YES

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

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

Source: Wikipedia

Nugget

We can specify programs entirely through functions

Write a function for factorial

- Fact(x) = x * fact(x-1) (if x>1)

- 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

Learn Scheme

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

Used in practice to solve difficult problems

Generative AI assistance for any writing challenge Try It for Free →

grammarly engineering NLP/ML INFRASTRUCTURE PRODUCT MOBILE SEARCH

Running Lisp in Production

Vsevolod Dyomkin
Updated on November 2, 2020
INFRASTRUCTURE

At Grammarly, the foundation of our business, our core grammar engine, is written in Common Lisp. It currently processes more than a thousand sentences per second, is horizontally scalable, and has reliably served in production for almost three years. We noticed that there are very few, if any, accounts of how to deploy Lisp software to modern cloud infrastructure, so we thought that it would be a good idea to share our experience. The Lisp runtime and programming environment provides several unique—albeit obscure—capabilities to support production systems (for the impatient, they are described in the final chapter).

Wut Lisp?!!

Contrary to popular opinion, Lisp is an incredibly practical language for building

https://www.grammarly.com/blog/engineering/running-lisp-in-production/

Used in practice to solve difficult problems

The hardest bug I've ever debugged

As ideal as this story is so far, it has not been all rainbows and unicorns.

We've built an esoteric application (even by Lisp standards), and in the process have hit some limits of our platform. One unexpected thing was heap exhaustion during compilation. We rely heavily on macros, and some of the largest ones expand into thousands of lines of low-level code. It turned out that the SBCL compiler implements a lot of optimizations that allow us to enjoy quite fast generated code, but some of them require exponential time and memory resources. Unfortunately, there's no way to influence that by turning them off or tuning somehow. However, there exists a well-known general solution, `call-with-* style`, in which you trade off a little performance for better modularity (which turned out crucial for our use case) and debuggability.

Nugget

Three major elements of a language

Kinds of Language Constructs

- Primitives
- Means of combination
- Means of abstraction

```python
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?
  - + → [#procedure …]
  - Evaluate by looking up value associated with name in a special table

Language elements – combinations

- How do we create expressions using these procedures?

(+ 2 3)

Close paren

Other expressions

Open paren

Expression whose value is a procedure

- 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) →10

  (* (+ 3 4) (- 8 2)) →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 → 23

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

  (define total (+ 12 13))

  (* 100 (/ score total)) → 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

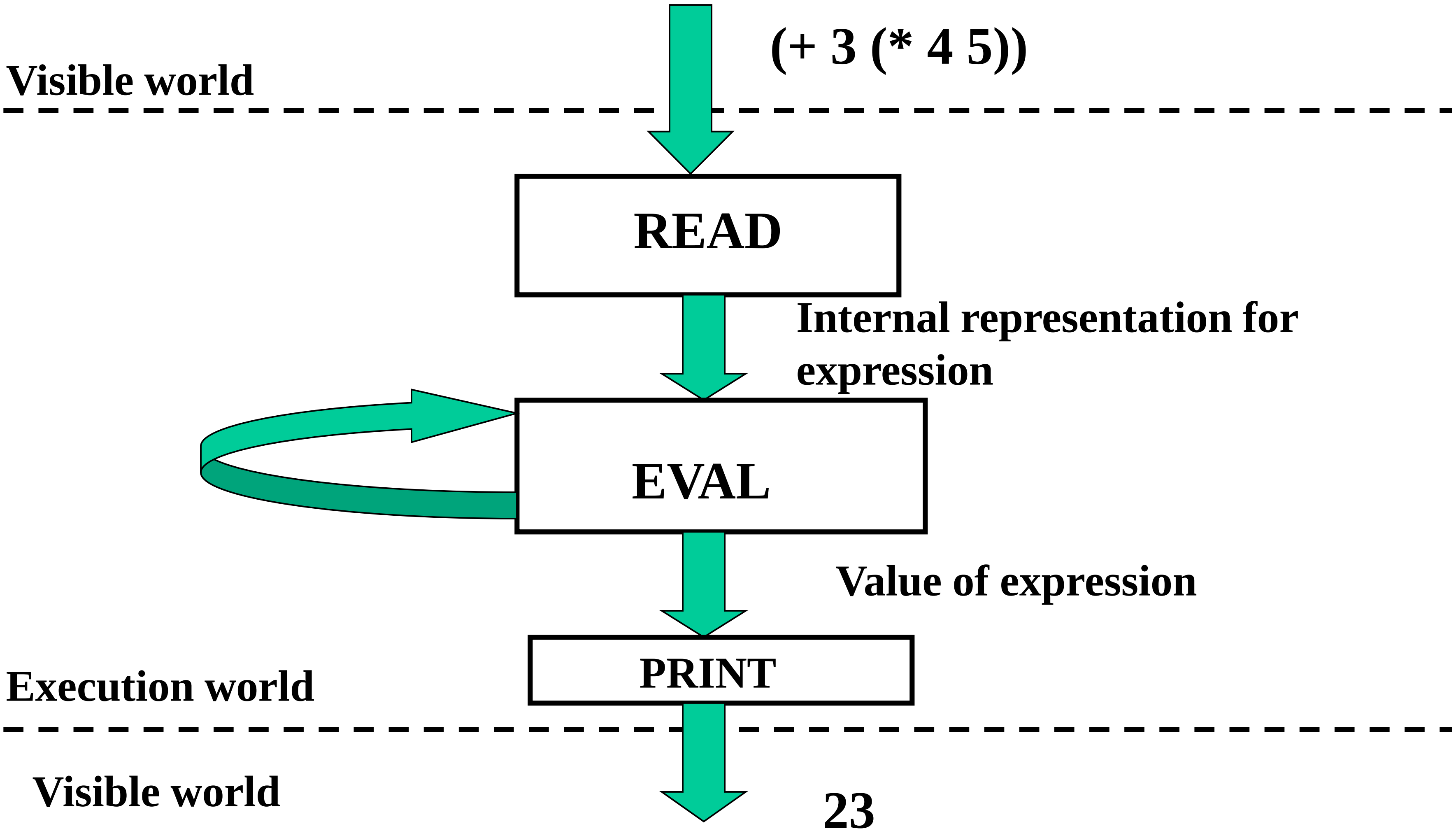
   a. *Evaluate* all of the subexpressions of combination (in any order)

   b. *apply* the operator to the values of the operands (arguments) and return result

Nugget

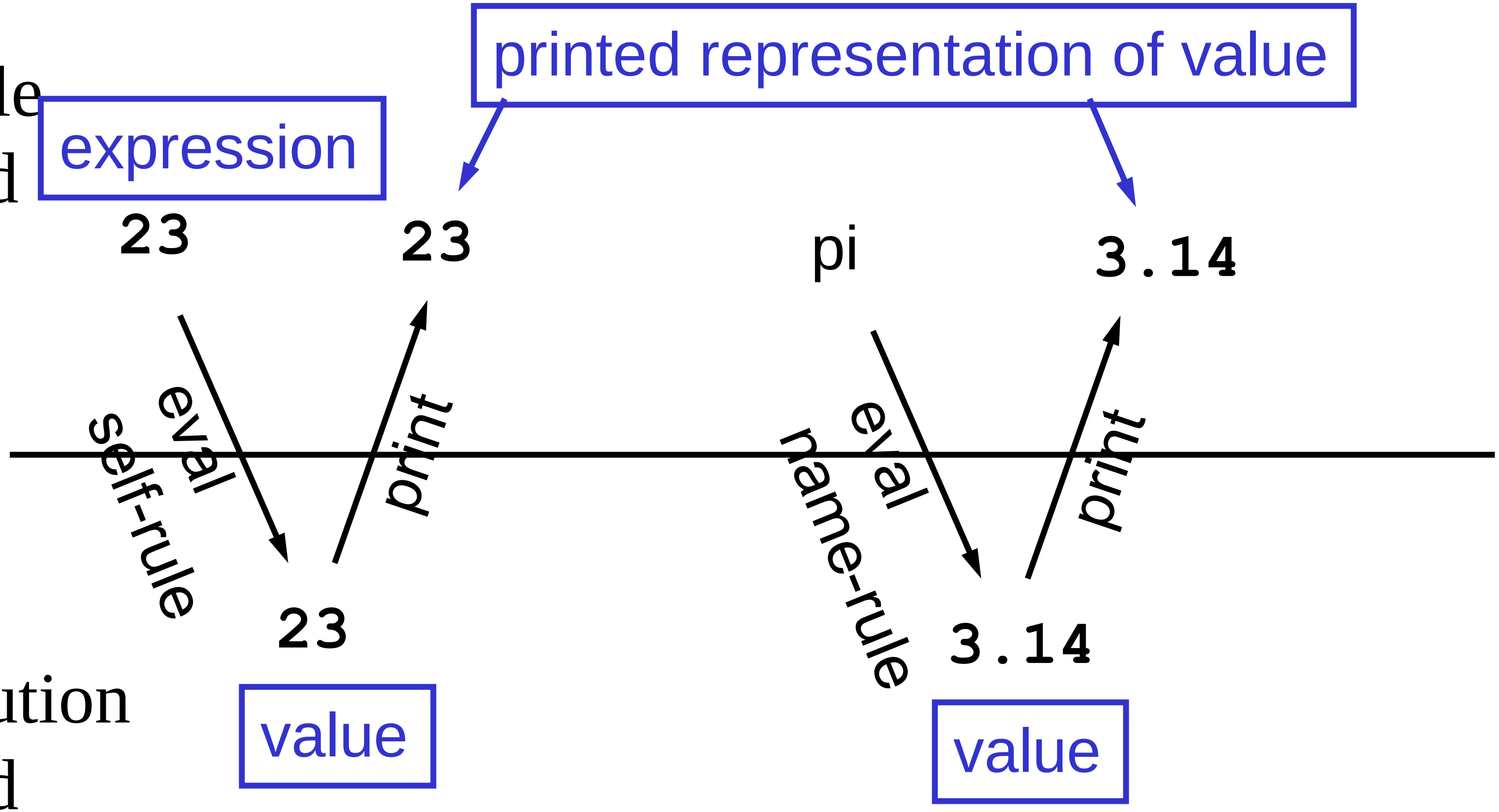
The concept of Read-Eval-Print

Read-Eval-Print



A new idea: two worlds

- visible world
- execution world



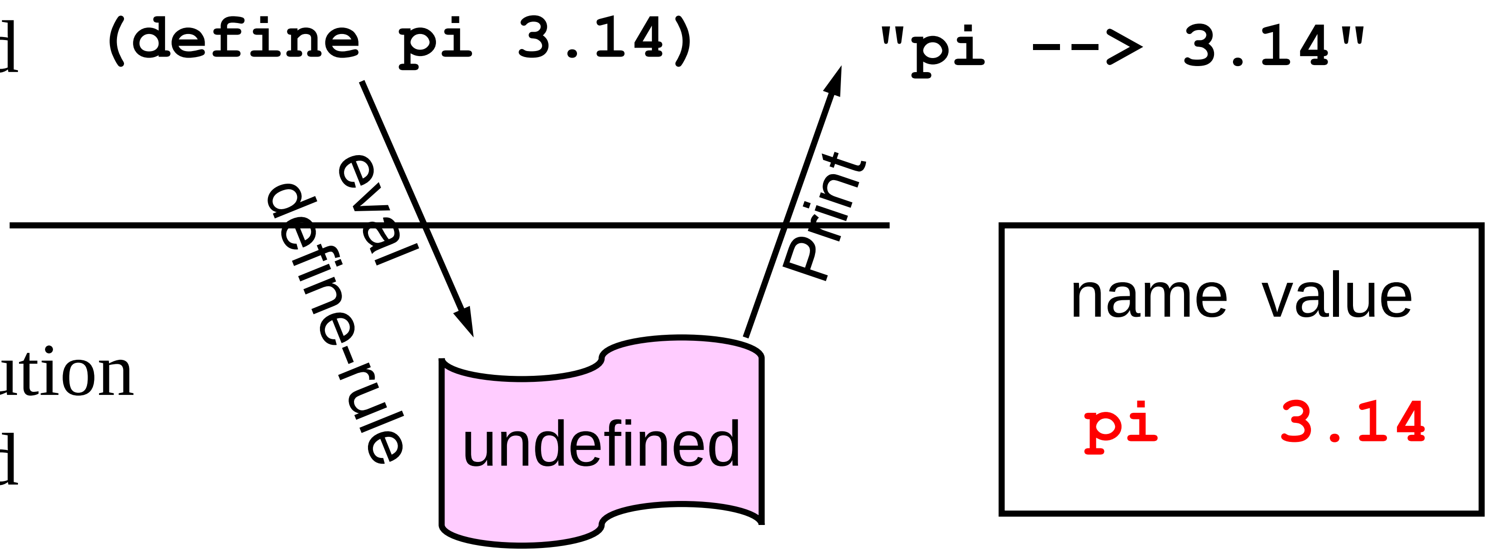
name-rule: look up value of name in current environment

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
- execution world



Mathematical operators are just names

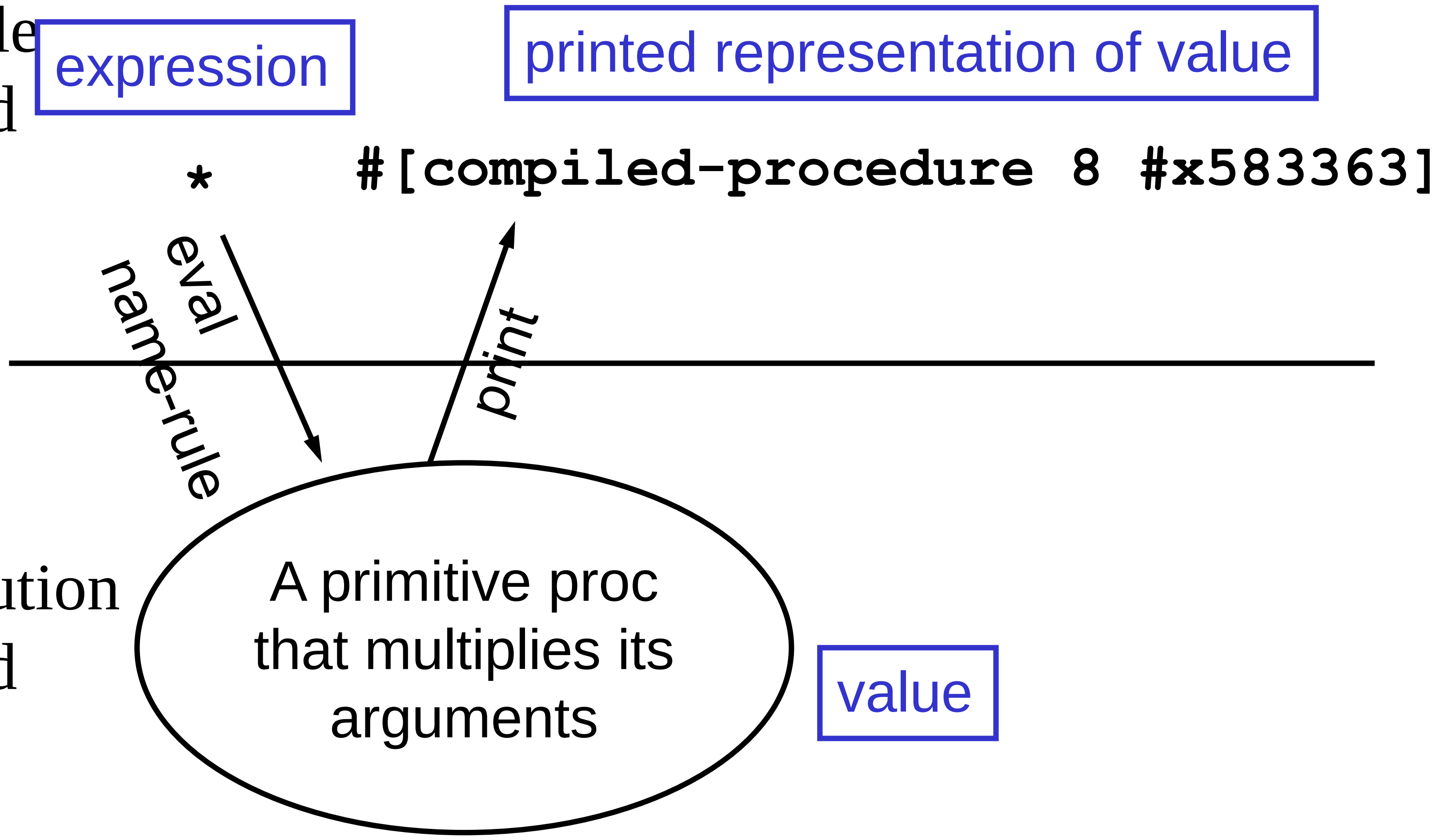
`(+ 3 5)`	➔ **`8`**
`(define fred +)`	➔ **`undef`**
`(fred 4 6)`	➔ **`10`**

- 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

- visible world
- execution world



Nugget

Functions are first class citizens

Hold your breath

Language elements -- abstractions

- Need to capture ways of doing things – use procedures

parameters

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

body

To process something multiply it by itself

•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)

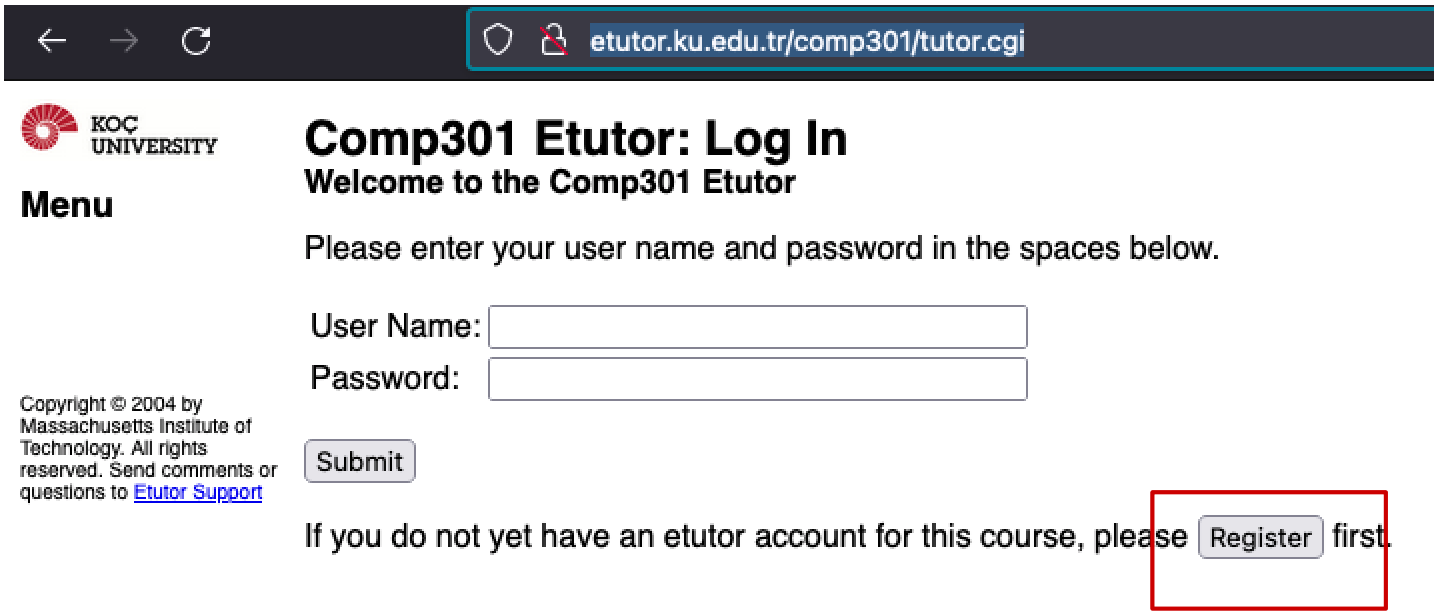
25

- 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



2. REGISTER WITH YOUR KU E-MAIL

KOÇ UNIVERSITY

Menu

Comp301 Etutor: Registration

Welcome to the Comp301 Etutor

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

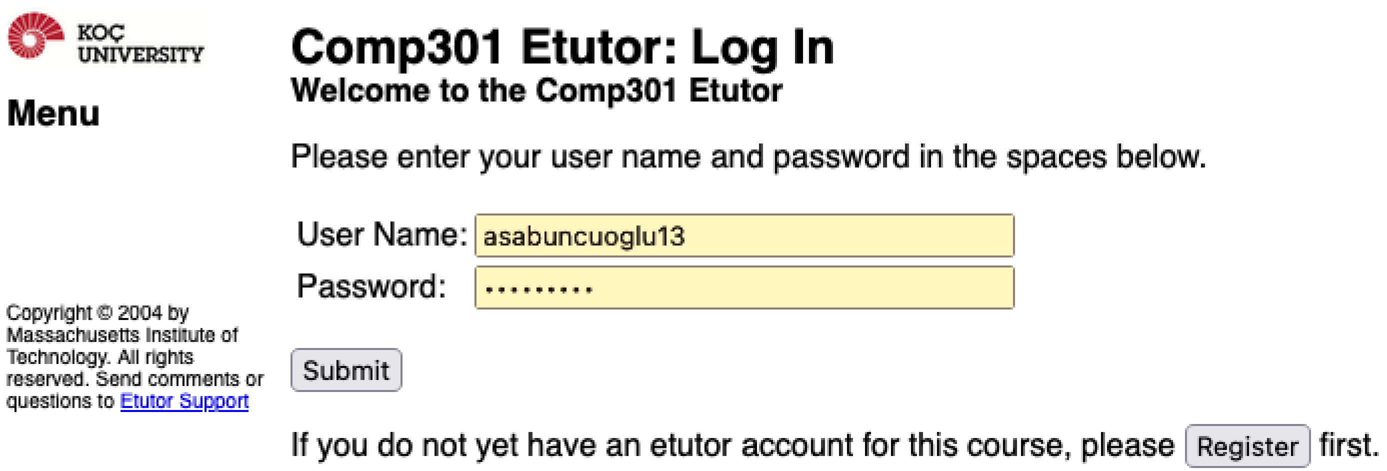
To use this registration form, we require that you have an Koc University e-mail address. Please enter your full email address in the form of username@ku.edu.tr in the space below.

E-mail address: asabuncuoglu13@ku.edu.tr

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.

Register

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



KOÇ UNIVERSITY

Menu

Comp301 Etutor: Log In

Welcome to the Comp301 Etutor

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

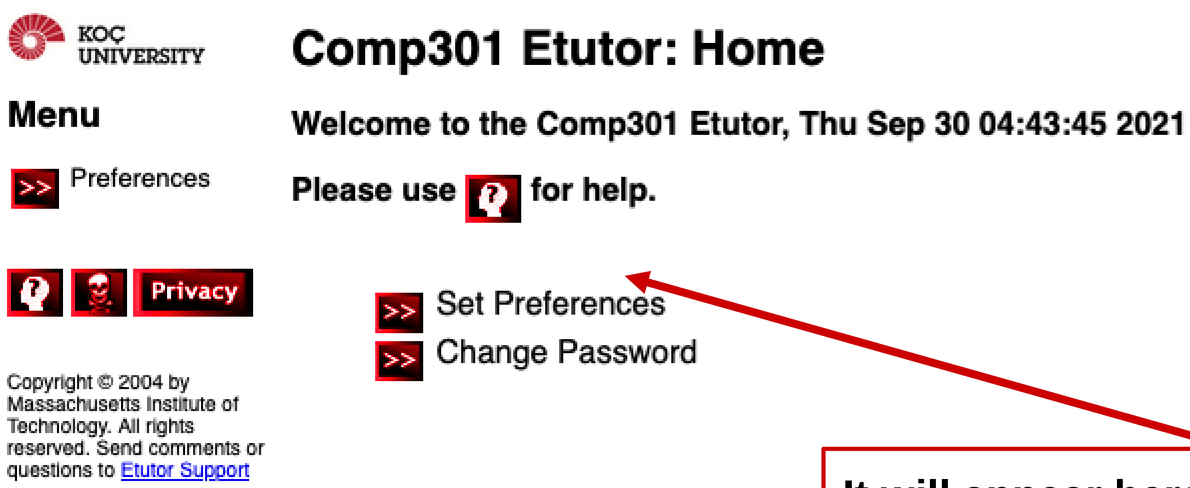
User Name: asabuncuoglu13

Password: ••••••••••

Submit

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

4. CHOOSE A PROBLEM SET TO START YOUR ASSIGNMENT



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