Scheme Review

Jiacai Cui

PASCAL Group @ Nanjing University

Content

- Basics
- Pairs and Lists
- Macros
- Streams
- Interpreters

Basics

Scheme Expressions

Scheme Program = Expressions

- Primitives:
  - *Self-evaluating*: numbers(`3 5.5 -10`), booleans(`#t #f`)
  - *Symbols*: names bound to values(`+ modulo list x foo hello-world`)
- Combinations: `(<operator> <operand1> <operand2> ...)`
  - *Call expression*
  - *Special form expression*

Call Expressions

```
(<operator> <operand1> <operand2> ...)
```

A call expression applies a procedure to some arguments

1. **Evaluate** the **operator** to get a **procedure**
2. **Evaluate** all **operands** from left to right to get the **arguments**
3. **Apply** the **procedure** to the **arguments**
   a) Create a local frame
   b) Bind arguments to parameters in the local frame
   c) Evaluate the body expression in the local frame and return its value

Special Forms - Define

```
(define <name> <expression>)
```

1. **Evaluate** the given **expression** to get a value
2. **Bind** the value to the given **name** in the current frame
3. **Return** the name as a **symbol**

Special Forms - Lambda

```
(lambda (<parameter1> <parameter2> ...) <body>)
```

1. **Create** a **procedure** with the given **parameters** and **body** expression
2. **Return** the procedure

```
(define (f <parameter1> <parameter2> ...) <body>)
```

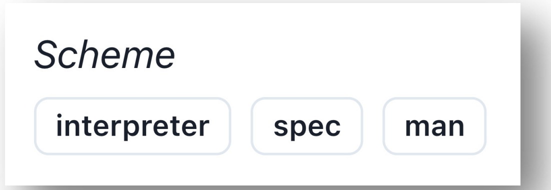
is short for

```
(define f (lambda (<parameter1> <parameter2> ...) <body>))
```

Special Forms - If

```
(if <predicate> <if-true> <if-false>)
```

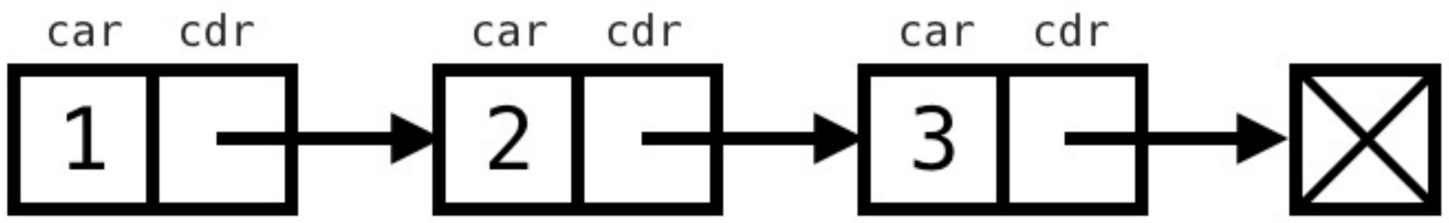
1. Evaluate the predicate
2. If the predicate isn't `#f`, evaluate `<if-true>` and return the value
   - `#f` is the only falsy value in Scheme
3. Otherwise, evaluate `<if-false>` and return the value



Pairs and Lists

Pairs

- Pairs are created using the **cons** expression in scheme.
- **car** selects the first elements in a pair.
- **cdr** selects the second elements in a pair.
- The second element of a pair must be another pair, or **nil** (empty).



Quotation

`'<expression>` short for `(quote <expression>)`

Quotation is a **special form** to indicate that **the expression itself is the value**.

- Be used to refer to **symbols** directly.
- Be applied to **combinations** to form **lists**.

Tail Recursion

- An expression is in a tail context only if it is **the last thing evaluated in every possible scenario** (no other action is performed afterwards).

```
  (fact 5)
= (* 5 (fact 4))
= (* 5 (* 4 (fact 3)))
= (* 5 (* 4 (* 3 (fact 2))))
= (* 5 (* 4 (* 3 (* 2 (fact 1)))))
= (* 5 (* 4 (* 3 (* 2 (* 1 (fact 0))))))
= (* 5 (* 4 (* 3 (* 2 (* 1 1)))))
= (* 5 (* 4 (* 3 (* 2 1))))
= (* 5 (* 4 (* 3 2)))
= (* 5 (* 4 6))
= (* 5 24)
= 120
```

V.S.

```
  (fact-optimized 5 1)

= (fact-optimized 4 5)

= (fact-optimized 3 20)

= (fact-optimized 2 60)

= (fact-optimized 1 120)

= (fact-optimized 0 120)

= 120
```

Macros

Expression as Data

- Expressions are either **primitives** or **combinations** (i.e. **lists**), which means they are also a kind of data!
- Quoting helps you get the **unevaluated expression as a kind of data**.
  - Quoting a **self-evaluating** primitive gets you **itself**.
  - Quoting a **name** gets you a **symbol** of that name.
  - Quoting a **combination** gets you a **list** of that combination.
- Calling `eval` on an unevaluated expression will evaluate that expression to get a value.

Macros

```
(define-macro (<name> <parameter1> <parameter2> ...) <body>)
```

Macros **take in and return expressions**, which are **then evaluated** in place of the call to the macro.

1. Evaluate the operator sub-expression, which evaluates to a macro procedure.
2. **Apply** the macro procedure to **the operand expressions without evaluating** them first.
3. **Evaluate** the **expression returned by the macro procedure** in the frame the macro was called in.

Quasiquotation

`` `<expression> `` is short for `(quasiquote <expression>)`

`,<expression>` is short for `(unquote <expression>)`

- Quasiquotation helps you write unevaluated expressions more easily.
- `quasiquote` **overall quote** an expression with **partially** some sub-expressions **unquoted** (i.e. evaluated) by `unquote`.

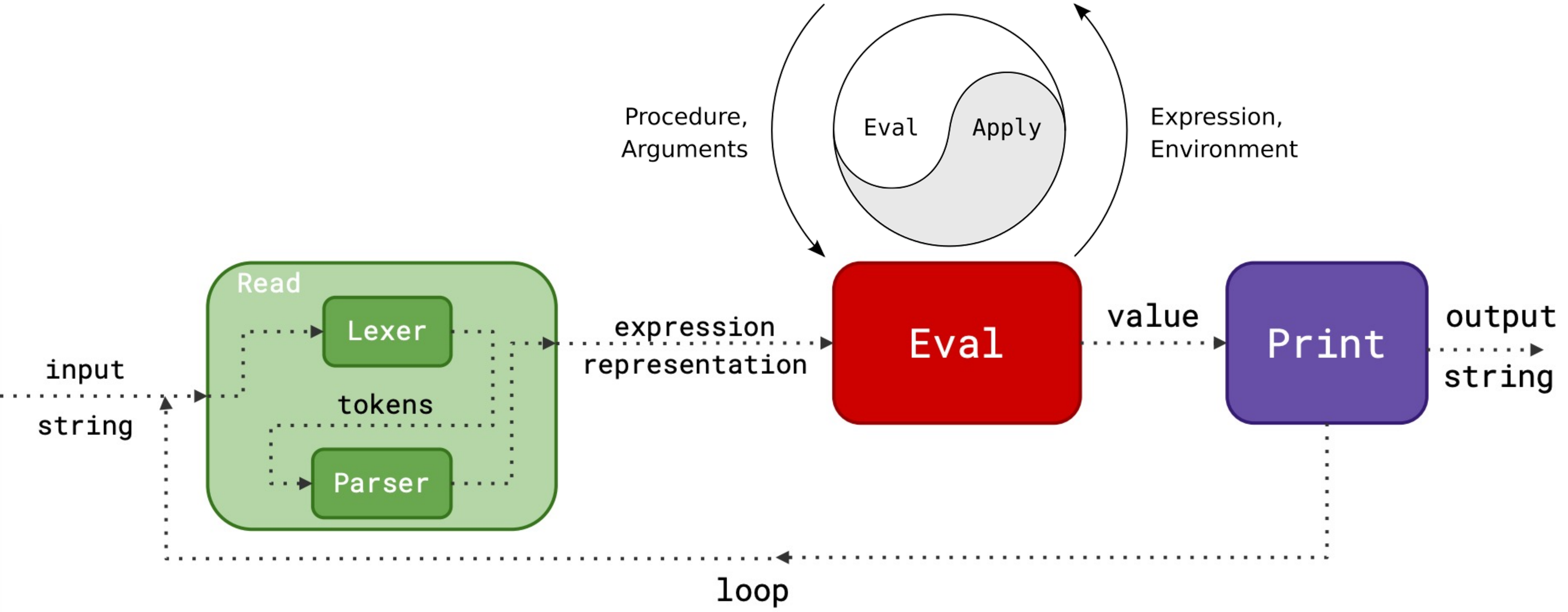
Streams

Stream - Lazy Evaluated List

- `nil` is the empty stream.
- `cons-stream` constructs a stream (i.e. pair) containing **the value of the first operand** and **a promise to evaluate the second operand**.
- `car` returns the first element of the stream (i.e. pair).
- `cdr-stream` **evaluates** and **returns** the rest of stream (i.e. pair).

Interpreters

Read-Evaluate-Print Loop (REPL)



Thanks!