Structure and Interpretation of Computer Programs

COMP200
BUILDING UP A LANGUAGE

1. eval/apply core
TODAY
A Broader Look

• Scheme Evaluator – A Grand Tour
• Scheme Evaluator – A Grand Tour

• Techniques for language design:
  • Interpretation: eval/apply
  • Semantics vs. syntax
  • Syntactic transformations
TODAY
A Broader Look

• Scheme Evaluator – A Grand Tour

• Techniques for language design:
  • Interpretation: eval/apply
  • Semantics vs. syntax
  • Syntactic transformations

• Beyond Scheme – designing language variants
  • Lexical scoping vs. Dynamic scoping
BUILDING UP A LANGUAGE

1. eval/apply core
2. syntax procedures
3. environment manipulation
4. primitives and initial env.
5. read-eval-print loop
THE CORE EVALUATOR

1. eval/apply core

- eval
- apply
- exp & env
- proc & args
THE CORE EVALUATOR

1. eval/apply core

- Core evaluator
  - eval: dispatch on expression type
  - apply: eval args then apply operator
BASIC SEMANTICS

Code
BASIC SEMANTICS
m-eval & m-apply

• primitive expressions
  • self-evaluating, quoted

• variables and the environment
  • variable definition, lookup, and assignment

• conditionals
  • if, cond

• procedure application

• sequences
  • begin
• The procedure body is a sequence of one or more expressions:

```
(define (foo x)
  (do-something (+ x 1))
  (* x 5))
```
• The procedure body is a sequence of one or more expressions:

\[
\text{(define (foo x)}
\text{(do-something (+ x 1))}
\text{(* x 5))}
\]

• In m-apply, we eval-sequence the procedure body.
MORE

Code
SYNTACTIC ABSTRACTION

2.

• Semantics
  • What the language *means*
  • Model of computation
SYNTACTIC ABSTRACTION

2.

- Syntax procedures

• Semantics
  • What the language *means*
  • Model of computation

• Syntax
  • Particulars of writing expressions
  • E.g. how to signal different expressions

2.

- Syntax procedures

- Eval/apply
SYNTACTIC ABSTRACTION

• Semantics
  • What the language *means*
  • Model of computation

• Syntax
  • Particulars of writing expressions
  • E.g. how to signal different expressions

• Separation of syntax and semantics:
  • allows one to easily alter syntax
BASIC SYNTAX

- Routines to detect expressions
  
  (define (if? exp) (tagged-list? exp 'if))
  (define (lambda? exp) (tagged-list? exp 'lambda))
  (define (application? exp) (pair? exp))
BASIC SYNTAX

• Routines to detect expressions
  (define (if? exp) (tagged-list? exp 'if))
  (define (lambda? exp) (tagged-list? exp 'lambda))
  (define (application? exp) (pair? exp))

• Routines to get information out of expressions
  (define (operator app) (car app))
  (define (operands app) (cdr app))
BASIC SYNTAX

• Routines to detect expressions
  (define (if? exp) (tagged-list? exp 'if))
  (define (lambda? exp) (tagged-list? exp 'lambda))
  (define (application? exp) (pair? exp))

• Routines to get information out of expressions
  (define (operator app) (car app))
  (define (operands app) (cdr app))

• Routines to manipulate expressions
  (define (no-operands? args) (null? args))
  (define (first-operand args) (car args))
  (define (rest-operands args) (cdr args))
CHANGING SYNTAX

Example
CHANGING SYNTAX

Example

• Suppose you wanted a "verbose" application syntax:

  (CALL <proc> ARGS <arg1> <arg2> ...)
• Suppose you wanted a "verbose" application syntax:

    (CALL <proc> ARGS <arg1> <arg2> …)

• Changes – only in the syntax routines!

  (define (application? exp) (tagged-list? 'CALL))
  (define (operator app) (cadr app))
  (define (operands app) (cdddr app))
SYNTACTIC SUGAR

Idea

• Implement a simple fundamental "core" in the evaluator
• Easy way to add alternative/convenient syntax?
SYNTACTIC SUGAR

Example

(let ((<name1> <val1>)
     (<name2> <val2>))
  <body>)
SYNTACTIC SUGAR

Example

• "let" as sugared procedure application:

(let ((<name1> <val1>)
     (<name2> <val2>))
  <body>)

  ((lambda (<name1> <name2>) <body>)
   <val1> <val2>))
DETECT AND TRANSFORM THE ALTERNATIVE SYNTAX

Code
(let ((x 23)
     (y 15))
  (do-something x y))
(let ((x 23)
     (y 15))
  (do-something x y))
(let ((x 23)
     (y 15))
  (do-something x y))
LET SYNTAX TRANSFORMATION

Details

Diagram showing the transformation process with nodes and edges labeled with variables and values like 'let', 'x', '23', 'Y', '15', and 'do-something x Y'.
LET SYNTAX TRANSFORMATION

Details

```
x 23  y 15
```

```
x -> let -> y
```

```
let -> do-something x Y
```

```
x -> y
```

```
x 23  y 15
```

```
x -> let -> y
```

```
let -> do-something x Y
```

```
x -> y
```
LET SYNTAX TRANSFORMATION

Details

```
x -> 23 -> let
    "do-something" x
Y -> 15
```

```
23 -> 15
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```

```
x -> y
```
LET SYNTAX TRANSFORMATION

Details

lambda

x

23

15

do-something x y

let

y

x

23
NAMED PROCEDURES
Syntax vs. Semantics

(define (foo <parm>) <body>)
NAMED PROCEDURES

Syntax vs. Semantics

(define (foo <parm>) <body>)

• Semantic implementation – just another define:

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
    (m-eval (definition-value exp) env)
    env))
NAMED PROCEDURES
Syntax vs. Semantics

(define (foo <parm>) <body>)

• Semantic implementation – just another define:

(define (eval-definition exp env)
    (define-variable! (definition-variable exp)
        (m-eval (definition-value exp) env)
        env))

• Syntactic transformation:

(define (definition-value exp)
    (if (symbol? (cadr exp))
        (caddr exp)
        (make-lambda (cdadr exp) ;formal params
            (cddr exp)))) ;body
ENVIRONMENT

3. environment manipulation
ENVIROMENT

• Abstractly – in our environment diagrams:

E2

E1

x: 10
plus: (procedure ...)

3. environment manipulation
ENVIRONMENT

3. environment manipulation

- Abstractly – in our environment diagrams:

  x: 10
  plus: (procedure ...)

- Concretely – our implementation:

  E2 → E1 → enclosing env.
  frame
ENVIRONMENT

3. Environment manipulation

- Abstractly – in our environment diagrams:
  - E1
    - x: 10
    - plus: (procedure ...)
  - E2
    - E2
    - plus: (procedure ...)

- Concretely – our implementation:
  - E2
    - list of variables
    - list of values
    - frame
    - enclosing env.
    - x
    - plus
    - 10
    - procedure
(extend-environment
  '(x y) (list 4 5) E2)
(extend-environment
   '(% x y) (list 4 5) E2)

• Abstractly

\[
\begin{array}{l}
\text{x: 10} \\
\text{plus: (procedure ...)}
\end{array}
\]
• Abstractly

(extend-environment
 '(* x y) (list 4 5) E2)
(extend-environment '((x y) (list 4 5)) E2)

- **Abstractly**

  - \(x: 10\)
  - \(\text{plus: (procedure ...)}\)

- **Concretely**

  - \(x: 4\)
  - \(y: 5\)
Abstractly

\[(\text{extend-environment}'(\text{x y} \ (\text{list 4 5}) \ \text{E2}))\]

Concretely

\[
\begin{cases}
\text{x: 10} \\
\text{plus: (procedure ...)} \\
\text{E2}
\end{cases}
\]

\[
\begin{cases}
\text{x: 4} \\
\text{y: 5} \\
\text{E3}
\end{cases}
\]

\[
\begin{cases}
\text{list of variables} \\
\text{frame} \\
\text{list of values}
\end{cases}
\]

\[
\begin{cases}
\text{x} \\
\text{y} \\
\text{4} \\
\text{5}
\end{cases}
\]
SCANNING THE ENVIRONMENT

• Look for a variable in the environment
SCANNING THE ENVIRONMENT

• Look for a variable in the environment

• Look for a variable in a frame
  • loop through the list of vars and list of vals in parallel
  • detect if the variable is found in the frame
SCANNING THE ENVIRONMENT

• Look for a variable in the **environment**

• Look for a variable in a **frame**
  • loop through the **list of vars** and **list of vals** in parallel
  • detect if the variable is found in the **frame**

• If not found in **frame** (out of **variables** in the **frame**), look in enclosing **environment**
SCANNING THE ENVIRONMENT

Code
THE INITIAL (GLOBAL) ENVIRONMENT

4. primitives and initial env.
4. setup-environment

(define (setup-environment)
  (let ((initial-env
    (extend-environment (primitive-procedure-names)
      (primitive-procedure-objects)
      the-empty-environment)))
    (define-variable! 'true #T initial-env)
    (define-variable! 'false #F initial-env)
    initial-env))
THE INITIAL (GLOBAL) ENVIRONMENT

4. primitives and initial env.

• setup-environment

  (define (setup-environment)
    (let ((initial-env
      (extend-environment (primitive-procedure-names)
        (primitive-procedure-objects)
        the-empty-environment)))
      (define-variable! 'true #T initial-env)
      (define-variable! 'false #F initial-env)
      initial-env))

• define initial variables we always want

• bind explicit set of "primitive procedures"
  • here: use underlying scheme
  • in other interpreters: assembly code, hardware, ....
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (m-eval input the-global-env)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
DIVING IN DEEPER

Lexical Scope
DIVING IN DEEPER
Lexical Scope

• How does our evaluator achieve lexical scoping?
  • environment chaining
  • procedures that capture their lexical environment
DIVING IN DEEPER

Lexical Scope

• How does our evaluator achieve lexical scoping?
  • environment chaining
  • procedures that capture their lexical environment

**make-procedure:**

• stores away the evaluation environment of `lambda`
• the "evaluation environment" is always the enclosing lexical scope
• why?
DIVING IN DEEPER
Lexical Scope

**make-procedure:**

- stores away the evaluation environment of `lambda`
- the "evaluation environment" is always the enclosing lexical scope
- why?
  - our semantic rules for procedure application!
  - "hang a new frame"
  - "bind parameters to actual args in new frame"
  - "evaluate body in this new environment"
(define (foo x y)
  (lambda (z) (+ x y z)))

GE

foo: p1

P1

p: x y
b: (λ (z))
  (+ x y z)
(define (foo x y)
  (lambda (z) (+ x y z)))

(define bar (foo 1 2))
(define (foo x y)
  (lambda (z) (+ x y z)))

(define (foo 1 2))
(define bar (foo 1 2))
(bar 3)
DYNAMIC SCOPING
An Alternative Model

• Dynamic scope:
  • Look up free variables in the caller's environment rather than the surrounding lexical environment.
DYNAMIC SCOPING
An Alternative Model

• Dynamic scope:
  • Look up free variables in the caller's environment rather than the surrounding lexical environment

(define (pooh x)
  (bear 20))

(define (bear y)
  (+ x y))

(pooh 9)  =>   29
(define (pooh x) (bear 20))

(define (bear y) (+ x y))

DYNAMIC SCOPE
Environment Diagram

GE

pooh: p1

bear: p2

P1

p: x
b: (bear 20)

P2

p: y
b: (+ x y)
(define (pooh x) (bear 20))

(define (bear y) (+ x y))

(pooh 9) => 29
(define (pooh x)
  (bear 20))

(define (bear y)
  (+ x y))

(pooh 9) => 29

E1
x: 9

E2
y: 20

(+ x y) |E2
=> 29
A DYNAMIC SCHEME

Code
SUMMARY

• Scheme Evaluator – Know it Inside & Out

• Techniques for language design:
  • Interpretation: eval/apply
  • Semantics vs. syntax
  • Syntactic transformations

• Able to design new language variants!
  • Lexical scoping vs. Dynamic scoping