Finite Automata Theory Based Optimization of Conditional Variable Binding

An efficient type-aware `destructuring-case`

Jim Newton

12th European Lisp Symposium

1,2 April 2019

Our Goal

We would like to introduce a *user-defined* construct called `destructuring-case`, which efficiently selects a clause to evaluate designated by a destructuring lambda list depending on run-time value of a given expression.

There semantics of the macro usage should be intuitive.

There are several cases to consider.

Different number of required arguments

```
(destructuring-case expression
  ((X)
   (* X 100)
  ((X Y)
   (* X Y))
  ((X Y Z)
   (+ (* X Y) Z))))
```

Different optional arguments

```
(destructuring-case expression
  ((X &optional (Y 1))
   (* X Y))
  ((X &key (Y 1))
   (* X Y))
  ((X &key (Y 1) (Z 0) &allow-other-keys)
   (+ (* X Y) Z)))
```

Types of arguments

```
(destructuring-case expression
  ((X Y)
   (declare (type fixnum X Y))
   (* X Y))
  ((X Y)
   (declare (type fixnum X)
            (type integer Y))
   (* X Y))
  ((X Y)
   (declare (type (or string fixnum) X)
            (type number Y))
   (* (if (stringp X)
          (string-to-number X)
          X)
      Y)))
```

Efficient Type-Based Pattern Matching

Does this sequence:

`(a 8 8.0 b "a" "an" "the" c 8 88 888 d 8/3)`

`follow the pattern:` $(\mathit{symbol} \cdot (\mathit{number}^{+} \ \vee \mathit{string}^{+}))^{+}$ `?`

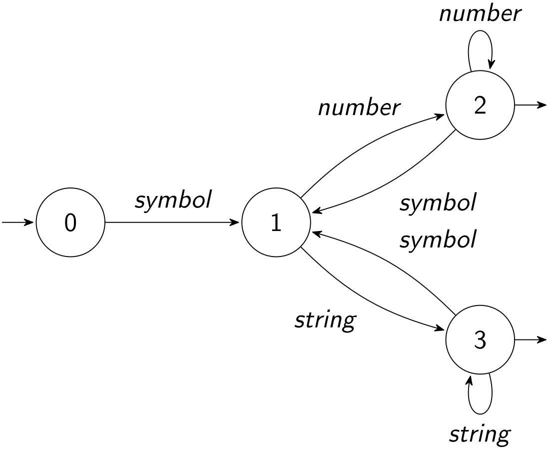
Does this sequence:

`(a 8 8.0 b "a" "an" "the" c 8 88 888 d 8/3)`

`follow the pattern:` $(symbol \cdot (number^{+} \ \vee string^{+}))^{+}$ `?`

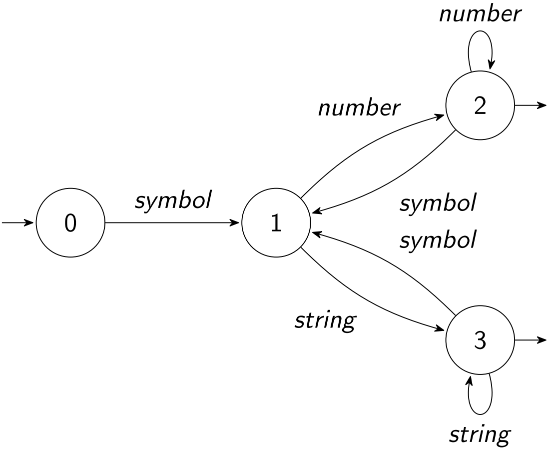
`We construct a` *`deterministic`* `finite automaton (DFA).`

`We want to support :not and :and in our DSL.`

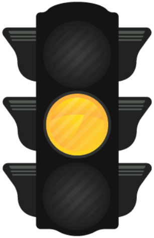


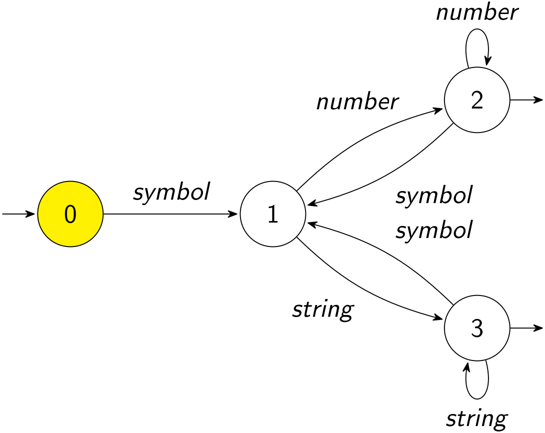
```
(a 8 8.0 b "a" "an" "the" c 8 88 888 d 8/3)
```

How does a DFA work as a type predicate?



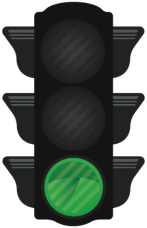
How does a DFA work as a type predicate?

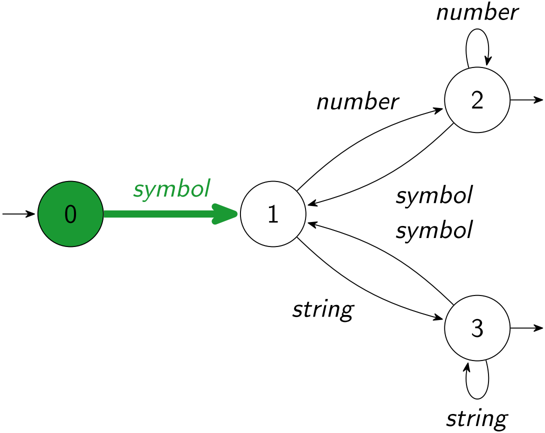





```
(a 8 8.0 b "a"
"an" "the" c 8
88 888 d 8/3)
```

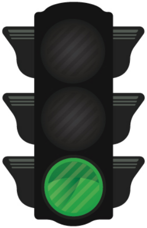
How does a DFA work as a type predicate?



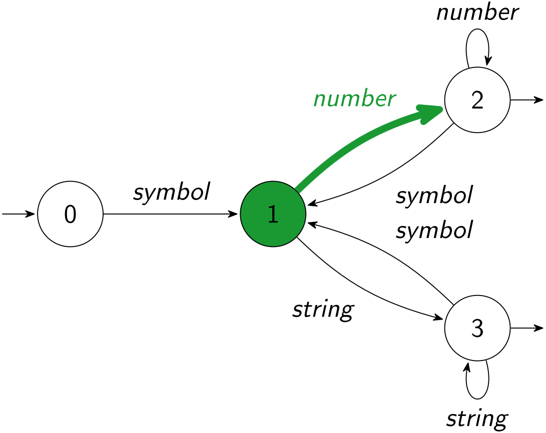



```
(a 8 8.0 b "a"
"an" "the" c 8
88 888 d 8/3)
```

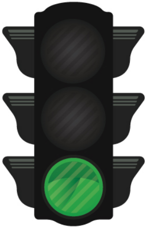
How does a DFA work as a type predicate?



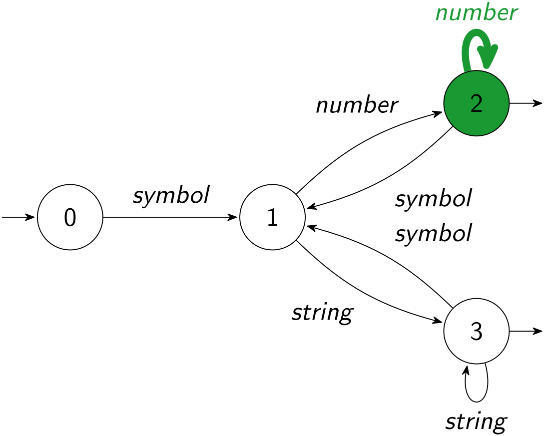
(a 8 8.0 "a" "an" "the" c 8 88 888 d 8/3)



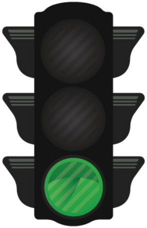
How does a DFA work as a type predicate?



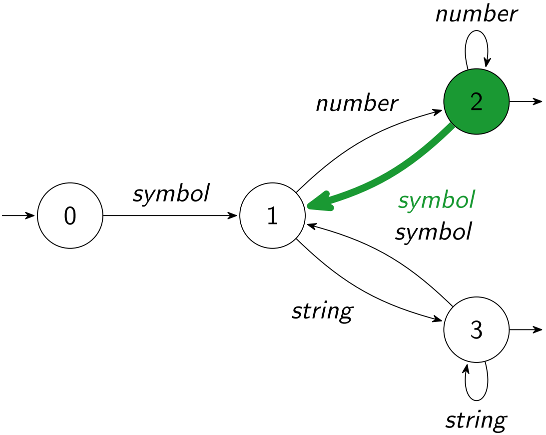
(a 8 8.0 b "a"
"an" "the" c 8
88 888 d 8/3)



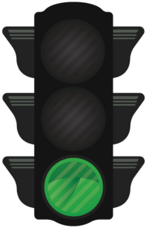
How does a DFA work as a type predicate?

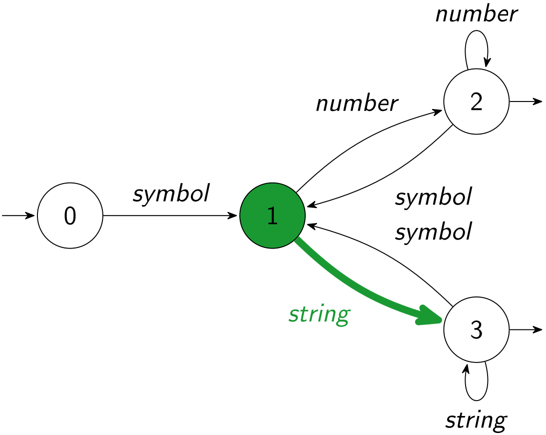


```
(a 8 8.0 b
"a" "an" "the"
c 8 88 888 d
8/3)
```



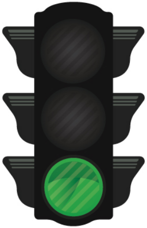
How does a DFA work as a type predicate?



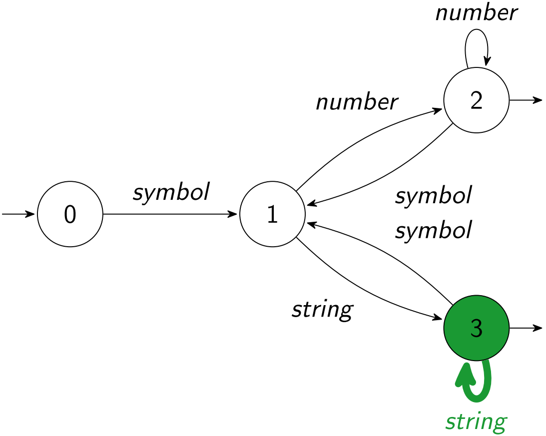


(a 8 8.0 b "a"
"an" "the" c 8
88 888 d 8/3)

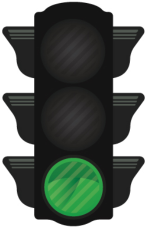
How does a DFA work as a type predicate?



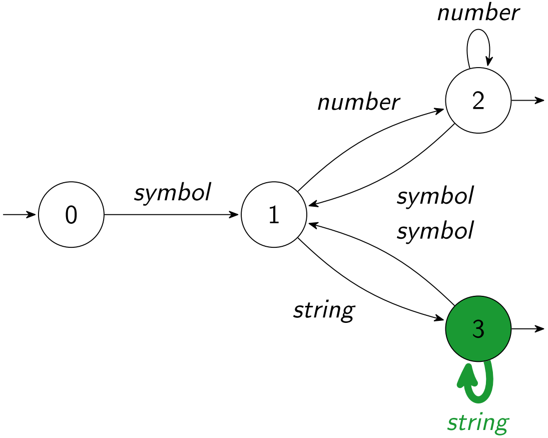
```
(a 8 8.0 b "a"
"an" "the" c 8
88 888 d 8/3)
```



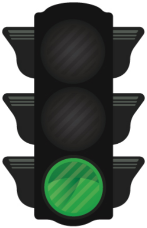
How does a DFA work as a type predicate?



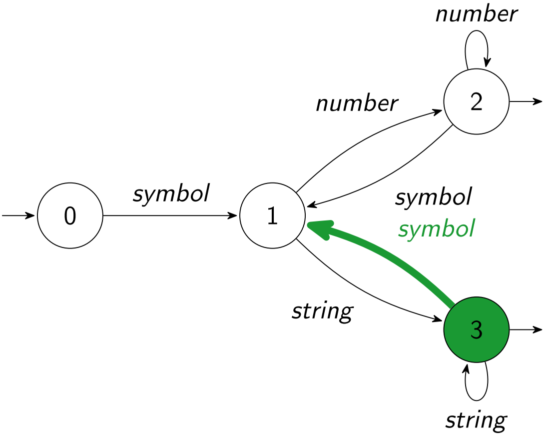
```
(a 8 8.0 b "a"
"an" "the" c 8
88 888 d 8/3)
```



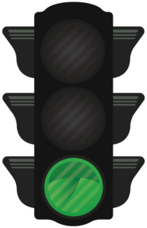
How does a DFA work as a type predicate?

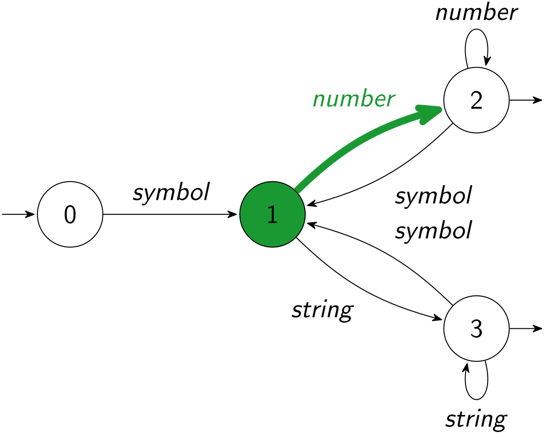


```
(a 8 8.0 b "a"
"an" "the" c 8
88 888 d 8/3)
```



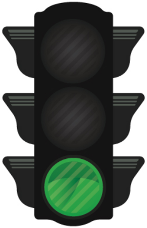
How does a DFA work as a type predicate?



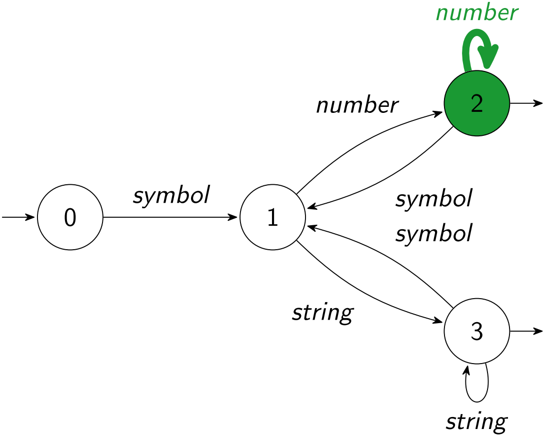



```
(a 8 8.0 b "a"
"an" "the" c
8 88 888 d
8/3)
```

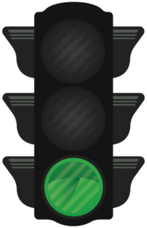
How does a DFA work as a type predicate?



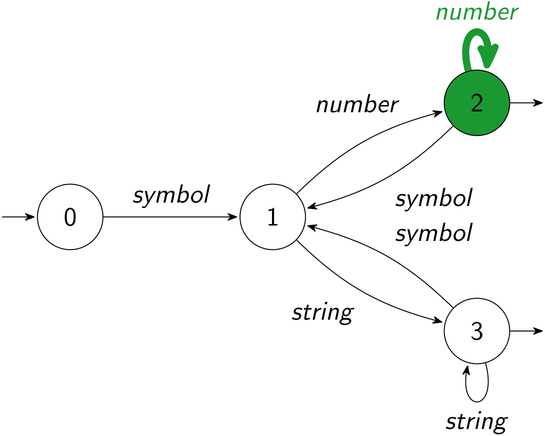
`(a 8 8.0 b "a" "an" "the" c 8 88 888 d 8/3)`



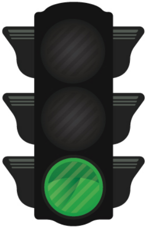
How does a DFA work as a type predicate?



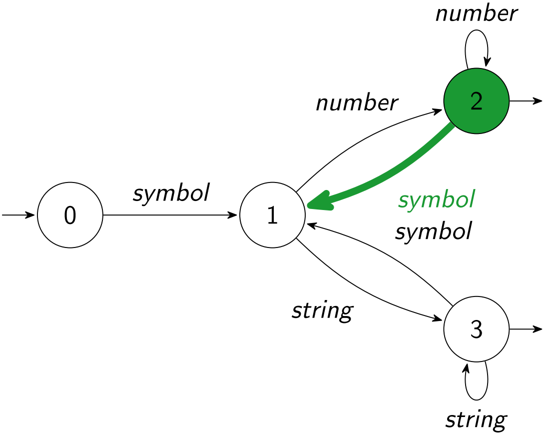
(a 8 8.0 b "a"
"an" "the" c 8
88 888 d 8/3)



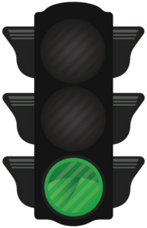
How does a DFA work as a type predicate?

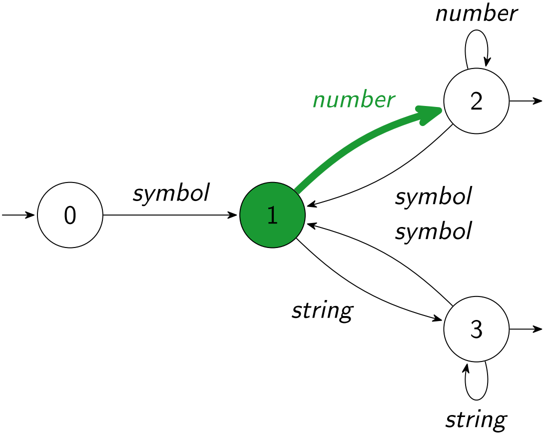


```
(a 8 8.0 b "a"
"an" "the" c 8
88 888 d 8/3)
```



How does a DFA work as a type predicate?

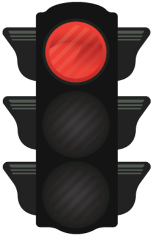




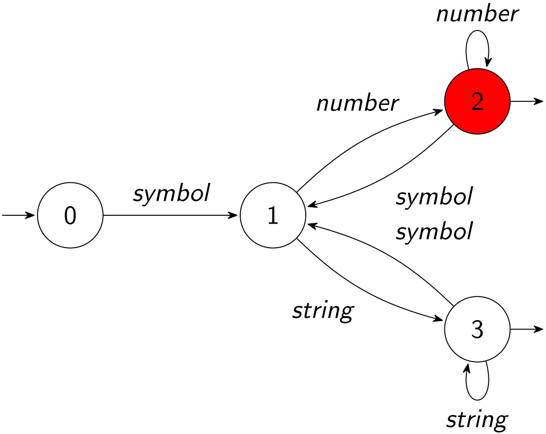
`(a 8 8.0 b "a" "an" "the" c 8 88 888 d 8/3)`

How does a DFA work as a type predicate?

Yes, it's a match!

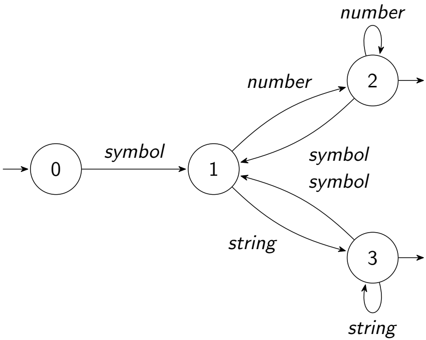


```
(a 8 8.0 b "a"
"an" "the" c 8
88 888 d 8/3)
```



Code generated from $(symbol \cdot (number^{+} \vee string^{+}))^{+}$

```
(tagbody
 0
   (unless seq (return nil))
   (typecase (pop seq)
     (symbol (go 1))
     (t (return nil)))
 1
   (unless seq (return nil))
   (typecase (pop seq)
     (number (go 2))
     (string (go 3))
     (t (return nil)))
 2
   (unless seq (return t))
   (typecase (pop seq)
     (number (go 2))
     (symbol (go 1))
     (t (return nil)))
```




```
 3
   (unless seq (return t))
   (typecase (pop seq)
     (string (go 3))
     (symbol (go 1))
     (t (return nil)))))
```

Introducing Regular Type Expression

A *Regular Type Expression (RTE)* is a surface syntax DSL expressing regular type patterns in sequences.

$$\left(symbol \cdot (rational^{*} \vee float^{+})\right) \quad \wedge \quad \overline{t \cdot ratio^{?} \cdot number}$$

RTE DSL notation:

```
(:and (:cat symbol
            (:or (:* rational)
                 (:+ float)))
      (:not (:cat t (:? ratio) number)))
```

Regular type expressions express components:

required, optional, repeating, and typed.

Destructuring Lambda lists as Patterns

Lambda-lists characterized by regular patterns

A lambda-list in Common Lisp has a fixed part

```
(destructuring-bind (a b)
    DATA
  ...)
```

Lambda-lists characterized by regular patterns

A lambda-list in Common Lisp has a fixed part, an optional part

```
(destructuring-bind (a b &optional c)
    DATA
  ...)
```

Lambda-lists characterized by regular patterns

A lambda-list in Common Lisp has a fixed part, an optional part, and a repeating part.

```
(destructuring-bind (a b &optional c &key x y)
    DATA
  ...)
```

Lambda-lists characterized by regular patterns

A lambda-list in Common Lisp has a fixed part, an optional part, and a repeating part part. Any of the variables may be restricted by type declarations.

```
(destructuring-bind (a b &optional c &key x y)
    DATA
  (declare (type integer a x)
           (type string b c y))
  ...)
```

Efficiently implementing `destructuring-case`

Macro: destructuring-case

```
(destructuring-case expression
  ((X Y)
   (declare (type fixnum X Y))
   :clause-1)
  ((X Y)
   (declare (type fixnum X)
            (type integer Y))
   :clause-2)
  ((X Y)
   (declare (type (or string fixnum) X)
            (type number Y))
   :clause-3))
```

Expansion of destructuring-case

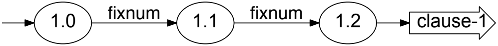
```
(rte-case expression
  ((:cat fixnum fixnum)
   (destructuring-bind (X Y) expression
     (declare (type fixnum X Y))
     :clause-1))

  ((:cat fixnum integer)
   (destructuring-bind (X Y) expression
     (declare (type fixnum X)
              (type integer Y))
     :clause-2))

  ((:cat (or string fixnum) number)
   (destructuring-bind (X Y) expression
     (declare (type (or string fixnum) X)
              (type number Y))
     :clause-3)))
```

Simplified rte-case expansion

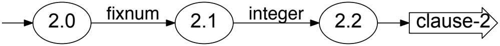
```
(rte-case expression
  ((:cat fixnum fixnum)
    :clause-1)
  ((:cat fixnum integer)
    :clause-2)
  ((:cat (or string fixnum) number)
    :clause-3))
```

Automata for clauses of `rte-case`

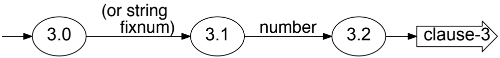
```
(rte-case expression
  ((:cat fixnum
         fixnum)
    :clause-1)
```




```
  ((:cat fixnum
         integer)
    :clause-2)
```

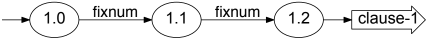



```
  ((:cat (or string
             fixnum)
         number)
    :clause-3))
```

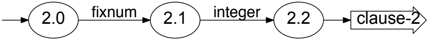


Automata for clauses of `rte-case`

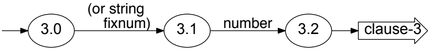
```
(rte-case expression
  ((:cat fixnum
         fixnum)
   :clause-1)
```




```
  ((:cat fixnum
         integer)
   :clause-2)
```



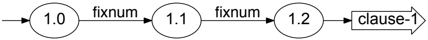

```
  ((:cat (or string
             fixnum)
         number)
   :clause-3))
```



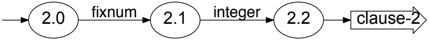
We could select the appropriate clause by executing the three automata in turn at run-time.

Automata for clauses of `rte-case`

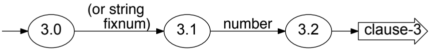
```
(rte-case expression
  ((:cat fixnum
         fixnum)
    :clause-1)
```




```
  ((:cat fixnum
         integer)
    :clause-2)
```



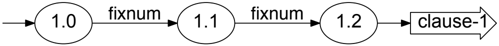

```
  ((:cat (or string
             fixnum)
         number)
    :clause-3))
```



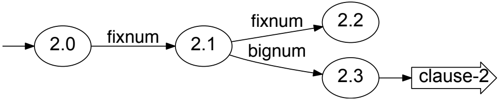
We can do better.

DFAs for disjoined clause-1, clause-2, and clause-3

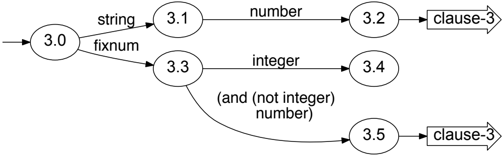
```
(rte-case expression
  ((:cat fixnum
         fixnum)
   :clause-1)
```




```
  ((and (:cat fixnum
              integer)
        (:not ...T1...))
   :clause-2)
```

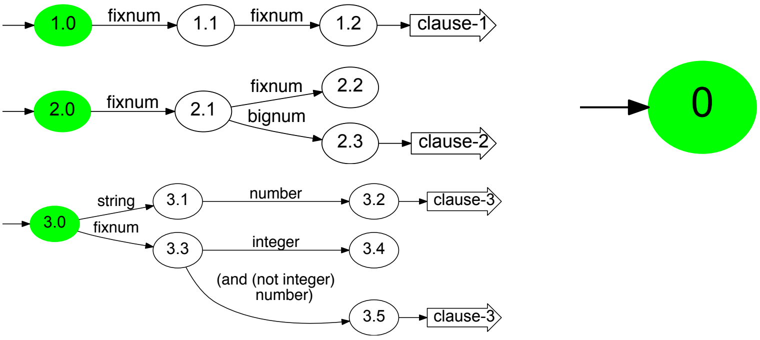



```
  ((:and (:cat (or string
                   fixnum)
               number)
         (:not ...T1...)
         (:not ...T2...))
   :clause-3))
```



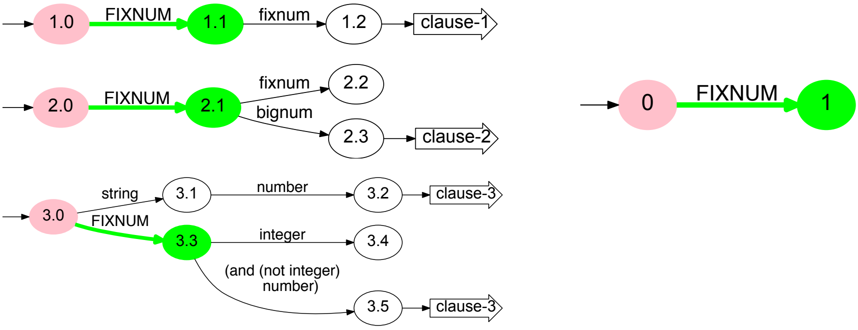
Calculating synchronized cross product

We can *merge* the three disjoint automata into one single automata.
Worst-case run-time is divided by 3.



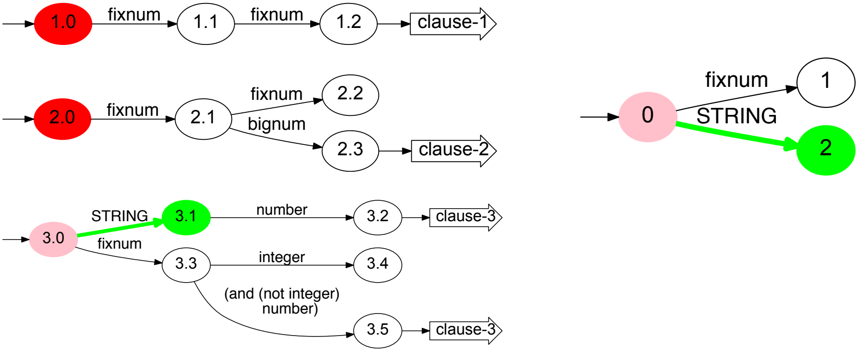
CXP: after `fixnum`

Easy, because `fixnum` transition is found on each input DFA.



CXP: after `string`

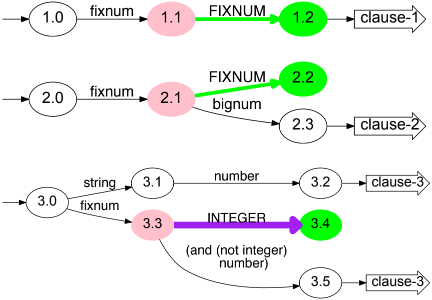
Easy, because `string` and `fixnum` are disjoint transitions of state 3.0.

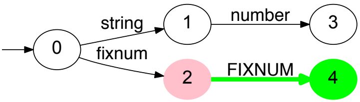


CXP: after `fixnum fixnum`

Challenging, because `fixnum` is not found on DFA 3.

`(subtypep fixnum integer)` ?



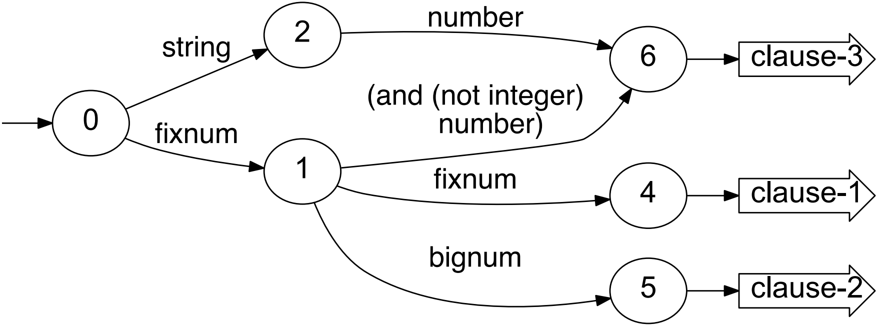


Challenging because `subtypep` might return `nil, nil`.

Consequence of `subtypep` returning `nil,nil`

Every time `subtypep` returns `nil,nil` the risk is that the remaining automata size doubles.

DFA representing synchronized-cross-product of `rte-case`



Short Demo

HyperSpec entry for DEFMETHOD

Macro **DEFMETHOD**

Syntax:

defmethod *function-name {method-qualifier}* specialized-lambda-list [[declaration* | documentation]] form**

=> *new-method*

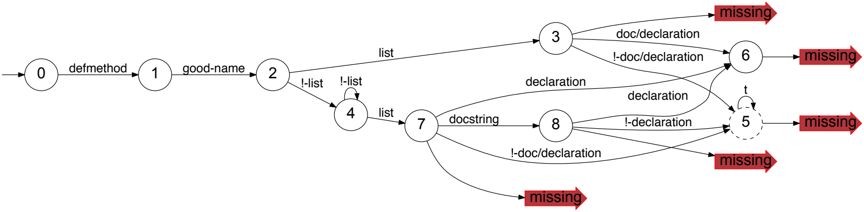
function-name::= {*symbol* | (setf *symbol*)}

method-qualifier::= *non-list*

```
specialized-lambda-list::= ({var | (var parameter-specializer-name)}*
                            [&optional {var | (var [initform [supplied-p-parameter] ])}*]
                            [&rest var]
                            [&key{var | ({var | (keywordvar)} [initform [supplied-p-parameter] ])}*
                                 [&allow-other-keys] ]
                            [&aux {var | (var [initform] )}*] )
parameter-specializer-name::= symbol | (eql eql-specializer-form)
```

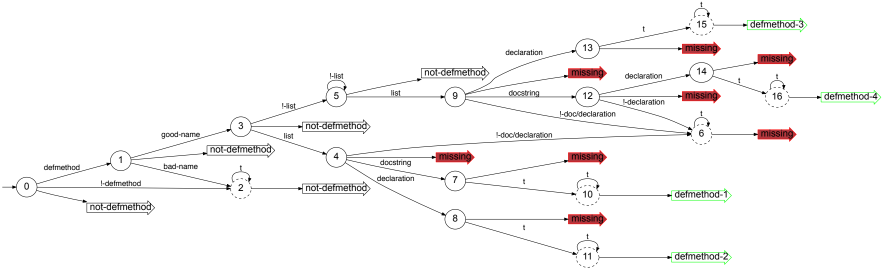
Short Demo

All the valid `defmethod` forms which are unaccounted for.



Short Demo

All the remaining ways a valid `defmethod` form can appear, some accounted for in the `destructuring-case` and some accounted for.



Summary

Our implmentation of an N-clause `destructuring-case` reduces the number of traversals of the sequence in question from $N+1$ to 2, once for descrimination, and one for binding.

The code is available from `quicklisp` via package `:rte`.

Perspectives

Lots more to be done: benchmarking, connection to method dispatch...

There are two CloJure libraries `seqspec` and `spec` which seem very related. According to the author of `seqspec`, `seqspec` does not optimize using finite automata because of some annoying limitations of the JVM.

Thanks to Didier Verna for begin my PhD advisor for the past 3 years. Also thanks to Robert Strandh, Pascal Costanza, and Christophe Rhodes for serving on my PhD defense committee.

Questions/Answers

Questions?