

Lecture 2

Functional Programming & Scheme



T. METIN SEZGIN

Announcements



1. Reading SICP 1.1 (pages 1-31) next lecture
2. Etutor – at the end
3. Etutor assignment due next Friday
4. Labs (PSes) start this week



Method	Description	Weight %
Quiz	10 quizzes	10,00
Project	Project	20,00
Midterm Exam	Midterm	25,00
Final Exam	Final	30,00
Participation	Participation	5,00
Laboratory	PS Sessions	10,00
Total:		100,00

Lecture Nuggets



nugget

/ˈnʌɡɪt/

noun

a small lump of gold or other precious metal found ready-formed in the earth.

- a small chunk or lump of another substance.
"nuggets of meat"

Benzer:

lump

chunk

small piece

hunk

mass

clump

wad



- a valuable idea or fact.
"nuggets of information"

Lecture Nuggets



- You only know one way of programming/thinking
 - You are imperative programmers
 - Functional programming an entirely new concept
- We can specify programs entirely through functions
- 3 major elements of language
 - Primitives
 - Means Combination
 - Abstraction
- Read-Eval-Print loop
- Functions are first class citizens

Nugget



You only know one way of
programming/thinking

Main programming paradigms

Paradigm	Description	Main traits	Related paradigm(s)	Examples
<u>Imperative</u>	Programs as <u>statements</u> that <i>directly</i> change computed <u>state</u> (<u>datafields</u>)	Direct <u>assignments</u> , common <u>data structures</u> , <u>global variables</u>		<u>C</u> , <u>C++</u> , <u>Java</u> , <u>Kotlin</u> , <u>PHP</u> , <u>Python</u> , <u>Ruby</u>
<u>Procedural</u>	Derived from structured programming, based on the concept of <u>modular programming</u> or the <i>procedure call</i>	<u>Local variables</u> , sequence, selection, <u>iteration</u> , and <u>modularization</u>	Structured, imperative	<u>C</u> , <u>C++</u> , <u>Lisp</u> , <u>PHP</u> , <u>Python</u>
<u>Functional</u>	Treats <u>computation</u> as the evaluation of <u>mathematical functions</u> avoiding <u>state</u> and <u>mutable</u> data	<u>Lambda calculus</u> , <u>compositionality</u> , <u>formula</u> , <u>recursion</u> , <u>referential transparency</u> , no <u>side effects</u>	Declarative	<u>C++</u> , ^[1] <u>C#</u> , ^[2] ^[circular reference] <u>Clojure</u> , <u>CoffeeScript</u> , ^[3] <u>Elixir</u> , <u>Erlang</u> , <u>F#</u> , <u>Haskell</u> , <u>Java</u> (since version 8), <u>Kotlin</u> , <u>Lisp</u> , <u>Python</u> , <u>R</u> , ^[4] <u>Ruby</u> , <u>Scala</u> , <u>SequenceL</u> , <u>Standard ML</u> , <u>JavaScript</u> , <u>Elm</u>
<u>Object-oriented</u>	Treats <u>datafields</u> as <i>objects</i> manipulated through predefined <u>methods</u> only	<u>Objects</u> , methods, <u>message passing</u> , <u>information hiding</u> , <u>data abstraction</u> , <u>encapsulation</u> , <u>polymorphism</u> , <u>inheritance</u> , <u>serialization</u> -marshalling	Procedural	<u>Common Lisp</u> , <u>C++</u> , <u>C#</u> , <u>Eiffel</u> , <u>Java</u> , <u>Kotlin</u> , <u>PHP</u> , <u>Python</u> , <u>Ruby</u> , <u>Scala</u> , <u>JavaScript</u> ^{[8][9]}
<u>Declarative</u>	Defines program logic, but not detailed <u>control flow</u>	<u>Fourth-generation languages</u> , <u>spreadsheets</u> , <u>report program generators</u>		<u>SQL</u> , <u>regular expressions</u> , <u>Prolog</u> , <u>OWL</u> , <u>SPARQL</u> , <u>Datalog</u> , <u>XSLT</u>

Nugget



We can specify programs entirely
through functions

Write a function for factorial



- $\text{Fact}(x) = x * \text{fact}(x-1)$ (if $x > 1$)
- $\text{Fact}(x) = 1$ (if $x == 1$)
- $Y = x^2$

Advantages of functional programming



- Intuitive
- Functions are first-class citizens
 - Create
 - Bind to variables
 - Pass to functions
 - Return
- Allows declarative and composable style
 - Emphasis on modularity
 - Purely functional programming is easy to reason about
 - No side effects
 - Formally verifiable, fewer bugs
 - Finding increasing use in modern development patterns/languages

Advantages of functional programming

- Functions are first-class citizens
 - Create
 - Bind to variables
 - Pass to functions
 - Return
- Allows declarative and composable style
 - Emphasis on modularity
 - Purely functional programming is easy to reason about
 - No side effects
 - Formally verifiable, fewer bugs
 - Finding increasing use in modern development patterns/languages



1. Understand functional way of thinking
2. Understand how interpreters work
3. Think like an interpreter
4. Build an interpreter using scheme

Nugget



Three major elements of a language

Kinds of Language Constructs

- Primitives
- Means of combination
- Means of abstraction

```
def create_adder(x):  
    global tic  
    tic = x  
  
    def adder():  
        global tic  
        tic = tic + 1  
        return tic  
  
    return adder  
  
fun_a = create_adder(0)  
fun_b = create_adder(0)  
  
print(fun_a(), fun_b(), fun_a(), fun_b())
```

Language elements – primitives

- Self-evaluating primitives – value of expression is just object itself
 - Numbers: 29, -35, 1.34, 1.2e5
 - Strings: “this is a string” “ this is another string with %&^ and 34”
 - Booleans: #t, #f

Language elements – primitives

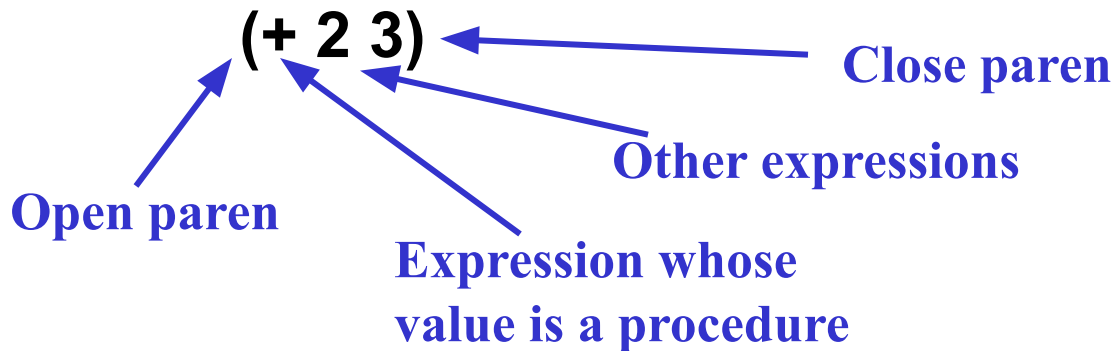
- Built-in procedures to manipulate primitive objects
 - Numbers: +, -, *, /, >, <, >=, <=, =
 - Strings: string-length, string=?
 - Booleans: boolean/and, boolean/or, not

Language elements – primitives

- Names for built-in procedures
 - $+$, $*$, $-$, $/$, $=$, ...
 - What is the value of such an expression?
 - $+ \square [\text{\#procedure ...}]$
 - Evaluate by looking up value associated with name in a special table

Language elements – combinations

- How do we create expressions using these procedures?



- Evaluate by getting values of sub-expressions, then applying operator to values of arguments

Language elements - combinations

- Can use nested combinations – just apply rules recursively

$$(+ (* 2 3) 4) \square 10$$

$$(* (+ 3 4) (- 8 2)) \square 42$$

Language elements -- abstractions

- In order to abstract an expression, need way to give it a name

(define score 23)

- This is a special form
 - Does not evaluate second expression
 - Rather, it pairs name with value of the third expression
- Return value is unspecified

Language elements -- abstractions

- To get the value of a name, just look up pairing in environment

score \square **23**

– Note that we already did this for +, *, ...

(define total (+ 12 13))

(* 100 (/ score total)) \square **92**

- This creates a loop in our system, can create a complex thing, name it, treat it as primitive

Scheme Basics

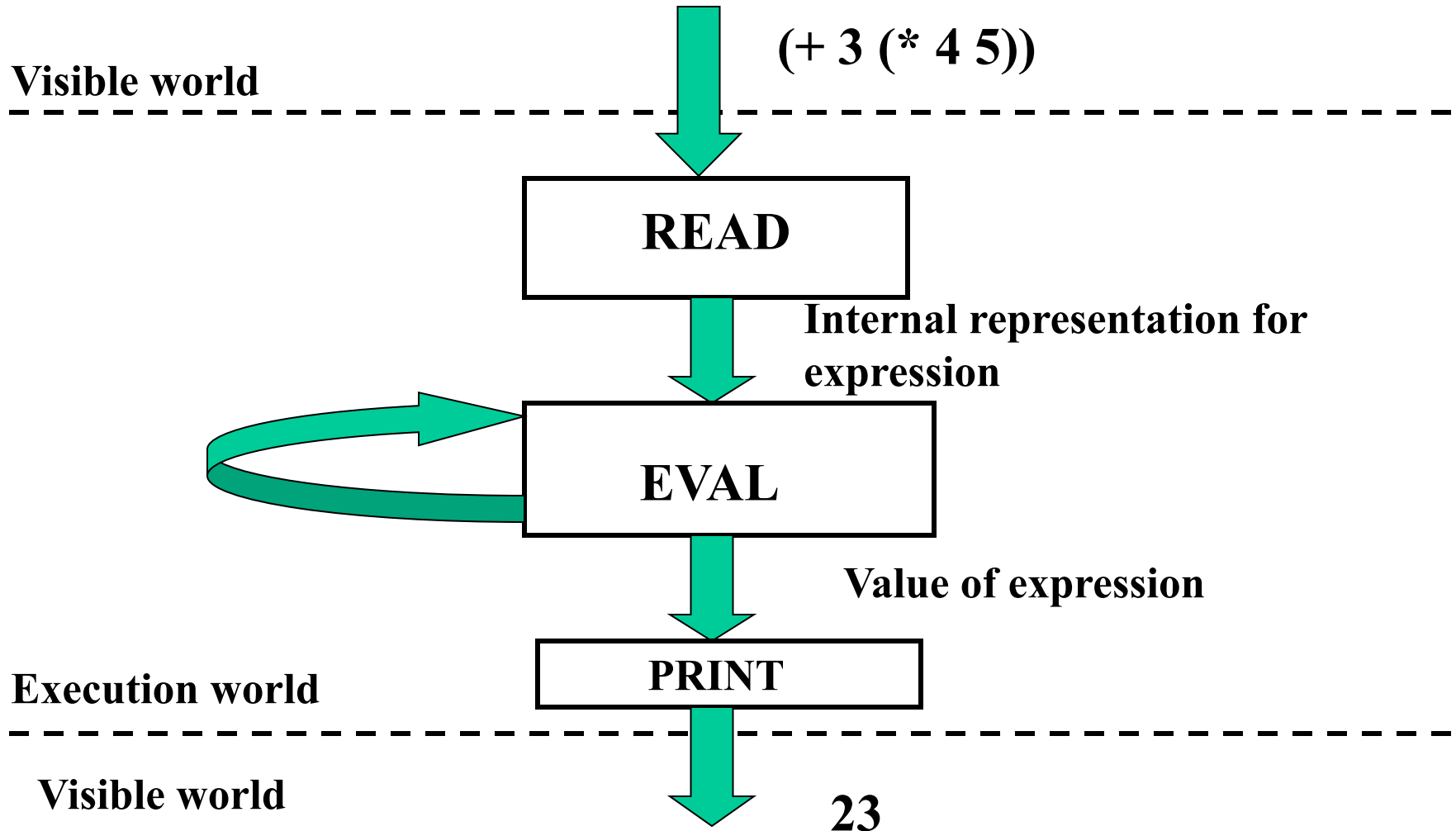
- Rules for evaluation
 1. If **self-evaluating**, return value.
 2. If a **name**, return value associated with name in environment.
 3. If a **special form**, do something special.
 4. If a **combination**, then
 - a. *Evaluate* all of the subexpressions of combination (in any order)
 - b. *apply* the operator to the values of the operands (arguments) and return result

Nugget

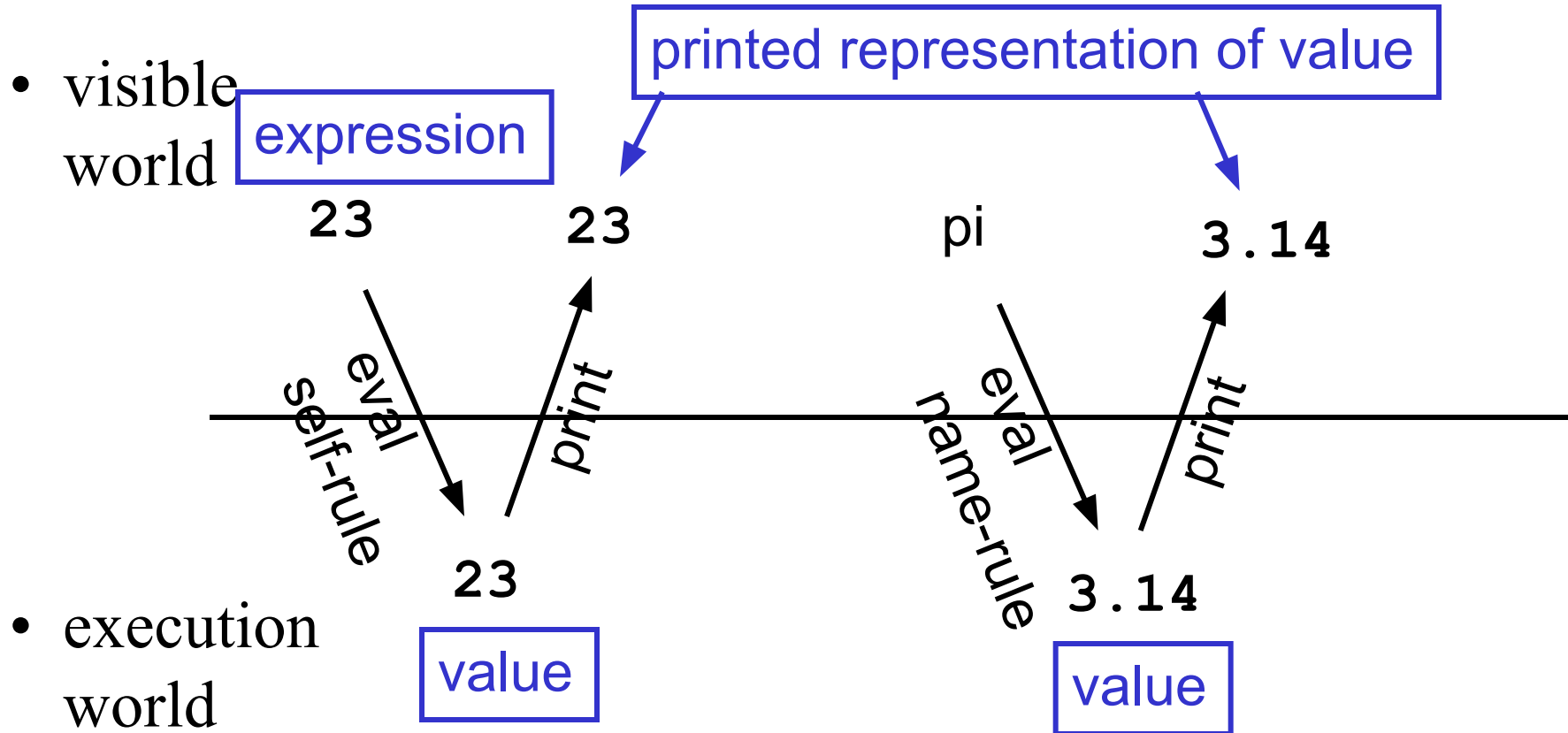


The concept of Read-Eval-Print

Read-Eval-Print



A new idea: two worlds



name-rule: look up value of name in current environment

Define special form

- define-rule:
 - evaluate 2nd operand only
 - name in 1st operand position is bound to that value
 - overall value of the define expression is undefined

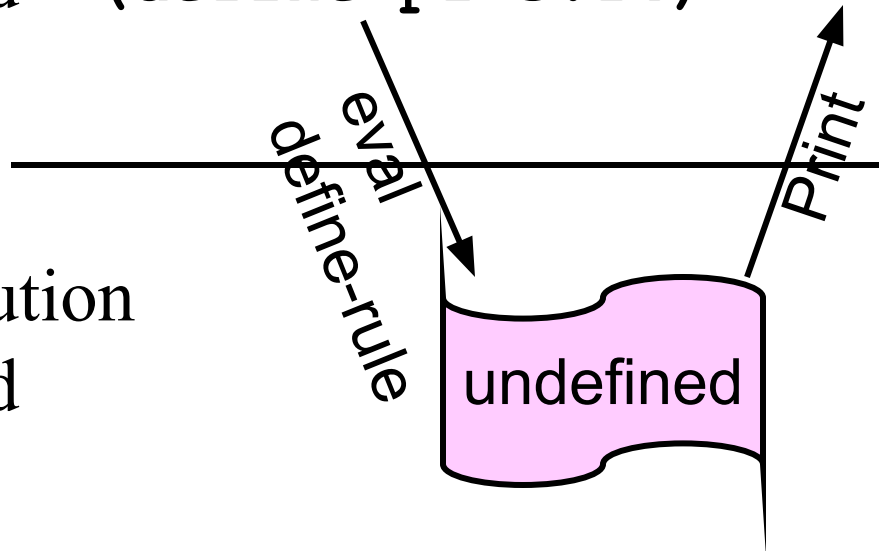
scheme versions differ

- visible

world `(define pi 3.14)`

`"pi --> 3.14"`

- execution
world



name value	
pi	3.14

Mathematical operators are just names

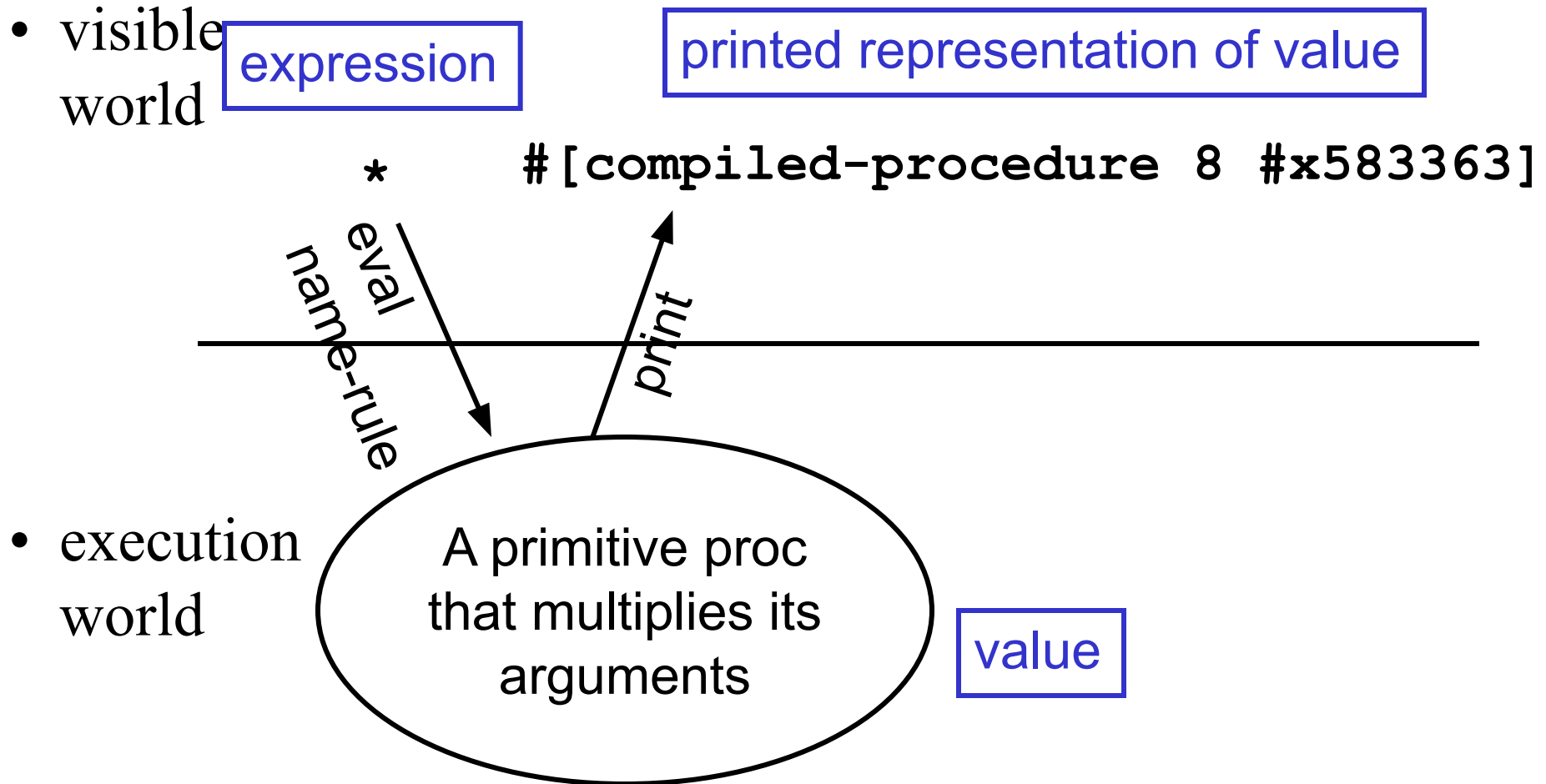
`(+ 3 5)` `□ 8`

`(define fred +)` `□ undef`

`(fred 4 6)` `□ 10`

- How to explain this?
- Explanation
 - `+` is just a name
 - `+` is bound to a value which is a procedure
 - line 2 binds the name **fred** to that same value

Primitive procedures are just values



Nugget



Functions are first class citizens

Hold your breath



Language elements -- abstractions

- Need to capture ways of doing things – use procedures

(lambda (x) (* x x))

The diagram shows the lambda expression **(lambda (x) (* x x))** with several annotations. A red arrow points from the word **parameters** to the **(x)** part. Another red arrow points from the word **body** to the **(* x x)** part. Below the expression, three blue arrows point upwards: one from **To process** to **lambda**, one from **something** to **(x)**, and one from **multiply it by itself** to **(* x x)**.

- Special form – creates a procedure and returns it as value

Language elements -- abstractions

- Use this anywhere you would use a procedure
`((lambda (x) (* x x)) 5)`

Scheme Basics

- Rules for evaluation
 1. If **self-evaluating**, return value.
 2. If a **name**, return value associated with name in environment.
 3. If a **special form**, do something special.
 4. If a **combination**, then
 - a. *Evaluate* all of the subexpressions of combination (in any order)
 - b. *apply* the operator to the values of the operands (arguments) and return result
- Rules for application
 1. If procedure is **primitive procedure**, just do it.
 2. If procedure is a **compound procedure**, then:
evaluate the body of the procedure with each formal parameter replaced by the corresponding actual argument value.

Language elements -- abstractions

- Use this anywhere you would use a procedure

((lambda (x) (* x x)) 5)

(* 5 5)

25

- Can give it a name

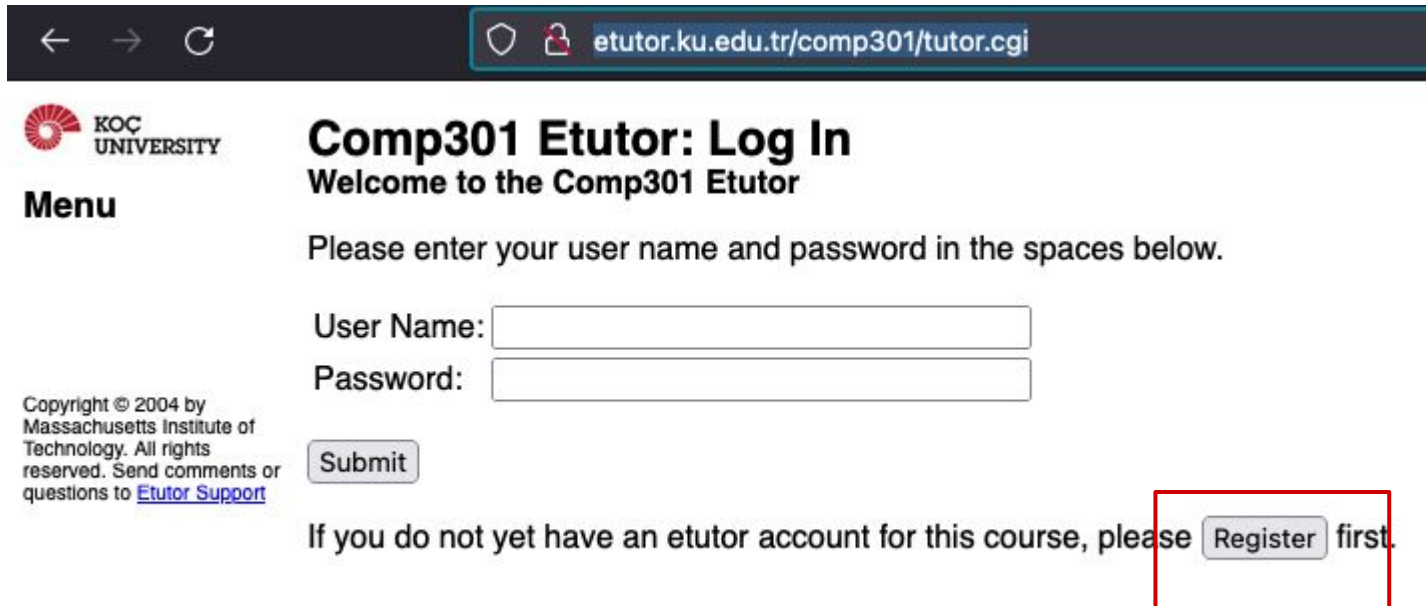
(define square (lambda (x) (* x x)))

(square 5) □ 25

Introducing Etutor




1.OPEN <http://etutor.ku.edu.tr/comp301/tutor.cgi>



The screenshot shows a web browser window with the address bar displaying etutor.ku.edu.tr/comp301/tutor.cgi. The page header includes the KOC UNIVERSITY logo and a "Menu" link. The main heading is "Comp301 Etutor: Log In" with a subtitle "Welcome to the Comp301 Etutor". Below this, a message says "Please enter your user name and password in the spaces below." There are two input fields: "User Name:" and "Password:". A "Submit" button is located below the password field. At the bottom, a message states "If you do not yet have an etutor account for this course, please" followed by a "Register" button and the word "first". The "Register" button is highlighted with a red rectangle.

← → ↻

etutor.ku.edu.tr/comp301/tutor.cgi

 **KOC UNIVERSITY**

Menu

Comp301 Etutor: Log In

Welcome to the Comp301 Etutor

Please enter your user name and password in the spaces below.

User Name:

Password:

Copyright © 2004 by
Massachusetts Institute of
Technology. All rights
reserved. Send comments or
questions to [Etutor Support](#)

If you do not yet have an etutor account for this course, please first.

2. REGISTER WITH YOUR KU E-MAIL



Menu

Copyright © 2004 by
Massachusetts Institute of
Technology. All rights
reserved. Send comments or
questions to [Etutor Support](#)


Comp301 Etutor: Registration

Welcome to the Comp301 Etutor

The use of this Etutor is restricted to students registered in Comp301.

To use this registration form, we require that you have an Koc University e-mail address. Please enter your full email address in the form of username@ku.edu.tr in the space below.

E-mail address:

When you register we will e-mail you a randomly generated password to the e-mail address you give us. You will log in to the Etutor using your Koc University user name and this password. You may change your password to something that you can more easily remember from the Menu. When you first log in, please review the help page using the  button.

3. A PASSWORD WILL BE SENT TO YOUR E-MAIL. USE YOUR KUNET ID AND PASSWORD TO LOGIN



Menu

Comp301 Etutor: Log In

Welcome to the Comp301 Etutor

Please enter your user name and password in the spaces below.

User Name:

Password:

Copyright © 2004 by
Massachusetts Institute of
Technology. All rights
reserved. Send comments or
questions to [Etutor Support](#)


If you do not yet have an etutor account for this course, please first.

4. CHOOSE A PROBLEM SET TO START YOUR ASSIGNMENT



Comp301 Etutor: Home



Menu

 Preferences

Welcome to the Comp301 Etutor, Thu Sep 30 04:43:45 2021

Please use  for help.



 Set Preferences
 Change Password

Copyright © 2004 by
Massachusetts Institute of
Technology. All rights
reserved. Send comments or
questions to [Etutor Support](#)

It will appear here, but not yet. When we assign the project, you will see here.

Lecture Nuggets



- You only know one way of programming/thinking
 - You are imperative programmers
 - Functional programming an entirely new concept
- We can specify programs entirely through functions
- 3 major elements of language
 - Primitives
 - Means Combination
 - Abstraction
- Functions are first class citizens