Macros
Announcements
Macros
Macros Perform Code Transformations
Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation.
Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation.

Macros exist in many languages, but are easiest to define correctly in a language like Lisp.
Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation.

Macros exist in many languages, but are easiest to define correctly in a language like Lisp.

Scheme has a `define-macro` special form that defines a source code transformation.
Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation.

Macros exist in many languages, but are easiest to define correctly in a language like Lisp. Scheme has a `define-macro` special form that defines a source code transformation:

```
(define-macro (twice expr)
  (list 'begin expr expr))
```
Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation.

Macros exist in many languages, but are easiest to define correctly in a language like Lisp. Scheme has a `define-macro` special form that defines a source code transformation.

```scheme
(define-macro (twice expr)
  (list 'begin expr expr))

(twice (print 2))
```
Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation

Macros exist in many languages, but are easiest to define correctly in a language like Lisp

Scheme has a `define-macro` special form that defines a source code transformation

```
(define-macro (twice expr)
  (list 'begin expr expr))
```

```
(twice (print 2))
```

Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation.

Macros exist in many languages, but are easiest to define correctly in a language like Lisp. Scheme has a `define-macro` special form that defines a source code transformation.

```scheme
(define-macro (twice expr)
  (list 'begin expr expr))
```

```scheme
(twice (print 2)) → (print 2) (print 2)
```

Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation.

Macros exist in many languages, but are easiest to define correctly in a language like Lisp.

Scheme has a `define-macro` special form that defines a source code transformation:

```
(define-macro (twice expr)
  (list 'begin expr expr))
```

```
(twice (print 2))  ; (begin (print 2) (print 2))
```

```
2
2
```
Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation.

Macros exist in many languages, but are easiest to define correctly in a language like Lisp.

Scheme has a `define-macro` special form that defines a source code transformation:

```
(define-macro (twice expr)
  (list 'begin expr expr))
```

```
(twice (print 2))  ➔ (begin (print 2) (print 2))
```

Evaluation procedure of a macro call expression:
Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation.

Macros exist in many languages, but are easiest to define correctly in a language like Lisp. Scheme has a `define-macro` special form that defines a source code transformation:

\[
(\text{define-macro} \ (\text{twice} \ \text{expr}) \ (\text{list} \ \text{\'begin} \ \text{expr} \ \text{expr}))
\]

\[
\Rightarrow \ (\text{twice} \ (\text{print} \ 2)) \Rightarrow (\text{begin} \ (\text{print} \ 2) \ (\text{print} \ 2))
\]

Evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro.
Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation.

Macros exist in many languages, but are easiest to define correctly in a language like Lisp.

Scheme has a `define-macro` special form that defines a source code transformation:

```
(define-macro (twice expr)
  (list 'begin expr expr))

> (twice (print 2))
2
2
```

Evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro.
- Call the macro procedure on the operand expressions `without evaluating them first`.
Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation.

Macros exist in many languages, but are easiest to define correctly in a language like Lisp.

Scheme has a `define-macro` special form that defines a source code transformation:

```
(define-macro (twice expr)
  (list 'begin expr expr))
```

Evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro
- Call the macro procedure on the operand expressions without evaluating them first
- Evaluate the expression returned from the macro procedure

Example:

```
(define-macro (twice expr)
  (list 'begin expr expr))

> (twice (print 2))
 2
 2
```

```
(begin (print 2) (print 2))
```
Macros Perform Code Transformations

A macro is an operation performed on the source code of a program before evaluation.

Macros exist in many languages, but are easiest to define correctly in a language like Lisp.

Scheme has a `define-macro` special form that defines a source code transformation

```
(define-macro (twice expr)
  (list 'begin expr expr))
```

```
(twice (print 2))
```

Evaluation procedure of a macro call expression:

- Evaluate the operator sub-expression, which evaluates to a macro.
- Call the macro procedure on the operand expressions *without evaluating them first*.
- Evaluate the expression returned from the macro procedure.

(Demo)
For Macro
Discussion Question

Define a macro that evaluates an expression for each value in a sequence
Discussion Question

Define a macro that evaluates an expression for each value in a sequence

scm> (for x (2 3 4 5) (* x x))
(4 9 16 25)
Discussion Question

Define a macro that evaluates an expression for each value in a sequence

```
scm> (map (lambda (x) (* x x)) (2 3 4 5))
```

```
scm> (for x (2 3 4 5) (* x x))
(4 9 16 25)
```
Discussion Question

Define a macro that evaluates an expression for each value in a sequence

```
(scm> (map (lambda (x) (* x x)) (2 3 4 5))
(4 9 16 25))
```

```
(scm> (for x (2 3 4 5) (* x x))
(4 9 16 25))
```
Define a macro that evaluates an expression for each value in a sequence

```scheme
(scm> (map (lambda (x) (* x x)) (2 3 4 5))
(4 9 16 25))
```

```scheme
(define-macro (for sym vals expr)
  (list 'map sym vals expr))
```

```scheme
(scm> (for x (2 3 4 5) (* x x))
(4 9 16 25))
```
Discussion Question

Define a macro that evaluates an expression for each value in a sequence

```
scm> (map (lambda (x) (* x x)) (2 3 4 5))
(4 9 16 25)
```

```
(define-macro (for sym vals expr)
  (list 'map (list 'lambda (list sym) expr) vals))
```

```
scm> (for x (2 3 4 5) (* x x))
(4 9 16 25)
```
Discussion Question

Define a macro that evaluates an expression for each value in a sequence

\[
\text{scm> (map (lambda (x) (* x x)) (2 3 4 5))}  \\
(4 9 16 25)
\]

\[
\text{(define-macro (for sym vals expr)}  \\
\text{ (list 'map (list 'lambda (list sym) expr) vals))}
\]

\[
\text{scm> (for x (2 3 4 5) (* x x))}  \\
(4 9 16 25)  \\
\text{(Demo)}
\]
Implementing Macros

(Demo)