Language Features To Support Macro Programming

https://clojure.org/reference/reader

Ken Q Pu, Faculty of Science, Ontario Tech University

Code generation is hard

Code generation requires different types of data:

- List
- Built-in keywords
- Symbols

Their default constructors are unreadable, and can make the macro code difficult to read and maintain.

Suppose we want a macro to generate the following Clojure code:

```
(let [x 10] (inc x))
```

This is the macro required:

```
(defmacro inc-10 []
 (list 'let (vector 'x 10)
  (list 'inc 'x)))
```

It works, but not maintainable nor scalable.

```
(inc-10)
⇒ 11
```

Clojure's templating features

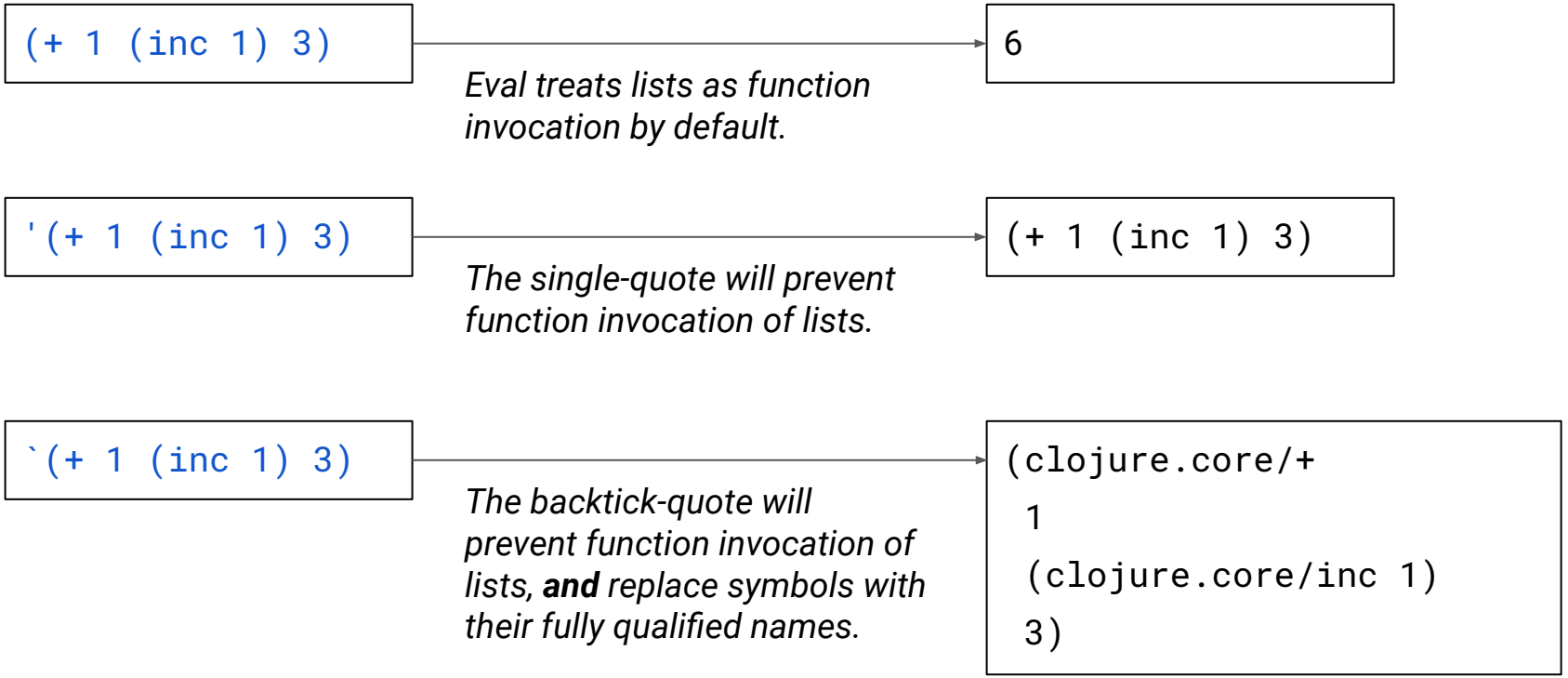
Clojure has built-in features to help with code generation.

- Syntax quote
- Substitution
- Slice substitution

These features are part of the Clojure language, not limited to macro programming.

However, they are designed specifically for developing macros easier.

Syntax Quote



Why do we need fully qualified names?

```
(let [inc (fn [x] (- x 1))
  (inc 10))

⇒ 9
```

We can redefine functions in a scope, which makes the behaviour of any generated code nondeterministic.

```
(let [inc (fn [x] (- x 1))
  (clojure.core/inc 10))

⇒ 11
```

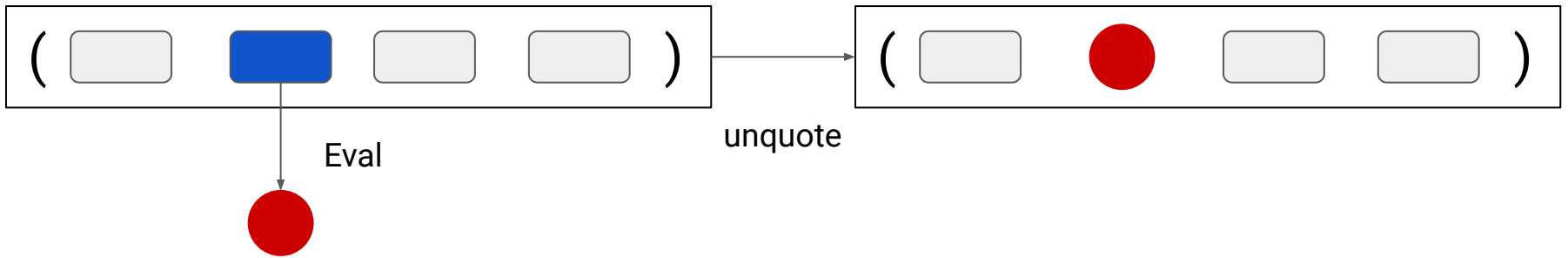
*The fully qualified name refers to the **original** version of the function.*

If we always use fully qualified names in the generated code, we can safely assume the version of the function we are using.

Computation inside quoted forms: substitutions

We need computation inside quoted forms.

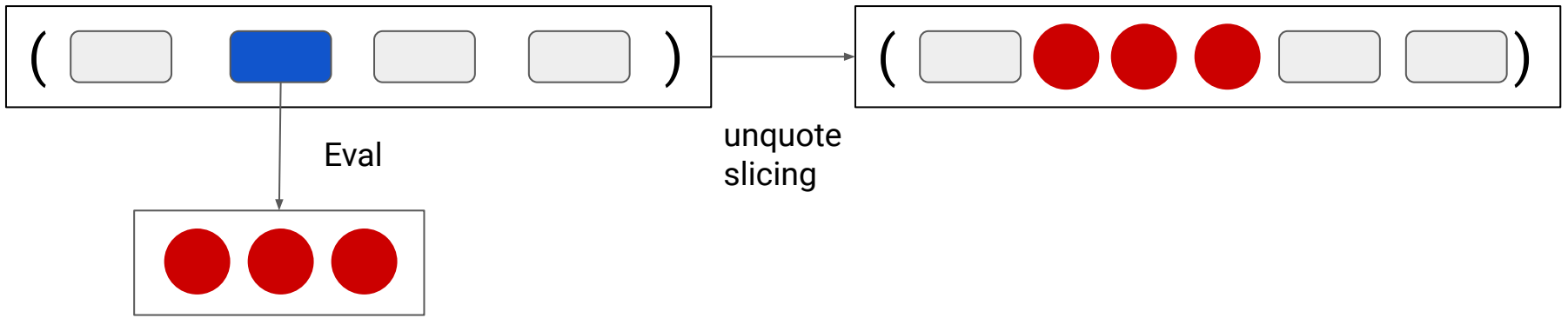
- Single value unquote
- unquote slicing



Computation inside quoted forms: unquotes

We need computation inside quoted forms.

- Single value unquote
- unquote-slicing



Ken Q Pu, Faculty of Science, Ontario Tech University

Single value substitution with unquote

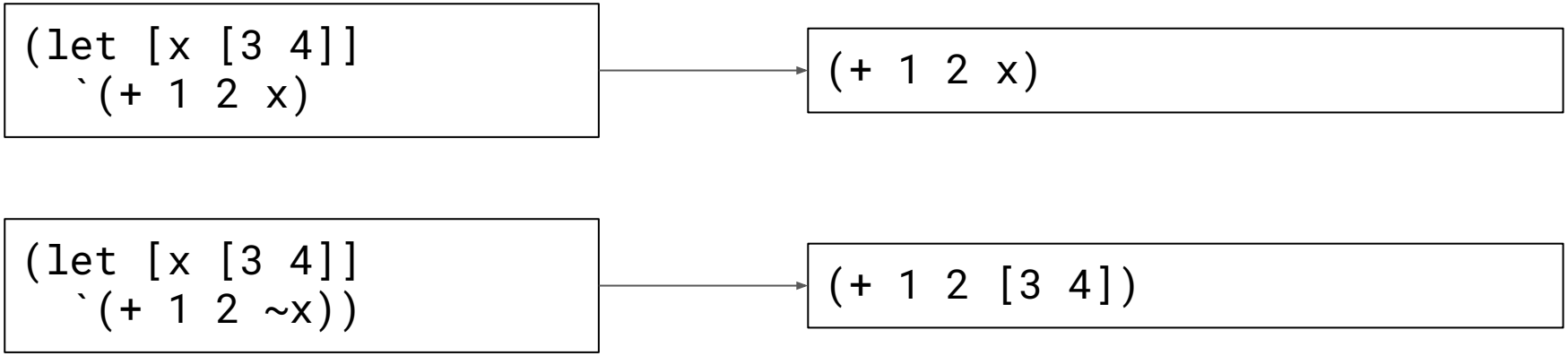
```
(let [x 2]
  `(+ 1 2 (inc x)))
```

```
(+ 1 2 (inc x))
```

```
(let [x 2]
  `(+ 1 2 ~(inc x)))
```

```
(+ 1 2 3)
```

Single value substitution with unquote



We really want to generate:

(+ 1 2 3 4)

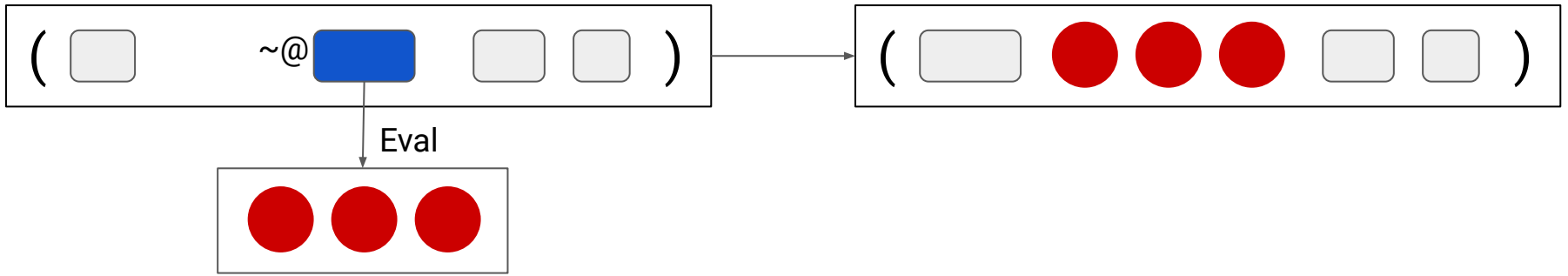
For this, we need slice substitution.

Slice substitution with unquote-slicing

```
(let [x [3 4]]
  `(+ 1 2 ~@x)
```

```
(+ 1 2 3 4)
```

~@ must be inside a quoted form. It will evaluate the next form which evaluates to a sequence to be inserted as a slice in place of ~@.



Ken Q Pu, Faculty of Science, Ontario Tech University

Dynamic symbol generation

Macros may need to generate code containing symbols that are unique to the generated code.

These symbols are *fresh* to the generated code.

Unsafe symbol in generated code because *name* may be used by runtime code in the inner scope.

```
(let [name "Ken Pu"]
  (... name ...))
```

Symbol is safe because the suffix makes sure that the name is unique.

```
(let [name_85930 "Ken Pu"]
  (... name_85930 ...))
```

Dynamic symbol generation

Clojure provides a function:

`(gensym prefix)`

which generates fresh symbols in the form of *prefix<random>*

Clojure also offers a convenient way to generate consistent fresh variables **inside** quoted forms.

#<prefix>

```
(let [name-var (gensym "name")]
  `(let [~name-var "Ken Pu"]
     (... ~name-var ...)))
```

```
`(let [name# "Ken Pu"]
   (... name# ...))
```

Ken Q Pu, Faculty of Science, Ontario Tech University

Summary

Generating code is helped by:

- Templating using quoted forms.
- Substitution can be done using
  - Unquotes
  - Unquote-slicing
- Dynamic generating of symbol names