Interpreters

Class outline:

- Interpreting Scheme
- Special forms
- Logical forms
- Quotation
- Lambda expressions
- Define expressions

Interpreting Scheme

The Structure of an Interpreter

Cover of "Structure and Interpretation of Computer Programs", JS adaptation

The Structure of an Interpreter

Eval

Base cases:

Recursive calls:

Apply

Base cases:

The Structure of an Interpreter

Eval

Base cases:

- Primitive values (numbers)

Recursive calls:

Apply

Base cases:

The Structure of an Interpreter

Eval

Base cases:

- Primitive values (numbers)

Recursive calls:

- Eval(operator, operands) of call expressions

Apply

Base cases:

The Structure of an Interpreter

Eval

Base cases:

- Primitive values (numbers)

Recursive calls:

- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)

Apply

Base cases:

The Structure of an Interpreter

Eval

Base cases:

- Primitive values (numbers)

Recursive calls:

- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)

Apply

Base cases:

- Built-in primitive procedures

The Structure of an Interpreter

Eval

Base cases:

- Primitive values (numbers)
- Look up values bound to symbols

Recursive calls:

- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)

Apply

Base cases:

- Built-in primitive procedures

The Structure of an Interpreter

Eval

Base cases:

- Primitive values (numbers)
- Look up values bound to symbols

Recursive calls:

- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)
- Eval(sub-expressions) of special forms

Apply

Base cases:

- Built-in primitive procedures

The Structure of an Interpreter

Eval

Base cases:

- Primitive values (numbers)
- Look up values bound to symbols

Recursive calls:

- Eval(operator, operands) of call expressions
- Apply(procedure, arguments)
- Eval(sub-expressions) of special forms

⤹ ⤴

Apply

Base cases:

- Built-in primitive procedures

Recursive calls:

- Eval(body) of user-defined procedures

Special forms

Scheme evaluation

The `scheme_eval` function chooses behavior based on expression form:

- Symbols are looked up in the current environment
- Self-evaluating expressions (booleans, numbers, nil) are returned as values
- All other legal expressions are represented as Scheme lists, called combinations
  `(<operator> <operand 0> ... <operand k>)`

Evaluating combinations

The special forms can all be identified by the first element:

```
(if <predicate> <consequent> <alternative>)
(lambda (<formal-parameters>) <body>)
(define <name> <expression>)
```

Any combination that is not a known special form must be a call expression.

```
(define (demo s)
    (if (null? s)
        '(3)
        (cons (car s) (demo (cdr s)))))
(demo (list 1 2))
```

Evaluating combinations

The special forms can all be identified by the first element:

```
(if <predicate> <consequent> <alternative>)
(lambda (<formal-parameters>) <body>)
(define <name> <expression>)
```

Any combination that is not a known special form must be a call expression.

```
(define (demo s)
    (if (null? s)
        '(3)
        (cons (car s) (demo (cdr s)))))   ; Special!
(demo (list 1 2))
```

Evaluating combinations

The special forms can all be identified by the first element:

```
(if <predicate> <consequent> <alternative>)
(lambda (<formal-parameters>) <body>)
(define <name> <expression>)
```

Any combination that is not a known special form must be a call expression.

```
(define (demo s)
    (if (null? s)
        '(3)
        (cons (car s) (demo (cdr s)))))   ; Special!
(demo (list 1 2))                         ; Call expression!
```

Logical special forms

Logical forms are special forms that may only evaluate some sub-expressions.

- If expression: `(if <predicate> <consequent> <alternative>)`
- And and or: `(and <e1> ... <en>)`, `(or <e1> ... <en>)`
- Cond expression: `(cond (<p1> <e1>) ... (<pn> <en>) (else <e>))`

Logical special forms

Logical forms are special forms that may only evaluate some sub-expressions.

- If expression: `(if <predicate> <consequent> <alternative>)`
- And and or: `(and <e1> ... <en>)`, `(or <e1> ... <en>)`
- Cond expression: `(cond (<p1> <e1>) ... (<pn> <en>) (else <e>))`

The value of an `if` expression is the value of a sub-expression:

- Evaluate the predicate
- Choose a sub-expression: <consequent> or <alternative>
- Evaluate that sub-expression to get the value of the whole expression

Quotation

`'<expression>` is shorthand for `(quote <expression>)`
`'(1 2)` is equivalent to `(quote (1 2))`

The `scheme_read` parser converts `'` to a combination that starts with `quote`.

Quotation

`'<expression>` is shorthand for `(quote <expression>)`
`'(1 2)` is equivalent to `(quote (1 2))`

The `scheme_read` parser converts `'` to a combination that starts with `quote`.

The `quote` special form evaluates to the quoted expression, which is not evaluated.

`(quote (+ 1 2))`
evaluates to the three-element Scheme list
`(+ 1 2)`

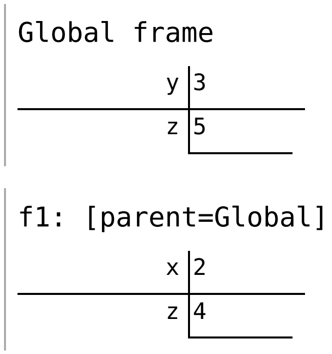
Symbols & Functions

Frames

A frame represents an environment by having a parent frame.

A frame is an instance of a `Frame` object, which has `lookup` and `define` methods.

In this interpreter, frames do **not** hold return values.



Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

`(define <name> <expression>)`

- Evaluate the `<expression>`
- Bind `<name>` to its value in the current frame

```
(define x (+ 1 2))
```

Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

`(define <name> <expression>)`

- Evaluate the `<expression>`
- Bind `<name>` to its value in the current frame

```
(define x (+ 1 2))
```

Procedure definition is shorthand of define with a lambda expression.

`(define (<name> <formal parameters>) <body>)`
`(define <name> (lambda (<formal parameters>) <body>))`

Lambda Expressions

Lambda expressions evaluate to user-defined procedures

`(lambda (<formal-parameters>) <body> ... )`

```
class LambdaProcedure:
    def __init__(self, formals, body, env):
        self.formals = formals  # A scheme list of sy
        self.body = body        # A scheme list of ex
        self.env = env          # A Frame instance
```

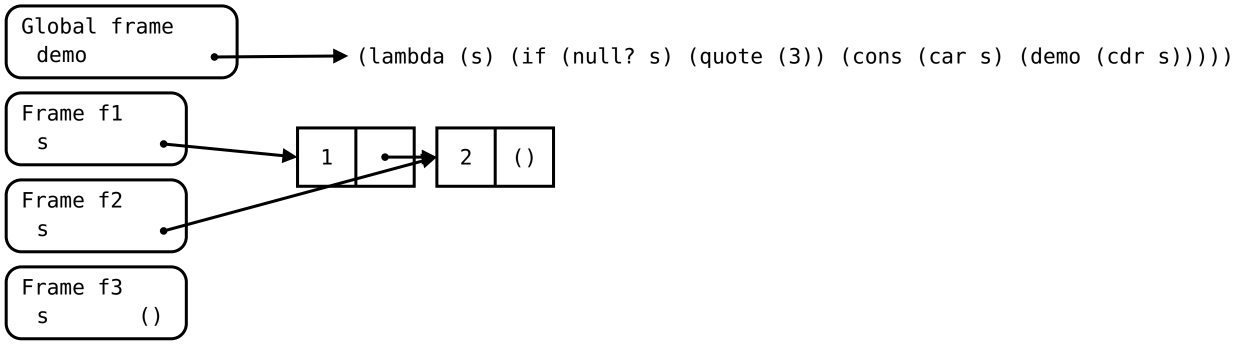
```
(lambda (x y) (* x y))
```

Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the env attribute of the procedure.

Evaluate the body of the procedure in the environment that starts with this new frame.

```
(define (demo s)
    (if (null? s)
        '(3)
        (cons (car s) (demo (cdr s)))))
(demo (list 1 2))
```



Global frame
demo
(lambda (s) (if (null? s) (quote (3)) (cons (car s) (demo (cdr s)))))
Frame f1
s
Frame f2
s
Frame f3
s
()
1
2
()