solver causes a runtime error if we replace the index range from [1 (- i 1)] to [1 ,(- i 1)]. The reason is because the index variable j does not stop at i−1 and moves to i. Therefore, the program causes an unbound variable error when the system refers to the value of a_j inside the value pattern.

If the index range is [1 2], the index variable moves 1 and 2. However, if the index range is [1 ,2], the index variable cannot stop at 2 and moves 1, 2, and 3. This is because the system does not detect the integers greater than 2 do not match with the pattern ",2". We allow any patterns for the end pattern and there are cases that value patterns are inside an or-pattern, and-pattern, or more complicated pattern. However, a special treatment for value patterns makes the implementation of the loop pattern complicated and unnatural. This is the reason why the end pattern is necessary.

6.2 Computational Complexity of Loop Patterns

A loop pattern is expanded only when it is necessary. Therefore, the time complexity of pattern matching using loop patterns is completely same with pattern matching written not using them.

The computational complexity of Egison pattern matching is discussed in [11] in detail.

7 FORMAL SEMANTICS

Figure 7 shows the formal semantics of pattern matching of Egison including the loop patterns. This formal semantics is based on the formal semantics presented in [11]. The rules newly added for handling the loop patterns are highlighted to make the difference from [11] clear.

First, we explain the notations used in highlighted rules in Figure 7. $[a_i]_i$ denotes a list $[a_1, a_2, ...]$. $[[a_{ij}]_j]_i$ denotes a list of lists $[[a_{11}, a_{12}, ...], [a_{21}, a_{22}, ...]]$. Similarly, $(a_i)_i$ denotes a tuple $(a_1, a_2, ...)$. A list of tuples is denoted as $[a_i, b_i]_i$ omitting the parenthesis. $l_1 + l_2$ denotes a concatenation of lists l_1 and l_2. $a : l$ has the same meaning with $[a] + l$. ϵ denotes an empty list. $\vec{x}$ for some metavariable x is denotes a list of what x denotes.

Matching states have a stack of loop contexts in addition to a stack of matching atoms, an environment, and an intermediate result of pattern matching. Λ denotes a stack of loop contexts. A loop context consists of a binding information of an index variable, end numbers, and a tuple of an end number pattern, repeat pattern, and end pattern. p_1, p_2, and p_3 denote an end number pattern, repeat pattern, and end pattern, respectively. The stack of the loop contexts for an initial matching state is empty.

Four rules are newly added to the rules for matching states in order to handle the loop patterns and ellipsis patterns. When the pattern of the top matching atom is a loop pattern, a new loop context is created and pushed to the stack of the loop contexts. When the pattern of the top matching atom is an ellipsis pattern, the system refers to the top loop context to replace the ellipsis pattern to the end pattern or repeat pattern.

8 CONCLUSION

This paper proposed the loop patterns that overcome the limitations of the Kleene star operator and the repeated patterns and showed the several working examples that demonstrate the expressiveness of the loop patterns. We showed that the loop pattern can be used to represent more expressive patterns for lists. Furthermore, we also showed that the loop pattern can represent expressive patterns even for trees and graphs by only using a small number of simple pattern constructors.

We leave integration of our pattern-matching system with various query languages including SQL [8], XML path language [7] and graph query languages [3, 16, 17] as future work. These query languages are focusing on handling only their target data structures and have many built-in functions to handle various patterns. On the other hand, our pattern-matching system allows users to describe various patterns for various data types in a unified way with a small number of pattern constructors and the loop patterns. For this integration, we need to discuss an efficient execution method of our pattern-matching system and the loop patterns.

Research for finding simpler language constructs for constructing the loop patterns is also interesting because the semantics of the loop patterns presented in this paper is a bit complicated as a built-in language feature.

The loop pattern expands the range of algorithms we can describe concisely using the user-customizable efficient non-linear pattern-matching facility proposed in [11]. We hope this work leads to the further research of pattern matching as supportive evidence for the importance of pattern matching for more intuitive representation of algorithms.

REFERENCES

[1] 2011. The Egison Programming Language. https://www.egison.org. (2011). [Online; accessed 14-June-2018].
[2] 2015. ViewPatterns - GHC. https://ghc.haskell.org/trac/ghc/wiki/ViewPatterns. (2015). [Online; accessed 14-June-2018].
[3] 2016. The Neo4j Manual v2.3.3. https://neo4j.com/docs/stable/index.html. (2016). [Online; accessed 14-June-2018].

Evaluation of `matcher` and `match-all`:

$$\frac{}{\Gamma, (\text{matcher } [pp_i \ M_i \ [dp_j \ N_j]_j]_i) \Downarrow ([pp_i, M_i, [dp_j, N_j]_j]_i, \Gamma)}$$

$$\frac{\Gamma, M \Downarrow v \qquad \Gamma, N \Downarrow m \qquad [[[p \sim_m v], \Gamma, \varnothing, \epsilon]] \Rightarrow [\Delta_i]_i \qquad \Gamma \cup \Delta_i, L \Downarrow v_i \quad (\forall i)}{\Gamma, (\text{match-all } M \ N \ [p \ L]) \Downarrow [v_i]_i}$$

Matching states:

$$\frac{}{\epsilon \to \text{none, none, none}} \qquad \frac{}{(\epsilon, \Gamma, \Delta, \Lambda) : \vec{s} \to (\text{some } \Delta), \text{none}, (\text{some } \vec{s})}$$

$$\frac{\Gamma, M \Downarrow n \qquad \Gamma, N \Downarrow \{e_i\}_i \qquad ((\ldots \sim_m v) : \vec{a}, \Gamma, \Delta, (\{i \mapsto n-1\}, \{e_i\}_i, (p_1, p_2, p_3)) : \Lambda) : \vec{s} \to \text{none}, (\text{some } \vec{s'}), (\text{some } \vec{s})}{(((\text{loop } \$i \ [M \ N \ p_1] \ p_2 \ p_3) \sim_m v) : \vec{a}, \Gamma, \Delta, \Lambda) : \vec{s} \to \text{none}, (\text{some } \vec{s'}), (\text{some } \vec{s})}$$

$$\frac{i \ne e_1}{((\ldots \sim_m v) : \vec{a}, \Gamma, \Delta, (\{i \mapsto i\}, \{e_i\}_i, (p_i)_i) : \Lambda) : \vec{s} \to \text{none}, (\text{some}[((p_2 \sim_m v) : \vec{a}), \Gamma, \Delta, (\{i \mapsto i+1\}, \{e_i\}_i, [p_i]_i) : \Lambda]), (\text{some } \vec{s})}$$

$$\frac{i = e_1}{((\ldots \sim_m v) : \vec{a}, \Gamma, \Delta, (\{i \mapsto i\}, \{e_i\}_i, (p_i)_i) : \Lambda) : \vec{s} \to \text{none}, \left(\text{some}\left[\begin{array}{l}((p_1 \sim_{\text{something}} i) : (p_3 \sim_m v) : \vec{a}, \Gamma, \Delta, \Lambda), \\ ((p_2 \sim_m v) : \vec{a}, \Gamma, \Delta, (\{i \mapsto i+1\}, \{e_i\}_{i'}, (p_i)_i) : \Lambda)\end{array}\right]\right), (\text{some } \vec{s})}$$

$$\frac{i = e_1}{((\ldots \sim_m v) : \vec{a}, \Gamma, \Delta, (\{i \mapsto i\}, \{e_1\}, (p_i)_i) : \Lambda) : \vec{s} \to \text{none}, (\text{some}[((p_1 \sim_{\text{something}} i) : (p_3 \sim_m v) : \vec{a}), \Gamma, \Delta, \Lambda]), (\text{some } \vec{s})}$$

$$\frac{p \sim_m^{\Gamma \cup \Delta \ \cup \sum_i \Delta'_i} v \Downarrow [\vec{a}_i]_i, \Delta''}{((p \sim_m v) : \vec{a}, \Gamma, \Delta, \{\Delta'_i, \vec{e}_i, \vec{p}_i\}_i) : \vec{s} \to \text{none}, (\text{some}[\vec{a}_i + \vec{a}, \Gamma, \Delta \cup \Delta'', \{\Delta'_i, \vec{e}_i, \vec{p}_i\}_i), (\text{some } \vec{s})}$$

$$\frac{\vec{s}_i \to \text{opt } \Gamma_i, \text{opt } \vec{s'}_i, \text{opt } \vec{s''}_i \quad (\forall i)}{[\vec{s}_i]_i \Rightarrow \sum_i (\text{opt } \Gamma_i), \sum_i (\text{opt } \vec{s'}_i) + \sum_i (\text{opt } \vec{s''}_i)} \qquad \frac{}{\epsilon \Rightarrow \epsilon} \qquad \frac{\vec{s} \Rightarrow \vec{\Gamma}, \vec{s'} \qquad \vec{s'} \Rightarrow \vec{\Delta}}{\vec{s} \Rightarrow \vec{\Gamma} + \vec{\Delta}}$$

Matching atoms:

$$\frac{}{\$x \sim_{\text{something}}^{\Gamma} v \Downarrow [\epsilon], \{x \mapsto v\}} \qquad \frac{pp \approx^{\Gamma} p \Downarrow \textbf{fail} \qquad p \sim_{(\vec{\phi}, \Delta)}^{\Gamma} v \Downarrow \vec{a}, \Gamma'}{p \sim_{((pp, M, \vec{\sigma}):\vec{\phi}, \Delta)}^{\Gamma} v \Downarrow \vec{a}, \Gamma'}$$

$$\frac{pp \approx^{\Gamma} p \Downarrow [p'_i]_i, \Delta' \qquad dp \approx v \Downarrow \textbf{fail} \qquad p \sim_{((pp, M, \vec{\sigma}):\vec{\phi}, \Delta)}^{\Gamma} v \Downarrow \vec{a}, \Gamma'}{p \sim_{((pp, M, (dp, N):\vec{\sigma}):\vec{\phi}, \Delta)}^{\Gamma} v \Downarrow \vec{a}, \Gamma'}$$

$$\frac{pp \approx^{\Gamma} p \Downarrow [p'_j]_j, \Delta' \qquad dp \approx v \Downarrow \Delta'' \qquad \Delta \cup \Delta' \cup \Delta'', N \Downarrow [[v'_{ij}]_j]_i \qquad \Delta, M \Downarrow [m'_j]_j}{p \sim_{((pp, M, (dp, N):\vec{\sigma}):\vec{\phi}, \Delta)}^{\Gamma} v \Downarrow [[p'_j \sim_{m'_j} v'_{ij}]_j]_i, \varnothing}$$

Pattern matching on patterns:

$$\frac{}{\$ \approx^{\Gamma} p \Downarrow [p], \varnothing} \qquad \frac{\Gamma, M \Downarrow v}{,\$y \approx^{\Gamma}, M \Downarrow \epsilon, \{y \mapsto v\}} \qquad \frac{pp_i \approx^{\Gamma} p_i \Downarrow \vec{p}_i, \Gamma_i \quad (\forall i)}{<C \ pp_1 \ldots pp_n> \approx^{\Gamma} <C \ p_1 \ldots p_n> \Downarrow \sum_i \vec{p}_i, \bigcup_i \Gamma_i}$$

Pattern matching on data:

$$\frac{}{\$z \approx v \Downarrow \{z \mapsto v\}} \qquad \frac{dp_i \approx v_i \Downarrow \Gamma_i \quad (\forall i)}{<C \ dp_1 \ldots dp_n> \approx <C \ v_1 \ldots v_n> \Downarrow \bigcup_i \Gamma_i}$$

Fig. 7. Formal semantics of the loop patterns

[4] 2018. Repeated - Wolfram Language Documentation. http://reference.wolfram.com/language/ref/Repeated.html. (2018). [Online; accessed 14-June-2018].

[5] Sergio Antoy and Michael Hanus. 2010. Functional logic programming. *Commun. ACM* 53, 4 (2010).

[6] Ilya Bagrak and Olin Shivers. 2004. trx: Regular-tree expressions, now in Scheme. *The Scheme and Functional Programming Workshop* (2004).

[7] Anders Berglund, Scott Boag, Don Chamberlin, Mary F Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. 2003. XML Path Language (XPath). *World Wide Web Consortium (W3C)* (2003).

[8] Donald D Chamberlin and Raymond F Boyce. 1974. SEQUEL: A structured English query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*.

[9] Satoshi Egi. 2014. Non-linear Pattern Matching against Non-free Data Types with Lexical Scoping. *arXiv preprint arXiv:1407.0729* (2014).

[10] Satoshi Egi. 2015. Egison: Non-Linear Pattern-Matching against Non-Free Data Types. *arXiv preprint arXiv:1506.04498* (2015).

[11] Satoshi Egi and Yuichi Nishiwaki. 2018. Non-linear Pattern Matching with Backtracking for Non-free Data Types. *arXiv preprint arXiv:1808.10603* (2018).

[12] M. Erwig. 1996. Active patterns. *Implementation of Functional Languages* (1996).

[13] M. Erwig. 1997. Functional programming with graphs. In *ACM SIGPLAN Notices*, Vol. 32.

[14] Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, Vol. 39.

[15] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2013. *Introduction to automata theory, languages, and computation*. Pearson Education.

[16] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2006. Semantics and Complexity of SPARQL. In *International semantic web conference*.

[17] Marko A Rodriguez. 2015. The Gremlin Graph Traversal Machine and Language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*.

[18] Don Syme, Gregory Neverov, and James Margetson. 2007. Extensible pattern matching via a lightweight language extension. In *ACM SIGPLAN Notices*, Vol. 42.

[19] Sam Tobin-Hochstadt. 2011. Extensible pattern matching in an extensible language. *arXiv preprint arXiv:1106.2578* (2011).

[20] M. Tullsen. 2000. First Class Patterns. *Practical Aspects of Declarative Languages* (2000).

[21] P. Wadler. 1987. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*.