Structure and Interpretation of Computer Programs

COMP200
COURSE INFORMATION

Logistics

• Syllabus and other handouts

• Course webpage
  https://sites.google.com/a/ku.edu.tr/comp200/fall2019

• Problem sets
  http://etutor.ku.edu.tr/comp200/tutor.cgi

• Email TAs and me!
COURSE INFORMATION

Grades

• 2 midterms 40%
• Final exam 25%
• 5 programming projects 25%
• Problem sets 10%
• Participation
COURSE INFORMATION

Contact

• Lecturer: Fatma Guney
  Office Hours: ENG-203A
  Time: TBD

• TAs: Ozan Arkan Can, Berkay Furkan Onder, Ulas Sert
TODAY
Outline

• Procedures and procedural abstractions
• Using procedures to capture processes
LAST TIME

Summary

- Primitive data types
- Primitive procedures
- Means of combination
- Means of abstraction - names
• Procedures and procedural abstractions
• Using procedures to capture processes
LANGUAGE ELEMENTS

Abstractions

(lambda(x)(* x x))
LANGUAGE ELEMENTS

Abstractions

\( \text{keyword} \)

\((\text{lambda}(x)(\ast \ x \ x))\)
LANGUAGE ELEMENTS

Abstractions

(lambda(x)(* x x))
LANGUAGE ELEMENTS

Abstractions

(keyword) (parameters) (body)

(lambda(x)(* x x))
LANGUAGE ELEMENTS

Abstractions

(keyword) (parameters) (body)

(lambda(x)(* x x))
LANGUAGE ELEMENTS

Abstractions

(keyword parameters body)

(lambda(x)(* x x))

to process
LANGUAGE ELEMENTS

Abstractions

(keyword) parameters (body)

(lambda(x)(*)x x)

to process something
LANGUAGE ELEMENTS

Abstractions

\((\text{lambda}(x)(\ast \ x \ x))\)

- keyword
- parameters
- body

to process something multiply it by itself
LANGUAGE ELEMENTS
Abstractions

(lamdba(x)(* x x))
LANGUAGE ELEMENTS
Abstractions

\[(\text{lambda}(x)(\ast \ x \ x))\]

Special form: creates a procedure and returns it as value
LANGUAGE ELEMENTS
Abstractions

(lambda(x)(* x x))
LANGUAGE ELEMENTS

Abstractions

\[((\text{lambda}(x)(\ast \ x \ x)) \ 5)\]
Abstractions

$$(((\text{lambda}(x)(\ast\ x\ x))\ 5)$$

25
Anywhere you can use a procedure.

```
(((\(x\))(\(*\ \ x\ \ x\))\ 5)\n
25
```
Rules for Evaluation

• If **self-evaluating**, return the value.

• If a **name**, return the value associated with the name in the environment.

• If a **special form**, do something special.

• If a **combination**, then
  • **evaluate** all of the sub-expressions of the combination (in any order)
  • **apply** the operator to the values of the operands (arguments) and return the result
• If procedure is a *primitive procedure*, just do it.
SCHEME
Rules for Application

• If procedure is a **primitive procedure**, just do it.

• If procedure is a **compound procedure**, then:
  • *evaluate* the body of the procedure with each formal parameter replaced by the corresponding actual argument value.
((lambda(x)(* x x)) 5)

(* 5 5)
LANGUAGE ELEMENTS

Abstractions

\(((\text{lambda}(x)(\ast \ x \ x)) \ 5)\)

\((\ast \ 5 \ 5)\)

25
Abstractions

\[(\text{lambda}(x)(\ast\ x\ x))\]
(define square (lambda(x)(* x x)))
(define square (lambda(x)(* x x)))

> (square 5)
25
(define square (lambda(x)(* x x)))

> (square 5)
25

> (square 4)
16
SCHEME
Two Worlds
Visible world

Execution world
LAMBDA
Two Worlds

Visible world

\( \text{expression} \)

\( \lambda x . (\ast x x) \)

Execution world
Two Worlds

Visible world

expression

Execution world

\(\text{lambda}(x)(\ast\ x\ x)\)
Two Worlds

A compound procedure that squares its argument

Visible world

Expression

(\text{lambda}(x)(\ast \ x \ x))

Execution world

eval

\text{lambda-rule}

Value
Two Worlds

Visible world

(expression)

Execution world

A compound procedure that squares its argument
Two Worlds

Visible world

(expression)

Lambda expression

Execution world

A compound procedure that squares its argument
INTRODUCTION

LAMBDA & DEFINE
(\texttt{lambda}(x)(\texttt{*} x x))
(\lambda(x) (\ast x x))  

(lambd (a1) ...)
LAMBDA & DEFINE

Introduction

\[(\text{lambda}(x)(\text{* } x \ x)) \quad \text{(lambda (a1) ...)}\]

\[(\text{define square (lambda}(x)(\text{* } x \ x)))\]
LAMBDA & DEFINE

Introduction

\[ (\text{lambda}(x)(\ast x x)) \quad \text{(lambda (a1) ...)} \]

\[ \text{(define square (lambda(x)(\ast x x))} \quad \text{undefined} \]
(lambda(x)(* x x))

(define square (lambda(x)(* x x)))

(square 5)

(lambda (a1) ...)

undefined
LAMBDA & DEFINE

Introduction

(lambda(x)(* x x))  (lambda (a1) ...)

(define square (lambda(x)(* x x)))

(square 5)  undefined

25
**LAMBDA & DEFINE**

**Introduction**

\[
\begin{align*}
    & (\text{lambda}(x)(\ast \ x \ x)) & (\text{lambda} \ (a1) \ \ldots ) \\
    & (\text{define} \ \text{square} \ (\text{lambda}(x)(\ast \ x \ x))) \\
    & (\text{square} \ 5) & \text{undefined} \\
    & (\text{define} \ (\text{square} \ x)(\ast \ x \ x)) \\
    & \text{undefined} & 25
\end{align*}
\]
LAMBDA & DEFINE

Introduction

```
(lambda (x) (* x x))  
(define square (lambda (x) (* x x)))  
(square 5)  
(define (square x) (* x x))  
(lambda (a1) ...)  
```

undefined

25

undefined

syntactic sugar
LAMBDA & DEFINE
Syntax

\( (\text{lambda} \ (x \ y) \ (/ \ (+ \ x \ y) \ 2)) \)
LAMBDA & DEFINE

Syntax

```
(lambda (x y) (/ (+ x y) 2))
```

special keyword
LAMBDA & DEFINE

Syntax

```
(lambda (x y) (/ (+ x y) 2))
```

1st operand: the parameter list:

- a list of names (could be empty)
- determines the number of operands required
(lambda (x y) (/ (+ x y) 2))

2nd operand: the body:

- any expression
- not evaluated when the lambda is evaluated.
- evaluated when the procedure is applied.
THE VALUE OF A LAMBDA EXPRESSION IS A ……….
LAMBDA & DEFINE

Semantics

THE VALUE OF A LAMBDA EXPRESSION IS A PROCEDURE
• Procedures and procedural abstractions

• Using procedures to capture processes
PROCEDURES & PROCESSES

A Common Pattern

> (⋆ 3 3)

> (⋆ 25 25)
> (\*) 3 3
> (\*) 25 25
> (\*) foobar foobar
PROCEDURES & PROCESSES

A Common Pattern

\[ > (* 3 3) \]
\[ > (* 25 25) \]
\[ > (* \text{foobar} \text{foobar}) \]
\[
(lambda(x)(* x x))
\]
PROCEDURES & PROCESSES
A Common Pattern

> (* 3 3)
> (* 25 25)
> (* foobar foobar)

(lambda(x)(* x x))

name for the thing that changes
PROCEDURES & PROCESSES

A Common Pattern

\[
\begin{aligned}
&> (\ast \; 3 \; 3) \\
&> (\ast \; 25 \; 25) \\
&> (\ast \; \text{foobar} \; \text{foobar}) \\
&\text{(lambda}(x) (\ast \; x \; x))
\end{aligned}
\]

name for the thing that changes

common pattern to capture
PROCEDURES & PROCESSES

Modularity

\((\text{sqrt} \ (\ + \ (\ * \ 3 \ 3)(\ * \ 4 \ 4)))\)

\((\text{sqrt} \ (\ + \ (\ * \ 9 \ 9)(\ * \ 16 \ 16)))\)

\((\text{sqrt} \ (\ + \ (\ * \ 4 \ 4)(\ * \ 4 \ 4)))\)
(define square (lambda (x) (* x x)))
(define square (lambda (x) (* x x)))

(define pythagoras
(define square (lambda (x) (* x x)))

(define pythagoras
  (lambda (x y)
(define square (lambda (x) (* x x)))

(define pythagoras
  (lambda (x y)
    (sqrt (+ (square x) (square y))))))
PROCEDURES & PROCESSES

Why Modularity?

\[
(\text{define} \ \text{square} \ (\text{lambda} \ (x) \ (* \ x \ x)))
\]

\[
(\text{define} \ \text{pythagoras} \\
 \quad (\text{lambda} \ (x \ y) \\
 \quad \quad (\text{sqrt} \ (+ \ (\text{square} \ x) \ (\text{square} \ y))))))
\]
PROCEDURES & PROCESSES

Why Modularity?

Breaking computation into modules that capture **commonality**
PROCEDURES & PROCESSES

Why Modularity?

Breaking computation into modules that capture commonality

• enables reuse (square)
PROCEDURES & PROCESSES

Why Modularity?

Breaking computation into modules that capture *commonality*

- *enables* reuse (square)
- *isolates* details of computation within a procedure from its use
PROCEDURES & PROCESSES

Modularity

Many ways to divide.
Many ways to divide.

\[
\begin{align*}
\text{(define square (lambda (x) (* x x)))} \\
\text{(define sum-squares (lambda (x y) (+ (square x) (square y))))}
\end{align*}
\]
Many ways to divide.

```
(define square (lambda (x) (* x x)))
(define sum-squares
  (lambda (x y) (+ (square x) (square y))))
(define pythagoras
  (lambda (x y) (sqrt (sum-squares x y))))
```
PROCEDURES & PROCESSES

Abstracting the Process

Stages in capturing common patterns of computation:
Stages in capturing common patterns of computation:

- identify modules or stages of process
Stages in capturing common patterns of computation:

• identify modules or stages of process

• capture each module within a procedural abstraction
PROCEDURES & PROCESSES
Abstracting the Process

Stages in capturing common patterns of computation:

• identify modules or stages of process
• capture each module within a procedural abstraction
• construct a procedure to control the interactions between the modules
PROCEDURES & PROCESSES
Abstracting the Process

Stages in capturing common patterns of computation:

• identify modules or stages of process
• capture each module within a procedural abstraction
• construct a procedure to control the interactions between the modules
• repeat the process within each module as necessary
Stages in capturing common patterns of computation:

- **identify** modules or **stages** of process
- **capture** each module within a **procedural** abstraction
- **construct a** **procedure to control**
  the interactions between the modules
- **repeat** the process within each module as necessary
PROCEDURES & PROCESSES
Abstracting the Process

Stages in capturing common patterns of computation:

- **identify** modules or **stages** of process
  - capture **each module** within a **procedural** abstraction
  - construct a **procedure to control** the interactions between the modules
  - **repeat** the process within each module as necessary
Find an approximation of $\sqrt{x}$

Make a guess $G$

Improve the guess by averaging $G$ and $x/G$

Keep improving the guess until it is good enough
Find an approximation of $\sqrt{x}$

Make a guess $G$

Improve the guess by averaging $G$ and $x/G$

Keep improving the guess until it is good enough
Find an approximation of $\sqrt{x}$

Make a guess $G$

Improve the guess by averaging $G$ and $x/G$

Keep improving the guess until it is good enough

how do we create a new guess?
PROCEDURES & PROCESSES

Identify stages

Find an approximation of \( \sqrt{x} \)

Make a guess \( G \)

**Improve the guess by averaging** \( G \) **and** \( x/G \)

Keep improving the guess until it is good enough

how do we control the process of using the new guess in place of the old one?
Find an approximation of $\sqrt{x}$

Make a guess $G$

Improve the guess by averaging $G$ and $x/G$

Keep improving the guess until it is good enough

when is something close enough?
PROCEDURES & PROCESSES

Identify stages

Find an approximation of $\sqrt{x}$

Make a guess $G$

Improve the guess by averaging $G$ and $x/G$

Keep improving the guess until it is good enough

how do we control the process of using the new guess in place of the old one?

how do we create a new guess?

when is something close enough?
STAGE 1: Identify modules or stages of process

STAGE 2: Capture each module within a procedural abstraction

STAGE 3: Construct a procedure to control the interactions between the modules

STAGE 4: Repeat the process within each module as necessary
PROCEDURES & PROCESSES
Each Module a Procedure

when is something close enough?

(define square (lambda (x) (* x x)))

(define close-enough
  (lambda (guess x)
    (<= (abs (- (square guess) x)) 0.001)))
PROCEDURES & PROCESSES
Each Module a Procedure

when is something close enough?

(define square (lambda (x) (* x x)))

(define close-enough (lambda (guess x)
  (< (abs (- (square guess) x)) 0.001)))
how do we control the process of using the new guess in place of the old one?

(define average
  (lambda (a b) (/ (+ a b) 2)))

(define improve
  (lambda (guess x)
    (average guess (/ x guess)))))
how do we control the process of using the new guess in place of the old one?

```
(define average
  (lambda (a b) (/ (+ a b) 2)))

(define improve
  (lambda (guess x)
    (average guess (/ x guess)))))
```
PROCEDURES & PROCESSES

Why Modularity?
PROCEDURES & PROCESSES

Why Modularity?

• taking average is a very common operation that we are likely to use in other places.
PROCEDURES & PROCESSES

Why Modularity?

• taking average is a very common operation that we are likely to use in other places.

• separate implementation details from use
PROCEDURES & PROCESSES

Why Modularity?

• taking average is a very common operation that we are likely to use in other places.

• separate implementation details from use
  • another implementation
PROCEDURES & PROCESSES

Why Modularity?

• taking average is a very common operation that we are likely to use in other places.

• separate implementation details from use

  • another implementation

(\text{define average} \\
 (\text{lambda} (m n) (* (+ m n) 0.5)))
PROCEDURES & PROCESSES

Why Modularity?

• taking average is a very common operation that we are likely to use in other places.

• separate implementation details from use
  • another implementation
  • no other changes to procedure that use average
PROcedures & Processes

Why Modularity?

- taking average is a very common operation that we are likely to use in other places.
- separate implementation details from use
  - another implementation
  - no other changes to procedure that use average
  - variables are internal to procedure they cannot be referred outside the scope of lambda
Stages in capturing common patterns of computation:

- identify modules or stages of process
- capture each module within a procedural abstraction

construct a **procedure to control**
the interactions between the modules

- repeat the process within each module as necessary
Find an approximation of $\sqrt{x}$

Make a guess $G$

Improve the guess by averaging $G$ and $x/G$

Keep improving the guess until it is good enough
PROCEDURES & PROCESSES

Controlling the Process

Find an approximation of $\sqrt{x}$

Make a guess $G$

Improve the guess by averaging $G$ and $x/G$

Keep improving the guess until it is good enough

we need to make a decision!
PROCEDURES & PROCESSES

The IF Special Form

(if <predicate> <consequence> <alternative>)
(if <predicate> <consequence> <alternative>)

- Evaluator first evaluates the <predicate> expression.
The IF Special Form

(if <predicate> <consequence> <alternative>)

• Evaluator first evaluates the <predicate> expression.

• If it evaluates to TRUE value, then it evaluates and returns the values of <consequence> expression.
(if <predicate> <consequence> <alternative>)

• Evaluator first evaluates the <predicate> expression.

• If it evaluates to **TRUE** value, then it evaluates and returns the values of <consequence> expression.

• Otherwise, it evaluates and return the value of the <alternative> expression.
PROCEDURES & PROCESSES

The IF Special Form

(if <predicate> <consequence> <alternative>)

• Evaluator first evaluates the <predicate> expression.

• If it evaluates to TRUE value, then it evaluates and returns the values of <consequence> expression.

• Otherwise, it evaluates and return the value of the <alternative> expression.

Why must this be a special form?
Find an approximation of $\sqrt{x}$

Make a guess $G$

Improve the guess by averaging $G$ and $x/G$

Keep improving the guess until it is good enough

we need to make a decision!
Main process:

\[
\text{(if (close-enough? G x) } \\
\quad G \text{ (improve G x))}
\]
Main process:

\[
\text{if (close-enough? G x)} \\
\quad G \\
\text{(improve G x)}
\]

We want to use the value returned by improving things as the new guess, and repeat the process.
Main process:

\[
\begin{align*}
&\quad \text{(define } \textit{sqrt-loop} \text{ (lambda } G \text{ x)} \\
&\quad \text{ (if (close-enough? } G \text{ x)} \\
&\quad \quad G \\
&\quad \quad \text{ sqrt-loop (improve } G \text{ x) x))}
\end{align*}
\]

Call the main process \textit{sqrt-loop} and \textit{reuse} it!
(define my-sqrt
  (lambda (x)
    (sqrt-loop 1.0 x)))
(define my-sqrt
  (lambda (x)
    (sqrt-loop 1.0 x)))

initial guess
(define sqrt-loop (lambda (G x)
  (if (close-enough? G x)
      G
      (sqrt-loop (improve G x) x)))

Recursion
(my-sqrt 2)
  (sqrt-loop 1.0 2)
  (if (close-enough? 1.0 2) ... ...)
  (sqrt-loop (improve 1.0 2) 2)
PROCEDURES & PROCESSES

Check by Tracing

(my-sqrt 2)
  (sqrt-loop 1.0 2)
    (if (close-enough? 1.0 2) ... ...)
    (sqrt-loop (improve 1.0 2) 2)

(sqrt-loop 1.5 2)
  (if (close-enough? 1.5 2) ... ...)
(sqrt-loop (improve 1.4166666 2)
Stages in capturing common patterns of computation:

- identify modules or stages of process
- capture each module within a procedural abstraction
- construct a procedure to control the interactions between the modules
- repeat the process within each module as necessary
PROCEDURES & PROCESSES
Abstracting the Process

Stages in capturing common patterns of computation:

• **identify** modules or **stages** of process

• **capture** each module within a **procedural** abstraction

• **construct a** **procedure to control** the interactions between the modules

• **repeat** the process within each module as necessary