Programming Languages and Compilers (CS 421)

Sasa Misailovic

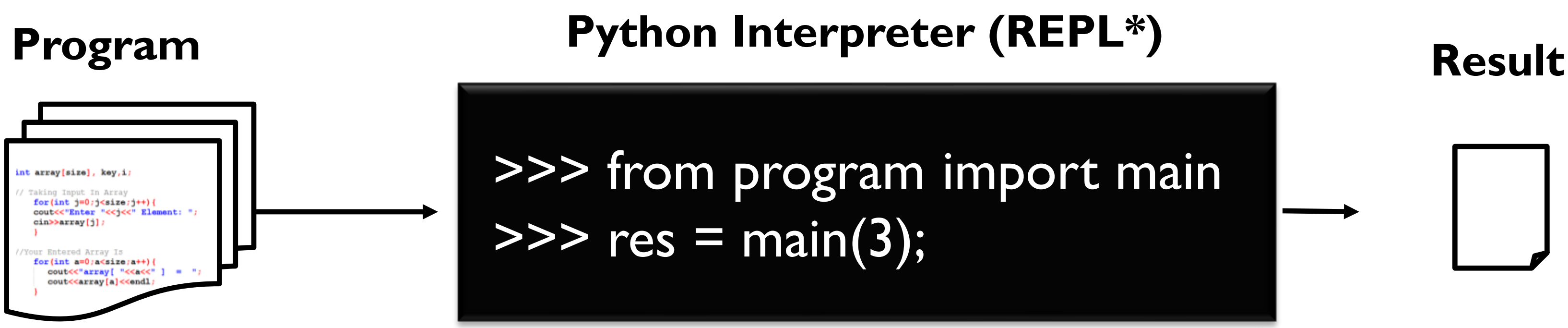
4110 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2024/CS421C

Based on slides by Elsa Gunter, which are based in part on previous slides by Mattox Beckman and updated by Vikram Adve and Gul Agha

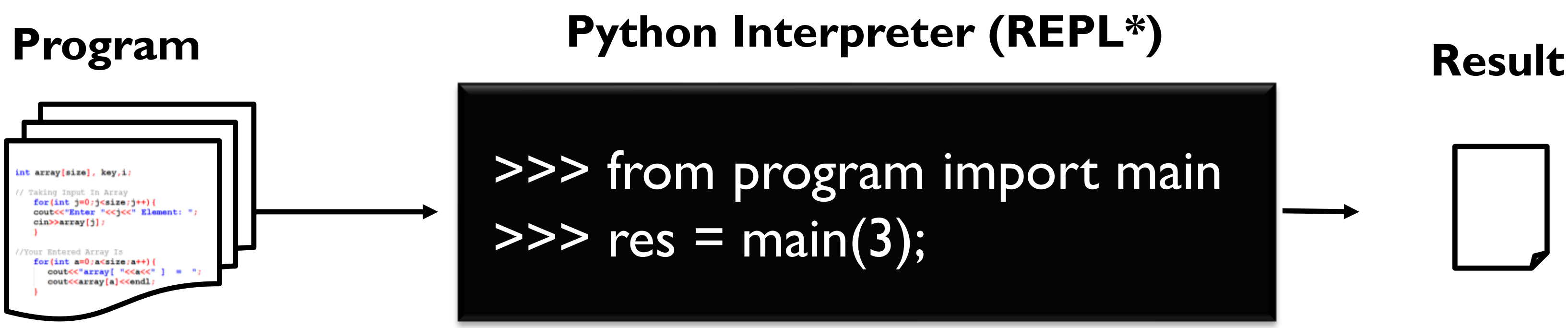
How It's Going

How It's Going



*REPL = Read, evaluate, print loop

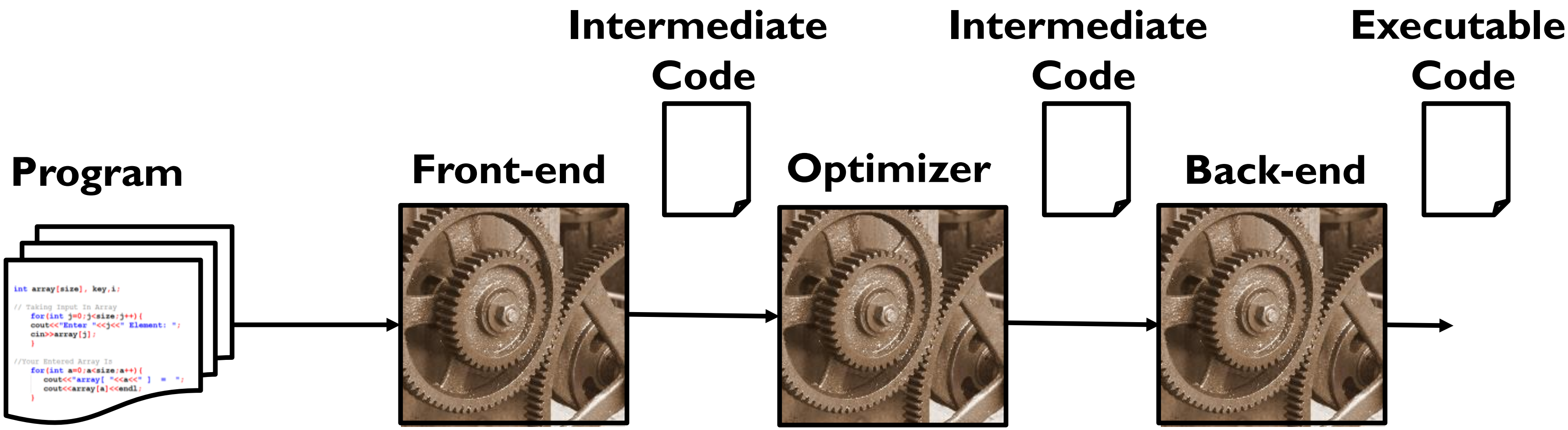
How It’s Going



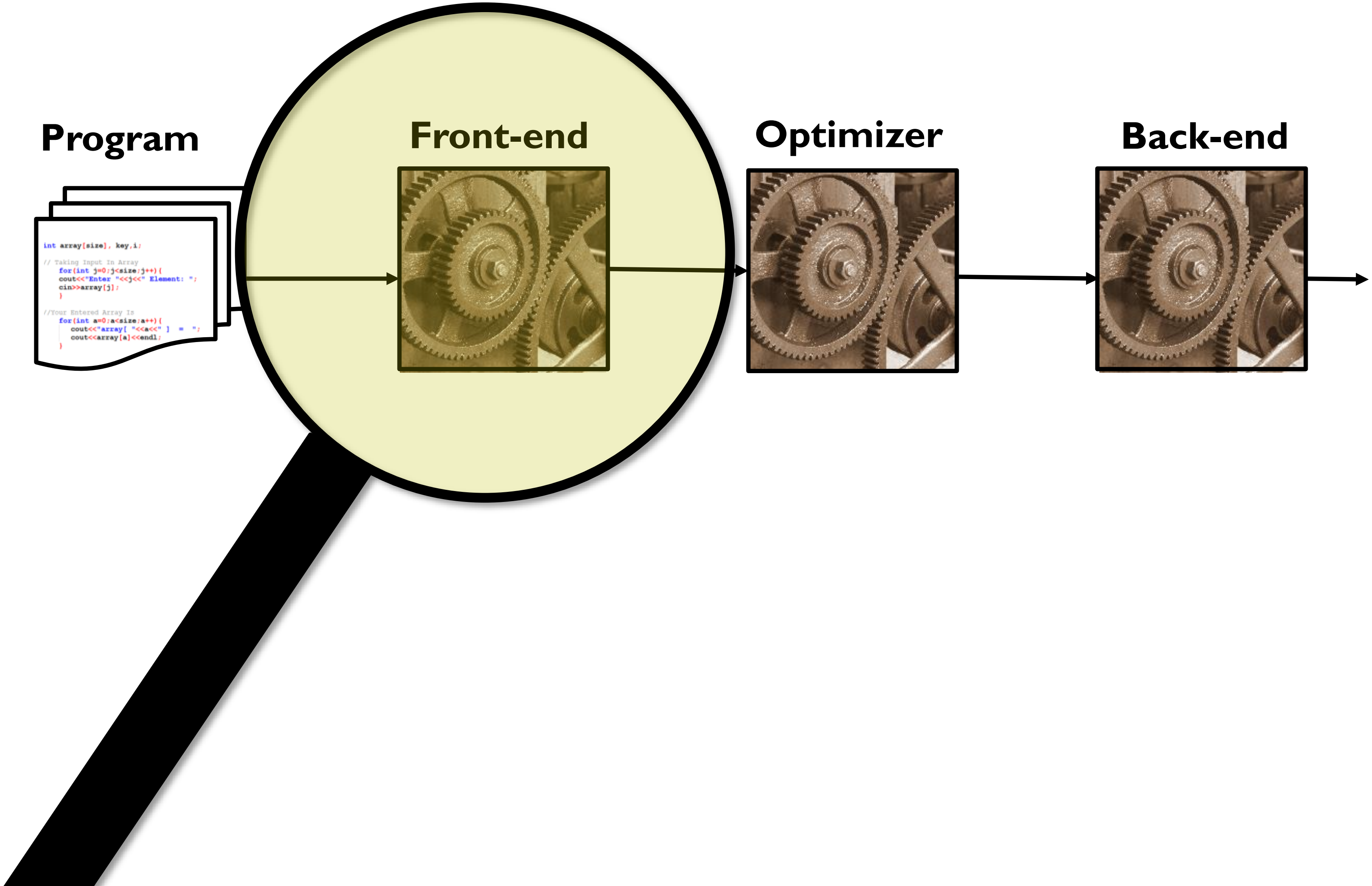
Our Journey: What’s inside?

*REPL = Read, evaluate, print loop

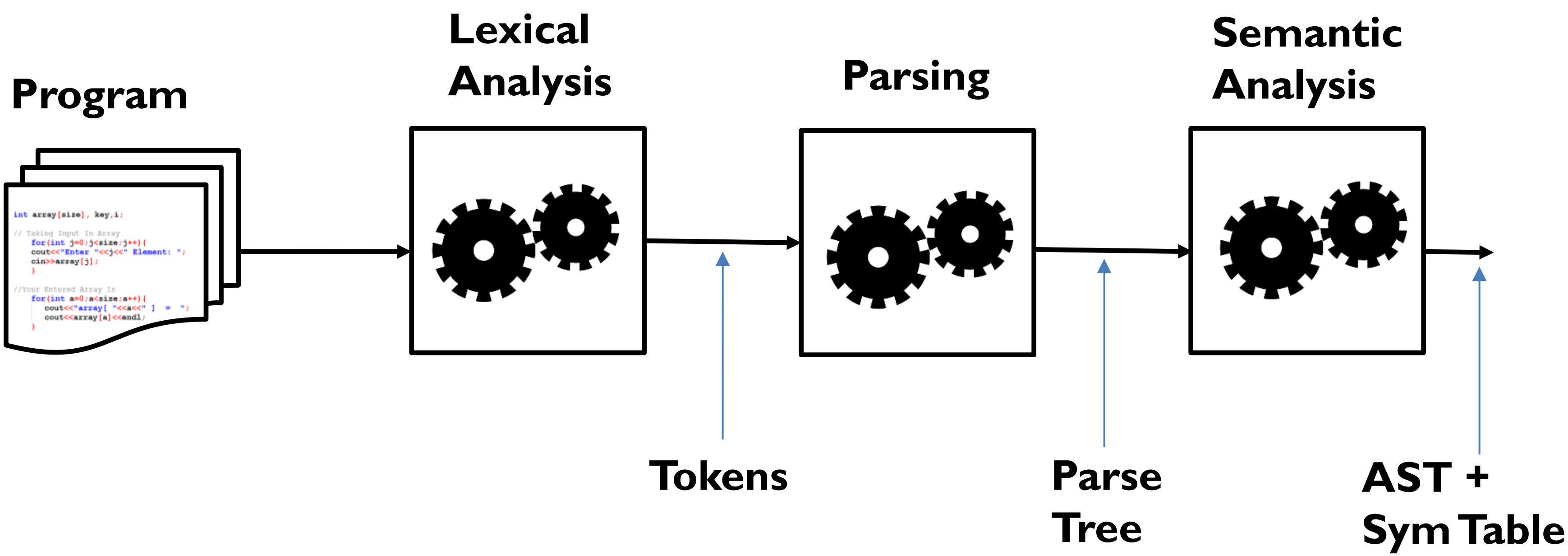
Compiler Overview



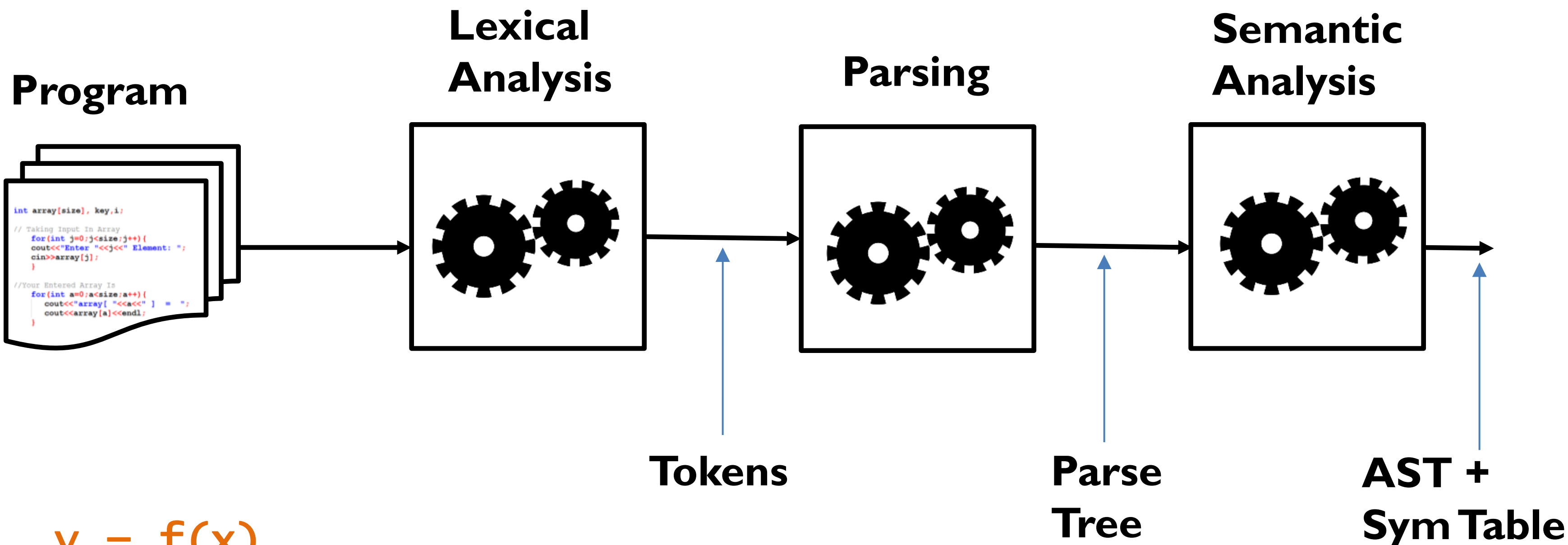
Compiler Overview



Role of a frontend

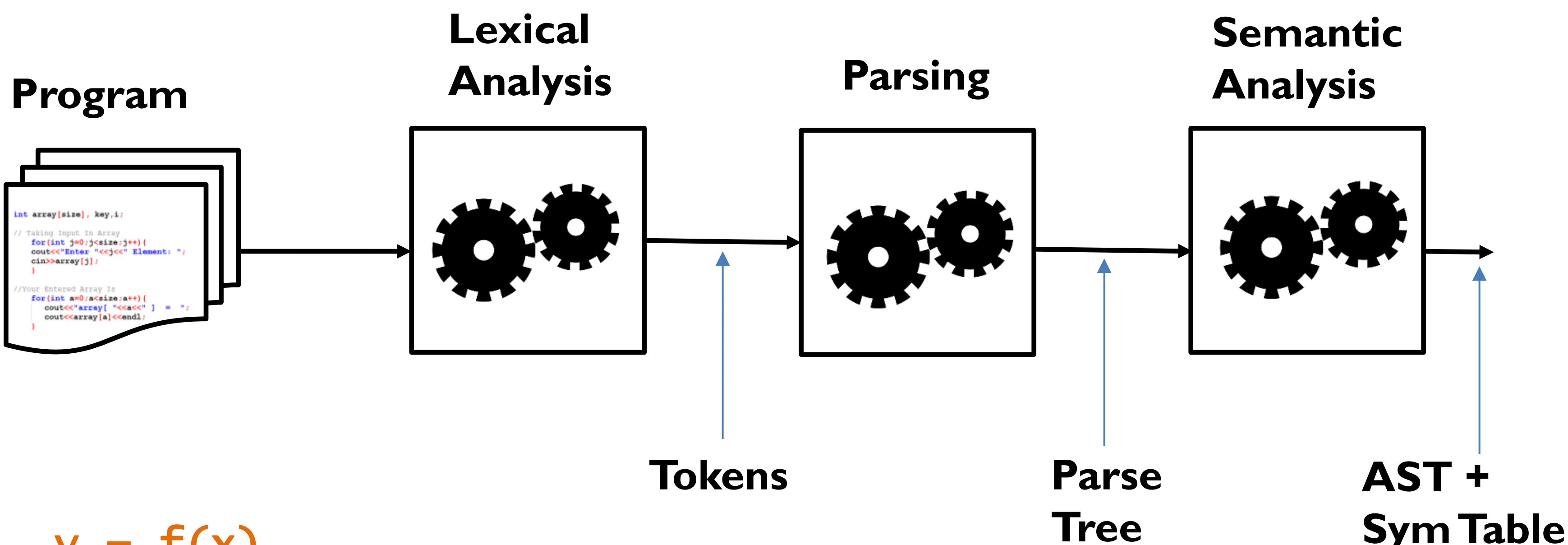


Role of a frontend



y = f(x)

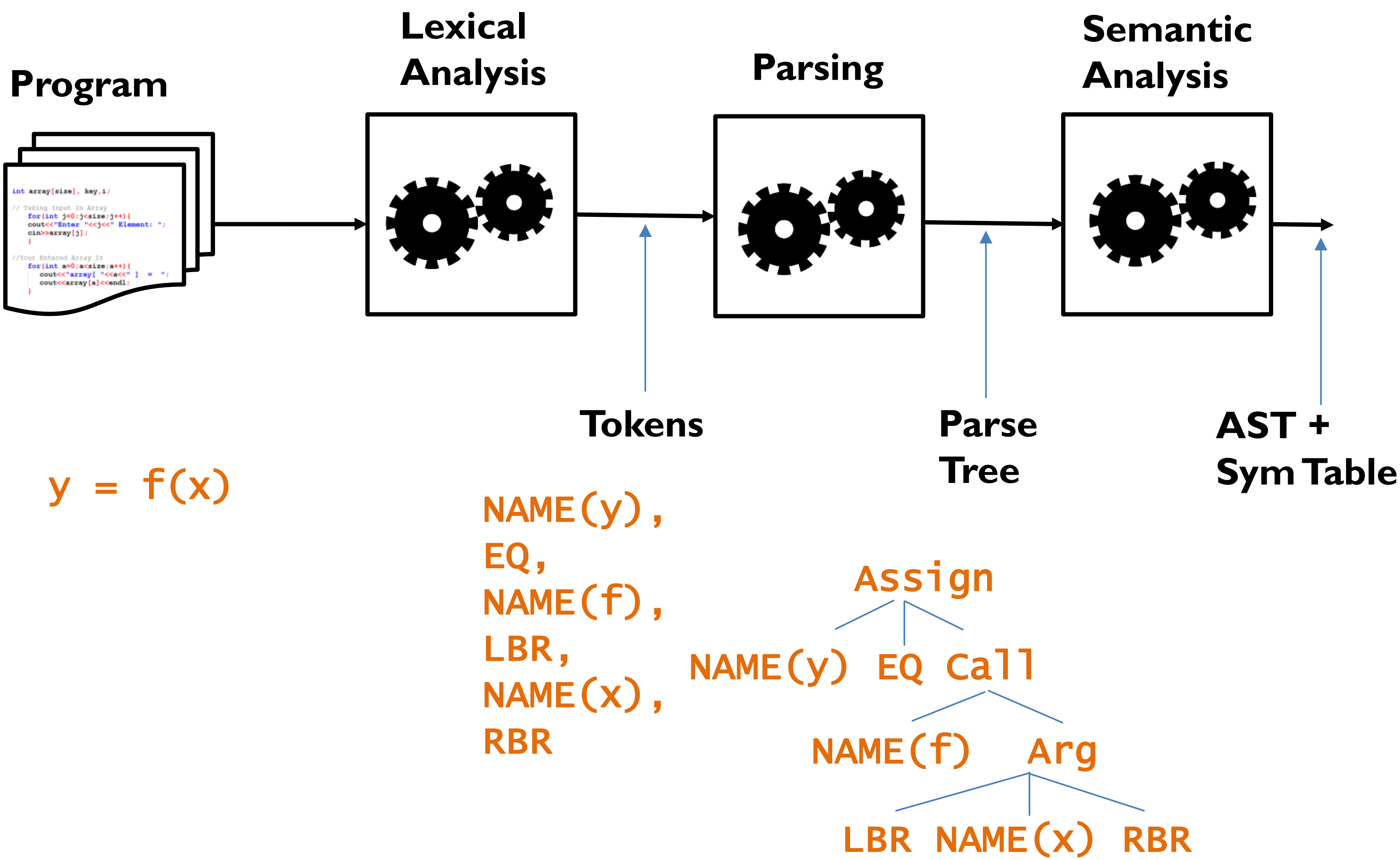
Role of a frontend



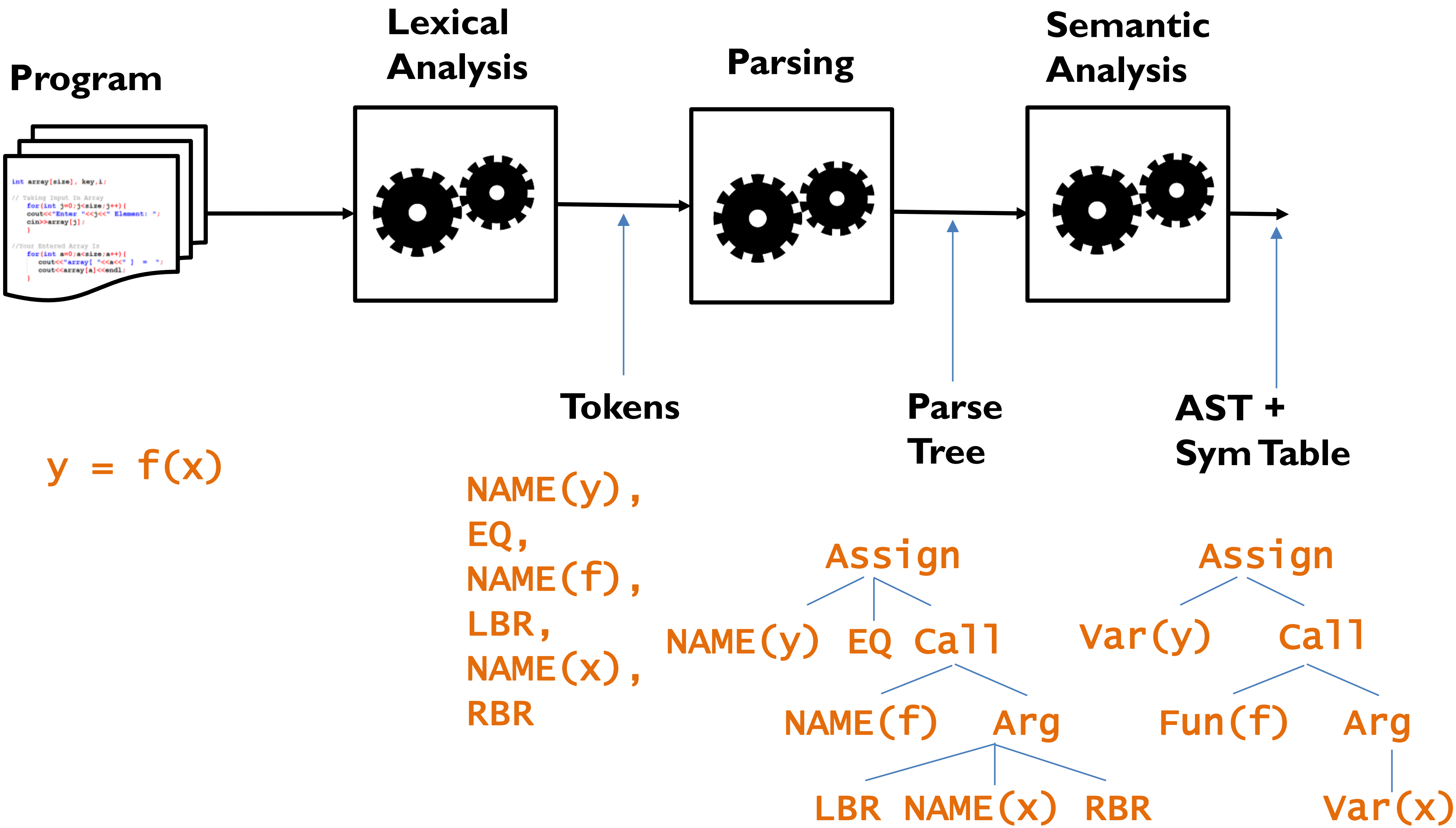

```
y = f(x)
```

```
NAME(y),
EQ,
NAME(f),
LBR,
NAME(x),
RBR
```

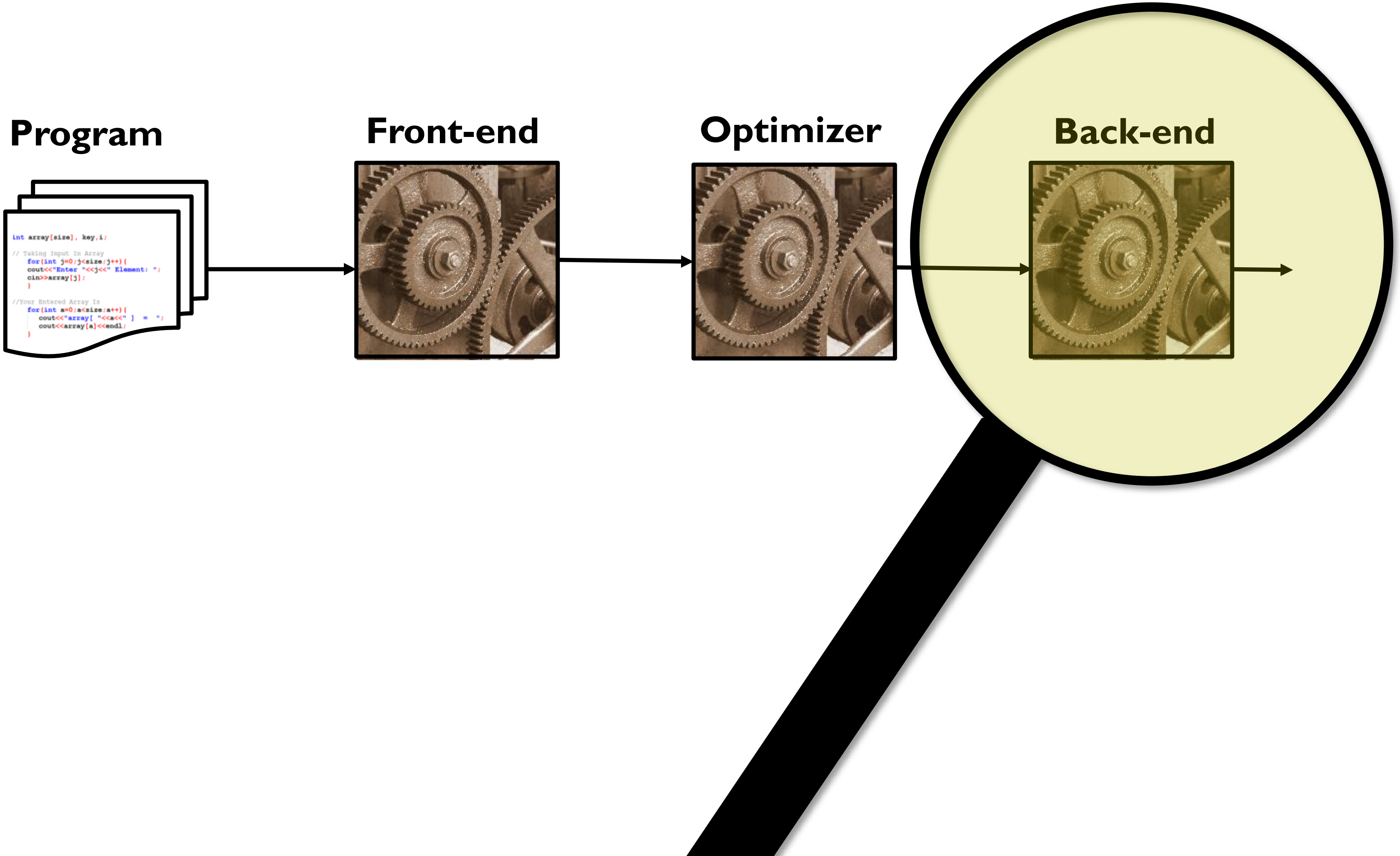
Role of a frontend



Role of a frontend

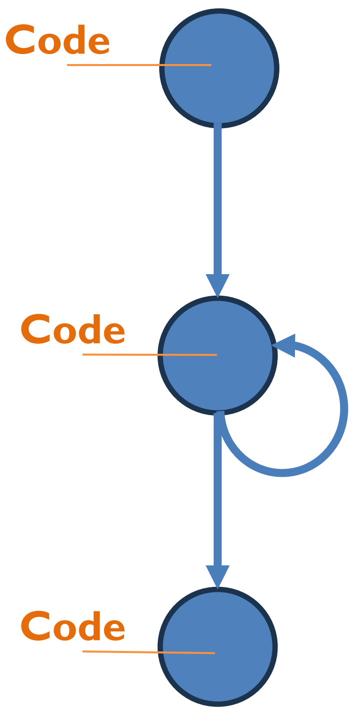


Compiler Overview



Role of a backend

Intermediate Representation



Assembly Language

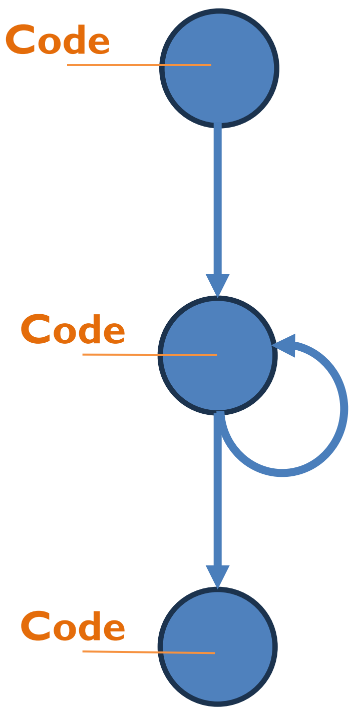
```
sumcalc(int, int, int):
        mov     ecx, edx
        lea     eax, [4*rdi]
        cdq
        idiv    esi
        test    ecx, ecx
        js      .LBB2_1
        lea     edx, [rax + 4]
        imul    edx, ecx
        add     eax, 5
        mov     esi, ecx
        lea     edi, [rcx - 1]
        imul    rdi, rsi
        add     ecx, -2
        imul    rcx, rdi
        shr     rdi
        imul    edi, eax
        add     edi, edx
        shr     rcx
        imul    eax, ecx, 1431655766
        add     eax, edi
        add     eax, 1
        ret
.LBB2_1:
        xor     eax, eax
        ret
```

Binary Code

```
000100011001
010011001010
101001010101
001010001011
110100101001
010100101010
010101001010
010101001010
010101001010
100010010101
001010000000
101011111110
001010010101
011110001100
000100011001
010011001010
101001010101
001010001011
110100101001
010100101010
010101001010
010101001010
010101001010
100010010101
001010000000
101011111110
001010010101
```

Role of a backend

Intermediate Representation



Interpreter Language

```
define i32 @sumcalc(int, int, int)
  (i32 noundef %0, i32 noundef %1, i32 noundef %2) {
  %6 = icmp slt i32 %2, 0
  br i1 %6, label %27, label %7

7:%8 = add i32 %5, 4,
  %9 = mul i32 %8, %2,
  %10 = add i32 %5, 5,
  %11 = zext i32 %2 to i33,
  %12 = add nsw i32 %2, -1,
  %13 = zext i32 %12 to i33,
  %14 = mul i33 %11, %13,
  %15 = lshr i33 %14, 1,
  %16 = trunc i33 %15 to i32,
  br label %16,

17:%18 = phi i32 [0, %3], [%16, %7]
  ret i32 %18,
}
```

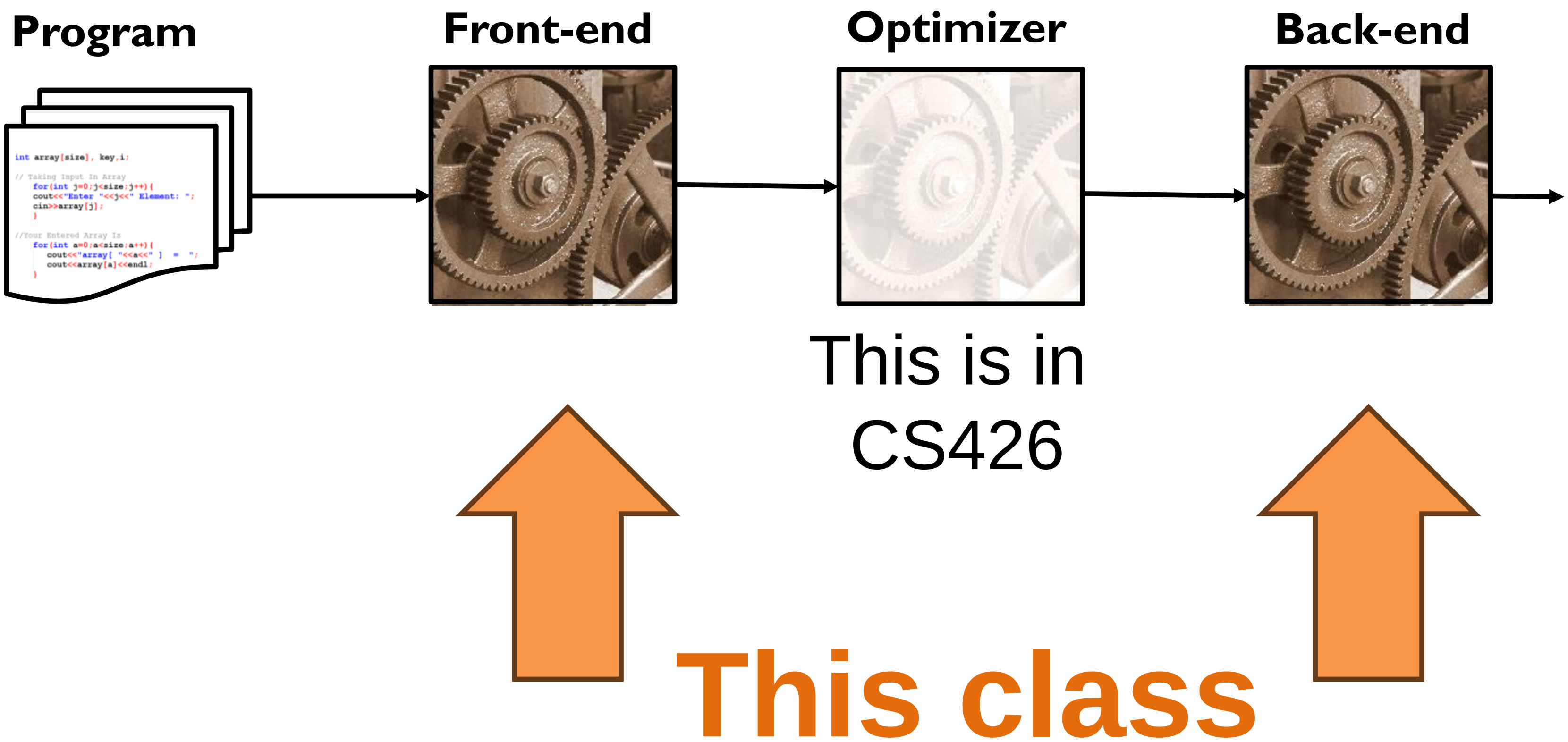
Runtime System

Interpreter executes those intermediate commands

Direct Eval

result1
result2
result3
result4
result5
result6
result7
result8
result8
result10
result11
result12
result13
result14
result15
result16
result17
result18
result19
result20
result21
….

Compiler Overview



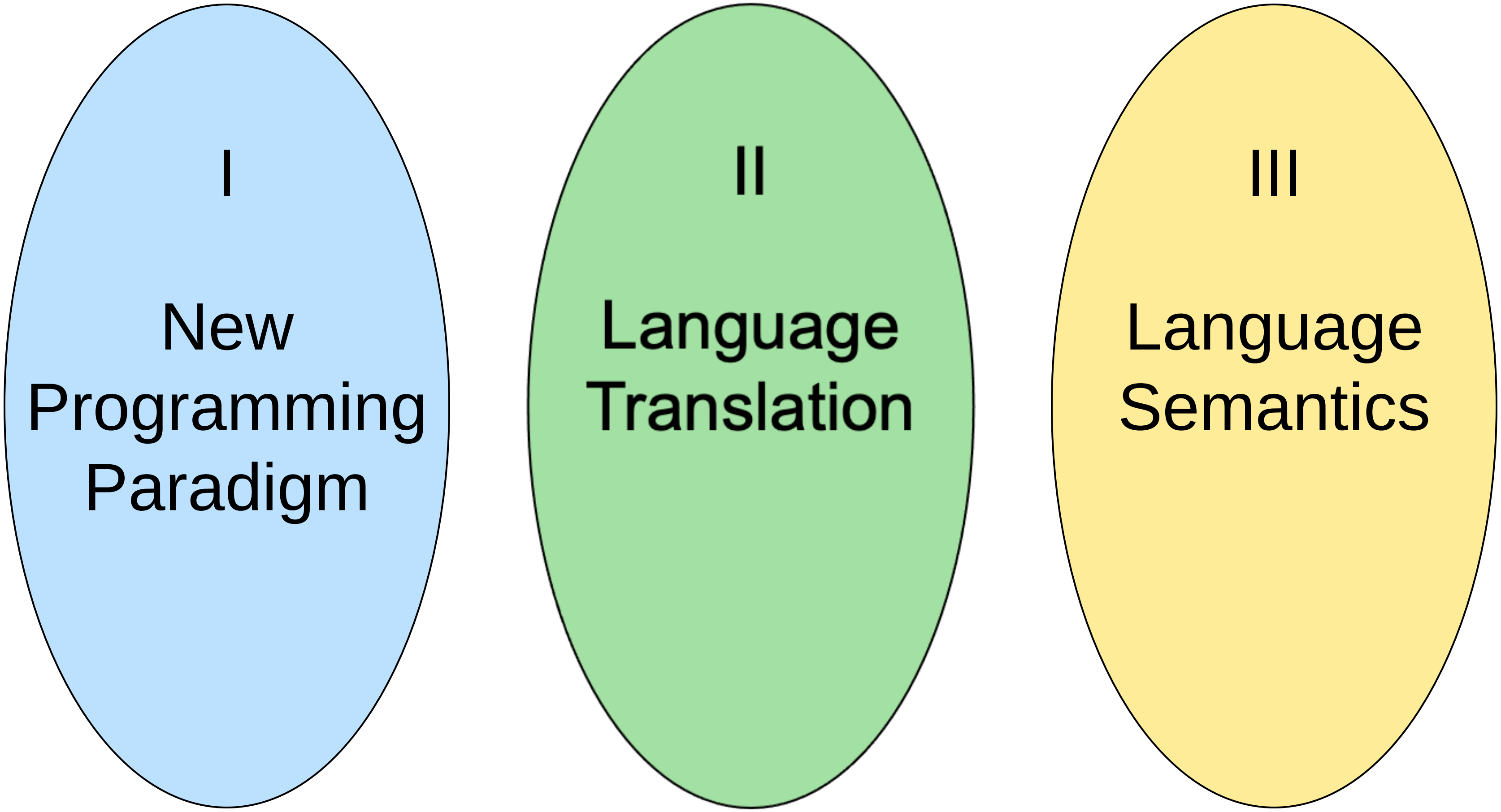
My Background

- Grew up writing code in C, C++ and Java
- Did my undergrad in ECE
- Explored full stack of programming systems in my PhD
- Interests: Programming systems for fast and accurate computing across the system stack, recently with a focus on ML/AI applications

- Courses taught: Basic and Advanced Compilers (in LLVM; cs 426 and cs526), Formal methods (cs477), Approximate and probabilistic programming systems (cs521), …

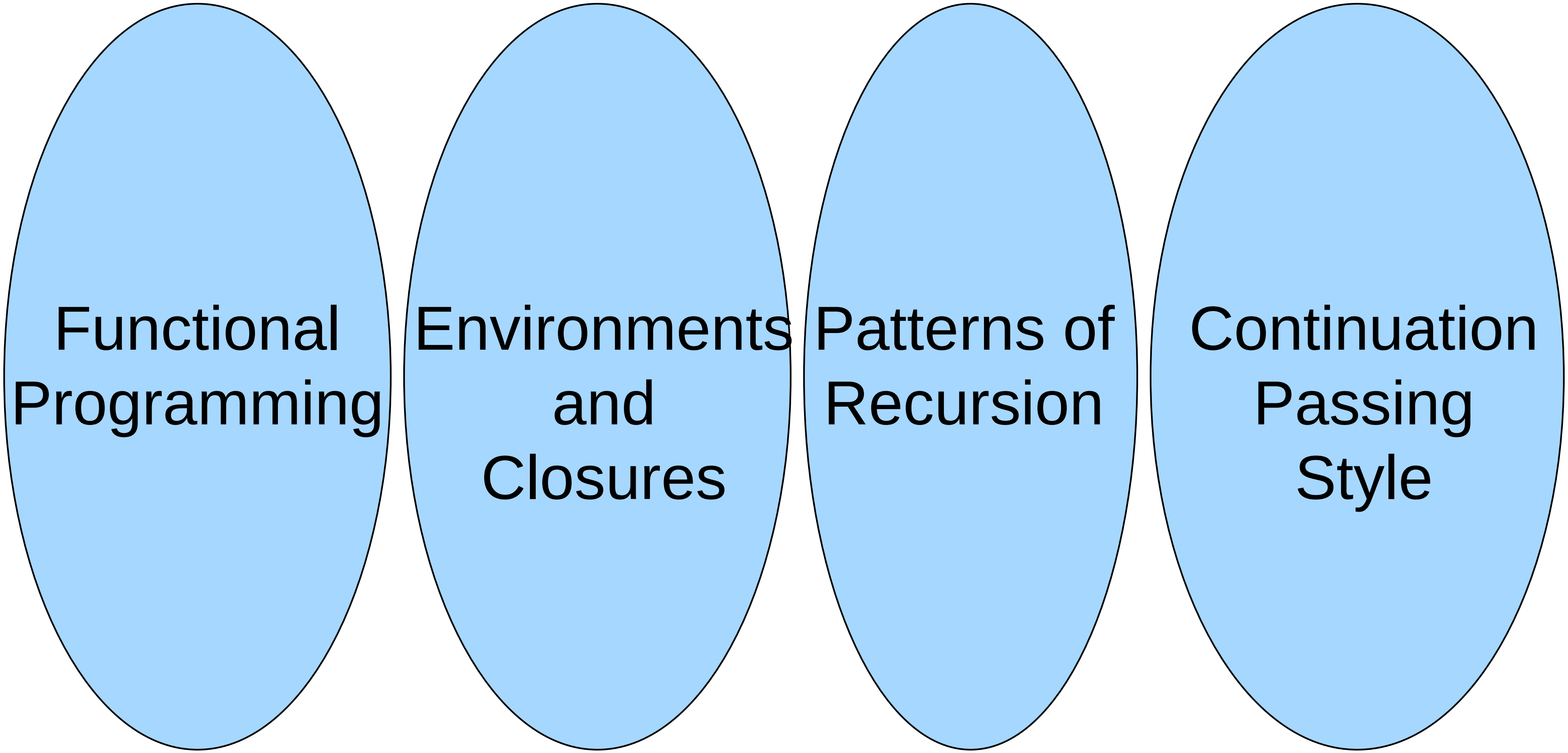
Programming Languages & Compilers

Three Main Topics of the Course



Programming Languages & Compilers

I : New Programming Paradigm



Programming Languages & Compilers

I : New Programming Paradigm

Functional Programming

Different discipline of programming from what you're used to:

- Immutable Program State
  - Yes!   x = y + 1
  - NO!   x = x + 1

- Functions are first-class citizens
  - Pass as arguments to other functions, manipulate as objects

Programming Languages & Compilers

I : New Programming Paradigm

Environments and Closures

Program state we are used to:

- Variables located on stack or heap

We will learn:

- Make the notion of program state more flexible

Programming Languages & Compilers

I : New Programming Paradigm

Patterns of Recursion

Iteration we are used to:

- for/while Loops
- iterators

We will learn:

- How to make functions calling themselves efficient
- Pattern matching

Programming Languages & Compilers

I : New Programming Paradigm

Continuation Passing Style

Program counter we are used to:

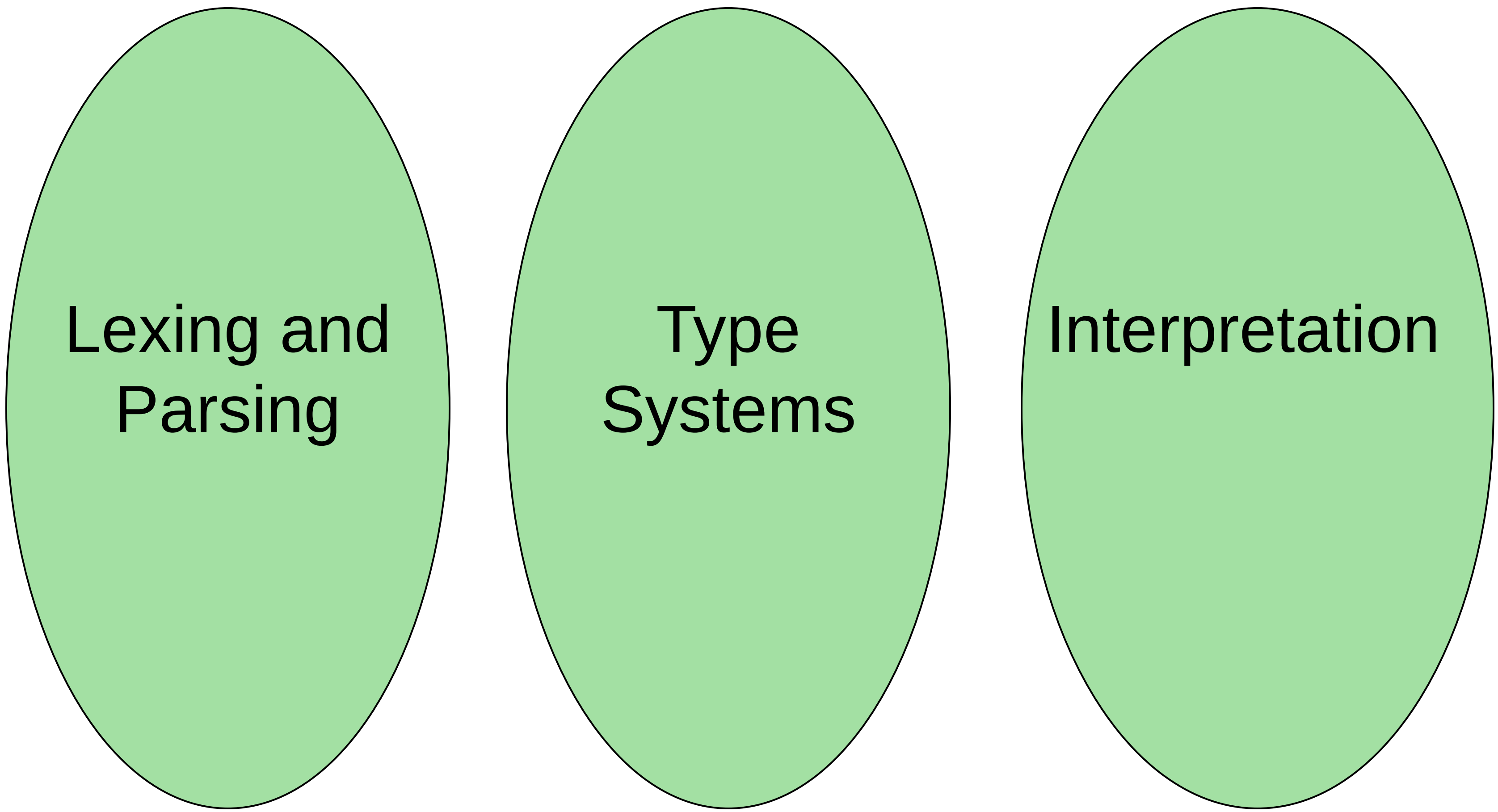
- The pointer to the next instruction to execute
- Function pointer or a label to make jumps

We will learn:

- How to abstract the notion of program counter and location where the execution continues

Programming Languages & Compilers

II : Language Translation

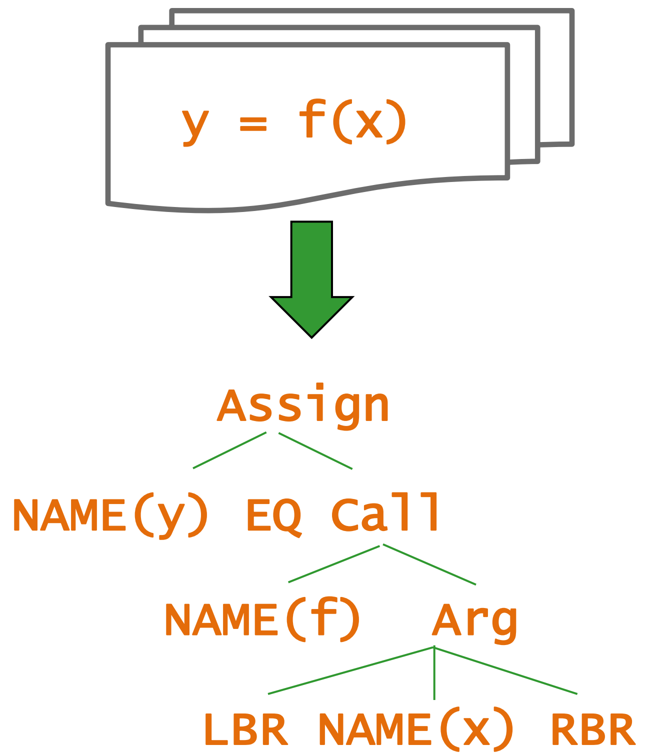


Programming Languages & Compilers

II : Language Translation

Lexing and Parsing

From program text to a data structure we can manipulate:



Programming Languages & Compilers

II : Language Translation

Type Systems

Is this a legal program:

int x = 3

int y = x * 2

int z = y + “one”

How to automatically check?

Programming Languages & Compilers

II : Language Translation

Interpretation

How do we make this text run on the machine?

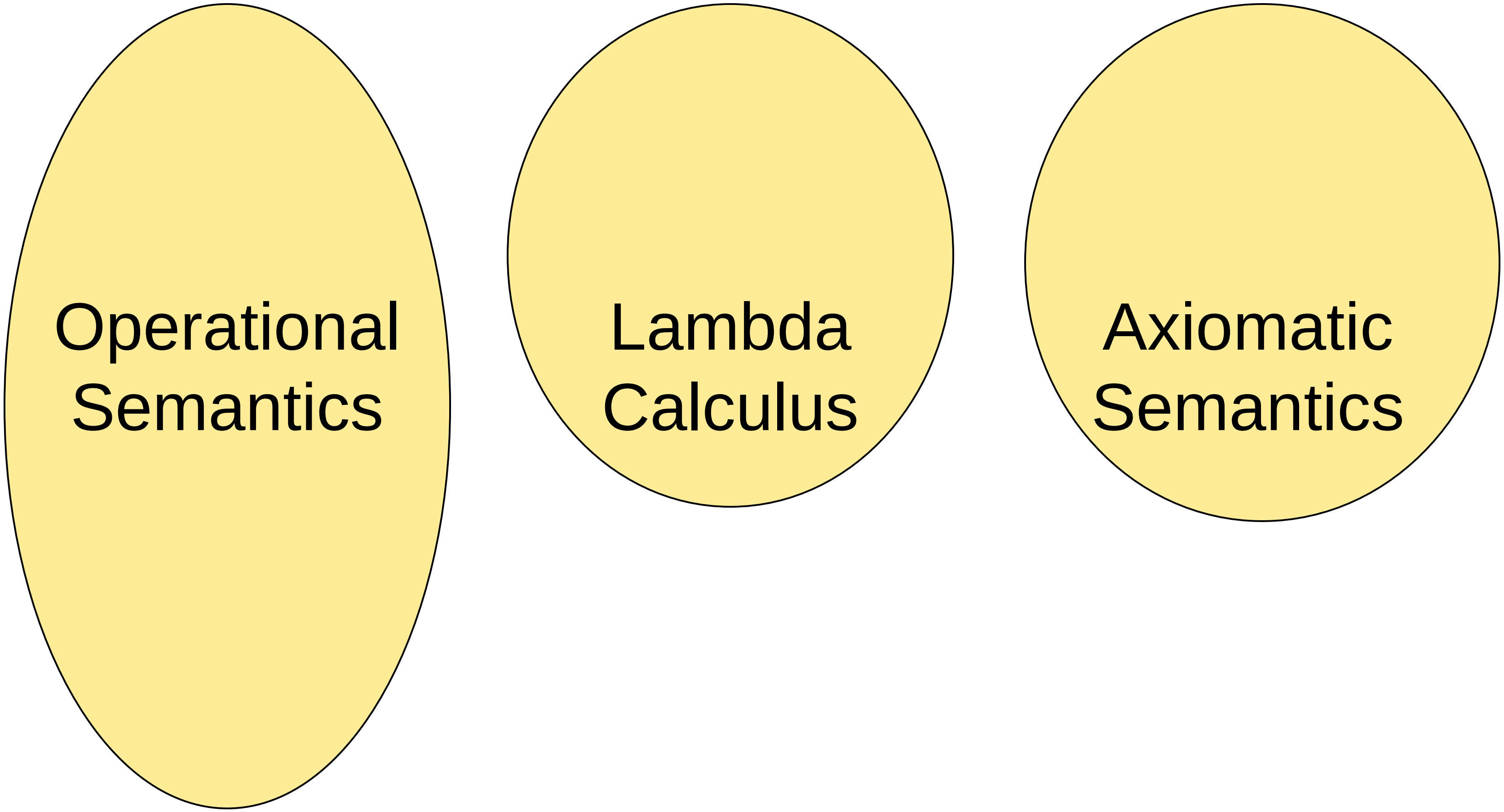
X: 0
Y: 0

```
int x = 3

int y = x * 2
```

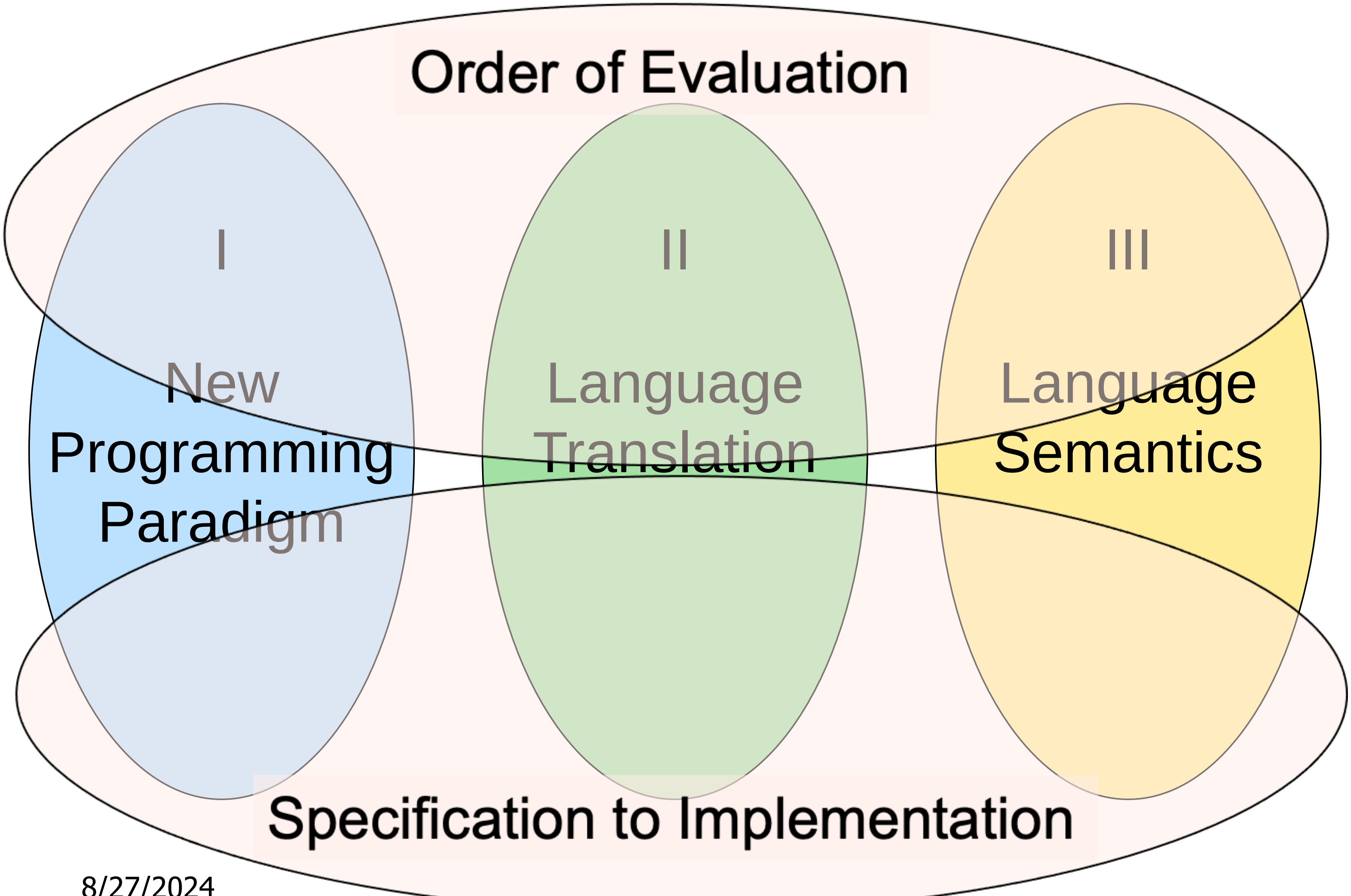
X: 3
Y: 6

Programming Languages & Compilers

III : Language Semantics



Programming Languages & Compilers



Beyond Class

- Functional Programming is used in industry

Jane Street® OCaml: High-Frequency Trading

https://blog.janestreet.com/why-ocaml/

https://ocaml.org/industrial-users

facebook Microsoft docker
Bloomberg ahrefs

Microsoft Haskell
F#

https://wiki.haskell.org/Haskell_in_industry

https://github.com/fsprojects/fsharp-companies

ERICSSON Erlang: Language for High-reliability concurrent systems

EPIC GAMES Verse: Language for XR based on Haskell

Beyond Class

- Occasionally we will have several slides that are not part of the exam, but give a broader context about programming
  - Functional patterns in imperative languages
  - Language design choices
  - Practical use cases and advanced theory
- Our friend Zafar the Camel will lead us into these stories. (Zafar is made by Dall-E)

Example

- Most of today's languages are multi-paradigm (Imperative, OOP, Functional, Concurrent…)
- You can write C/C++ programs in functional styles
- You can also write Ocaml in imperative style (but not in this class)

Contact Information: Sasa Misailovic

- Office: 4110 SC
- Office hours:
  - Urbana: Tuesday 9:45am – 10:30am
  - Chicago: Thursday 1:45pm – 2:30pm
    - Can attend on zoom
  - Also by appointment
- Email: misailo@illinois.edu
  - Use Campuswire

Course Sections

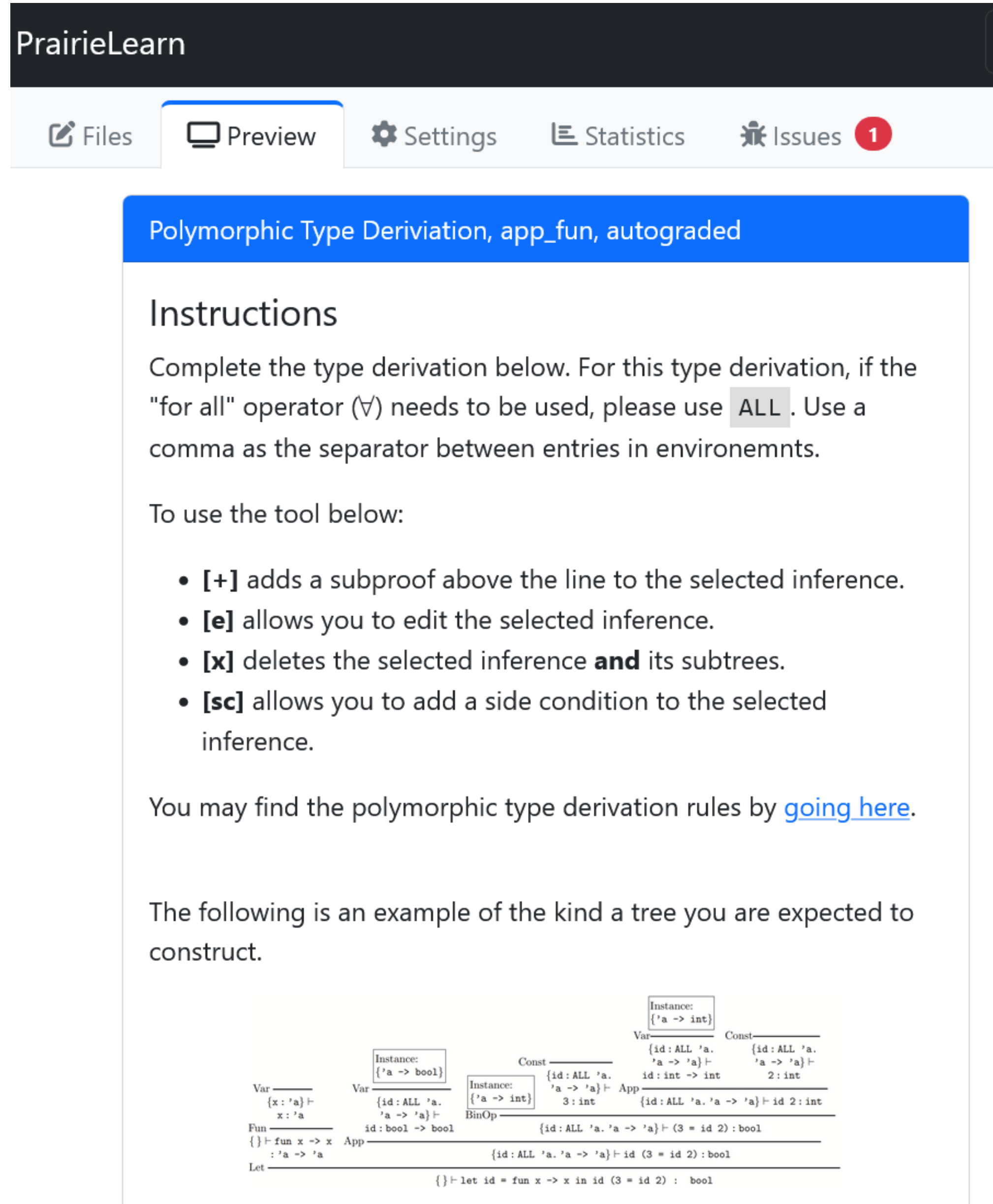
- Sasa’s section (CS421C): 12:30 – 1:45
  - Tuesdays in Urbana and Zoom
  - Thursdays in Chicago and Zoom
    - Urbana students can still use the classroom

- Elsa’s section (CS421D):
  - Tuesday, Thursday in Urbana

- Both sections cover the same material for the exam, in the same order
  - The midterms/exams are the same
  - All lectures are recorder

Course Infrastructure

- Discuss: Campuswire
- Exams: PrairieLearn and PrairieTest
- Autogenerated exams and autograding
  - Big shout-out to Elsa!

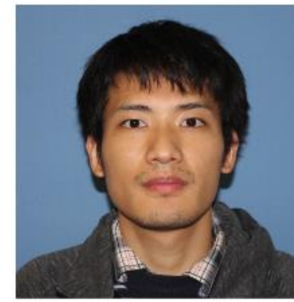


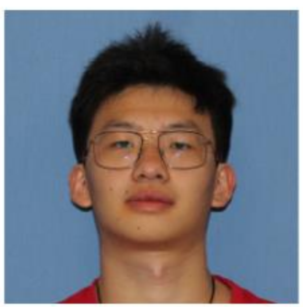
Course TAs


Shams Alshabani


Athena Fung

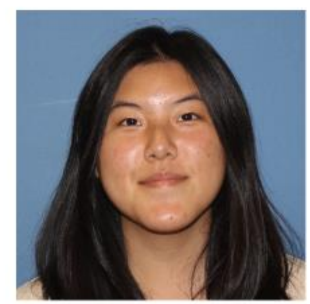

Paul Krogmeier


Yerong Li


Siheng Pan


Cody Rivera


Uche Uche-Ike

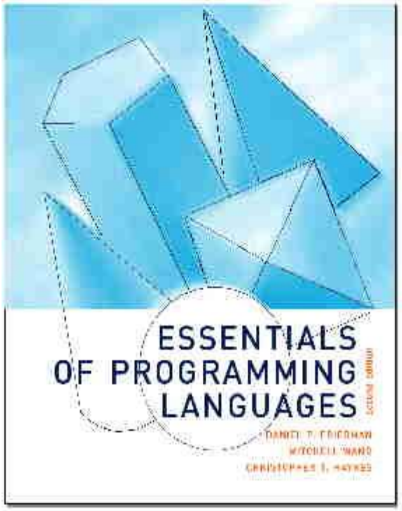

Allison Ye

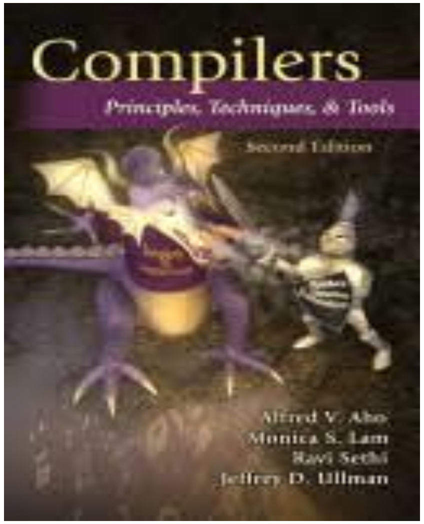
Course Website

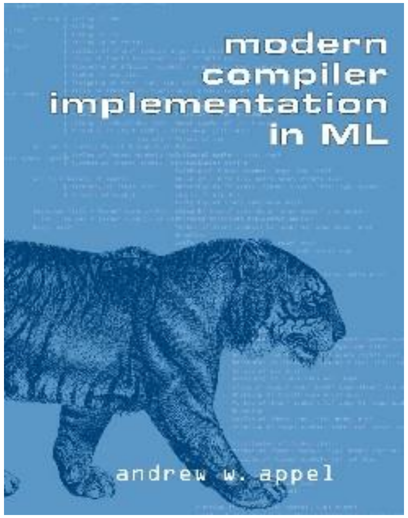
- https://courses.engr.illinois.edu/cs421/fa2024/CS421C
- Main page - summary of news items
- Policy - rules governing course
- Lectures - syllabus and slides
- MPs - information about assignments
- Exams – Syllabi and review material for Midterms and finals
- Unit Projects - for 4 credit students
- Resources - tools and helpful info
- FAQ

Some Course References

- No required textbook
- Some suggested references







Some Course References

- No required textbook.
- Pictures of the books on previous slide
- Essentials of Programming Languages (2nd Edition) by Daniel P. Friedman, Mitchell Wand and Christopher T. Haynes, MIT Press 2001.
- Compilers: Principles, Techniques, and Tools, (also known as "The Dragon Book"); by Aho, Sethi, and Ullman. Published by Addison-Wesley. ISBN: 0-201-10088-6.
- Modern Compiler Implementation in ML by Andrew W. Appel, Cambridge University Press 1998
- Additional ones for Ocaml given separately

Course Grading

- Assignments
  - Web Assignments (WA)
  - MPs (in Ocaml)
  - All WAs and MPs Submitted by **PrairieLearn**
  - Late submission penalty: capped at 80% of total
- Quizzes
  - 3 quizzes (20min) done in class: same as MP
- Midterms
  - 3 midterms: All done in CBTF (Urbana/Chicago)
- Final
  - Done in CBTF (Urbana/Chicago)

Course Grading

Grading Breakdown		
Work	**Weight (3cr)**	**Weight (4cr)**
Machine Problems, and Web Assignments (combined)	14%	10.5%
Quizzes	6%	4.5%
Midterm 1	15%	11.25%
Midterm 2	15%	11.25%
Midterm 3	15%	11.25%
Final Exam	35%	26.25%
Project	NA	25%

Grading Scale	
Grade	If your overall score is at least
A+	100%
A	93%
A-	90%
B+	87%
B	83%
B-	80%
C+	77%
C	73%
C-	70%
D+	67%
D	63%
D-	60%

Course Assingments – WA & MP

- You may discuss assignments and their solutions with others
  - You may work in groups, but you must **list members with whom you worked** if you share solutions or solution outlines
- **Each student must write up and turn in their own solution separately**
- You may look at examples from class and other similar examples from any source – **cite appropriately**
  - **This includes LLMs! (although you shouldn't use them)**
  - Note: University policy on plagiarism still holds - cite your sources if you are not the sole author of your solution
  - Do not have to cite course notes or me

OCAML

- Locally:
  - Will use ocaml inside VSCode inside PrairieLearn problems this semester
- Globally:
  - Main OCAML home: http://ocaml.org
  - To install OCAML on your computer see: http://ocaml.org/docs/install.html
  - For Windows: just install WSL and then do Linux https://learn.microsoft.com/en-us/windows/wsl/install

- **To try on the web:** https://try.ocamlpro.com

References for OCaml

- Supplemental texts (not required):
- The Objective Caml system release 4.05, by Xavier Leroy, online manual
- Introduction to the Objective Caml Programming Language, by Jason Hickey
- Developing Applications With Objective Caml, by Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano, on O'Reilly
  - Available online from course resources

OCAML Background

- CAML is European descendant of original ML
  - American/British version is SML
  - O is for object-oriented extension
- ML stands for Meta-Language
- ML family designed for implementing theorem provers
  - It was the meta-language for programming the "object" language of the theorem prover
  - Despite obscure original application area, OCAML is a full general-purpose programming language

Features of OCAML

- Higher order applicative language
- Call-by-value parameter passing
- Modern syntax for functional languages
- Parametric polymorphism
  - Aka structural polymorphism
- Automatic garbage collection
- User-defined algebraic data types

Session in OCAML

```
% ocaml
Objective Caml version 4.07.1
# (* Read-eval-print loop; expressions and
  declarations *)
   2 + 3;;        (* Expression *)
- : int = 5
# 3 < 2;;
- : bool = false
```

Declarations; Sequencing of Declarations

```
# let x = 2 + 3;;    (* declaration *)
val x : int = 5

# let test = 3 < 2;;
val test : bool = false

# let a = 1 let b = a + 4;; (* Sequence of dec *)
val a : int = 1
val b : int = 5
```

Functions

```
# let plus_two n = n + 2;;
val plus_two : int -> int = <fun>

# plus_two 17;;
- : int = 19
```

Functions

```
let plus_two n = n + 2;;

plus_two 17;;
- : int = 19
```

Fun with Functions

```
# let plus_two n = n + 2;;
val plus_two : int -> int = <fun>

# plus_two 17;;
- : int = 19

# let just_plus n m = n + m;;
val just_plus : int -> int -> int = <fun>

# just_plus 17
- : int -> int = <fun>
```

What happens next?

Fun with Functions

```
# let just_plus n m = n + m;;
val just_plus : int -> int -> int = <fun>

# let plus17 = just_plus 17
val plus17 : int -> int = <fun>

# plus17 2
- : int = 19
```

No Overloading for Basic Arithmetic Operations

```
# 15 * 2;;
- : int = 30

# 1.35 + 0.23;;  (* Wrong type of addition *)
Characters 0-4:
  1.35 + 0.23;;  (* Wrong type of addition *)
  ^^^^
Error: This expression has type float but an
   expression was expected of type
          int

# 1.35 +. 0.23;;
- : float = 1.58
```

Sequencing Expressions

```
# "Hi there";;  (* has type string *)
- : string = "Hi there"
# print_string "Hello world\n";;  (* has type unit *)
Hello world
- : unit = ()
# (print_string "Bye\n"; 25);;  (* Sequence of exp *)
Bye
- : int = 25
```

Recursive Functions

```
# let rec factorial n =
    if n = 0 then 1
    else n * factorial (n - 1);;
  val factorial : int -> int = <fun>

# factorial 5;;
- : int = 120
```

“rec” is needed for recursive function declarations

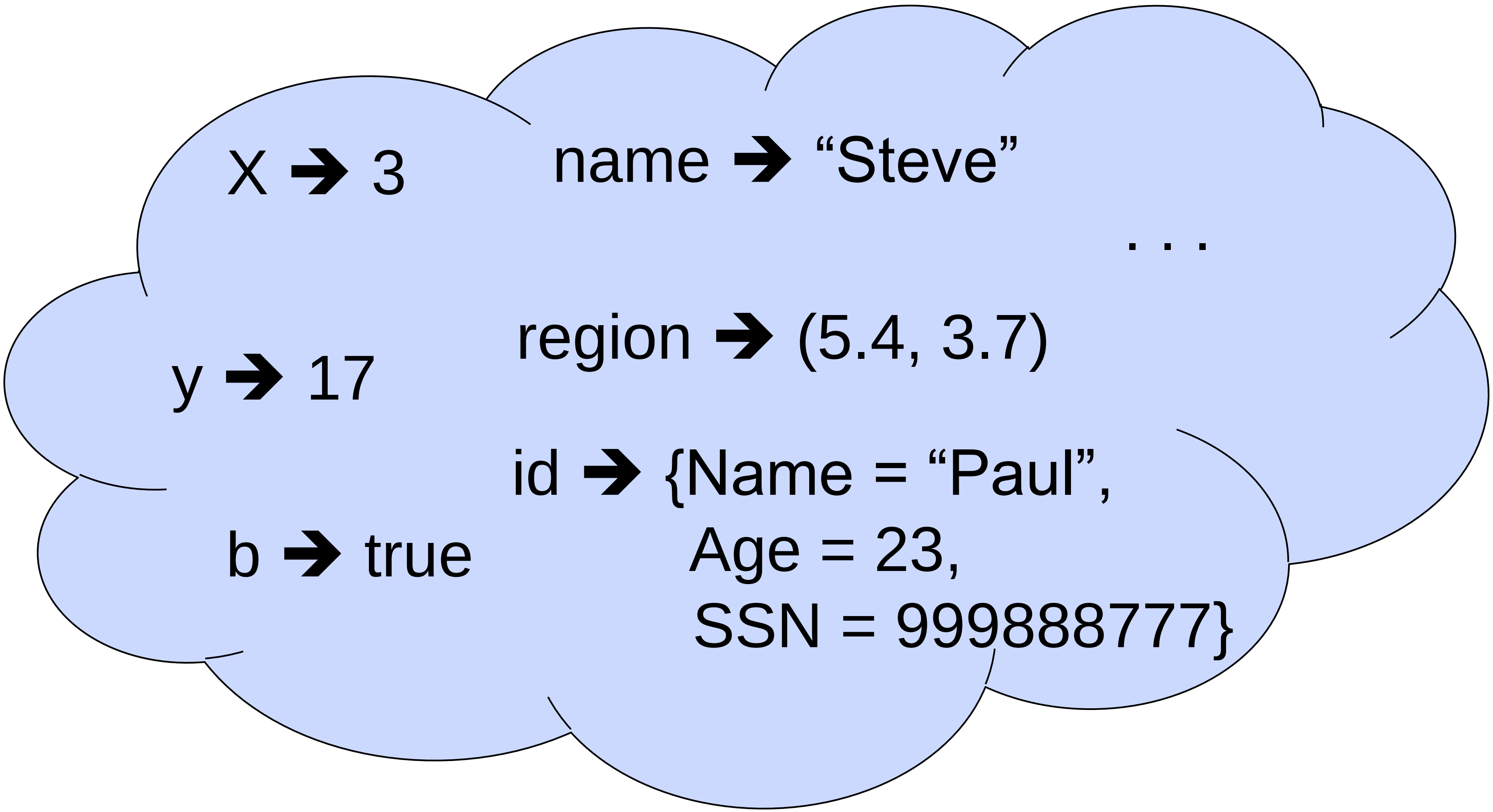
Environments

- **Environments** record what value is associated with a given identifier
- Central to the semantics and implementation of a language
- Notation

  $$\rho = \{name_1 \rightarrow value_1, name_2 \rightarrow value_2, \ldots\}$$

  Using set notation, but describes a partial function
- Often stored as list, or stack
  - To find value start from left and take first match

Environments



Global Variable Creation

2 + 3;; (* Expression *)

// doesn't affect the environment

let test = 3 < 2;; (* Declaration *)

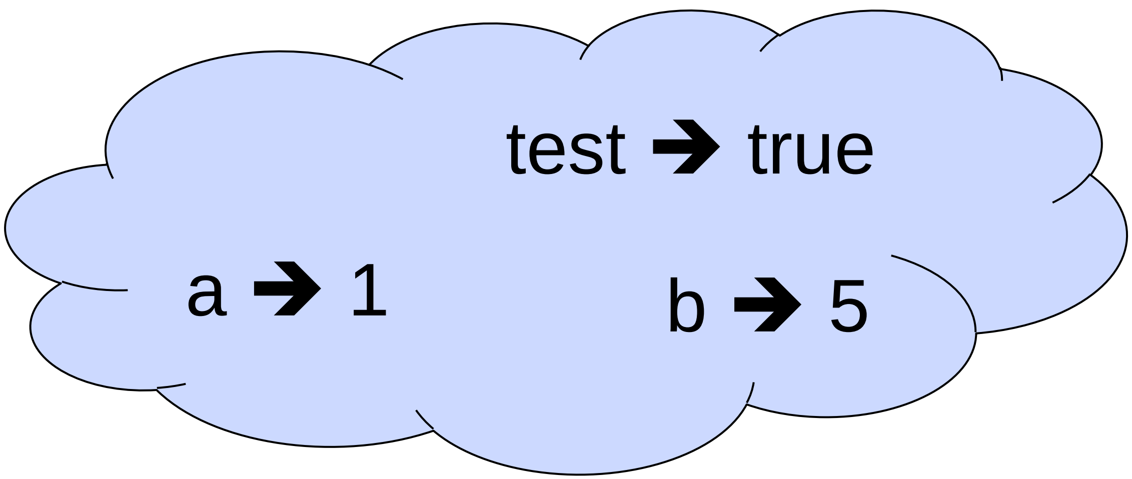
val test : bool = false

// ρ_1 = {test → false}

let a = 1 let b = a + 4;; (* Seq of dec *)

// ρ_2 = {b → 5, a → 1, test → false}

Environments



New Bindings Hide Old

// $\rho_2 = \{b \rightarrow 5, a \rightarrow 1, test \rightarrow false\}$

```
let test = 3.7;;
```

- What is the environment after this declaration?

New Bindings Hide Old

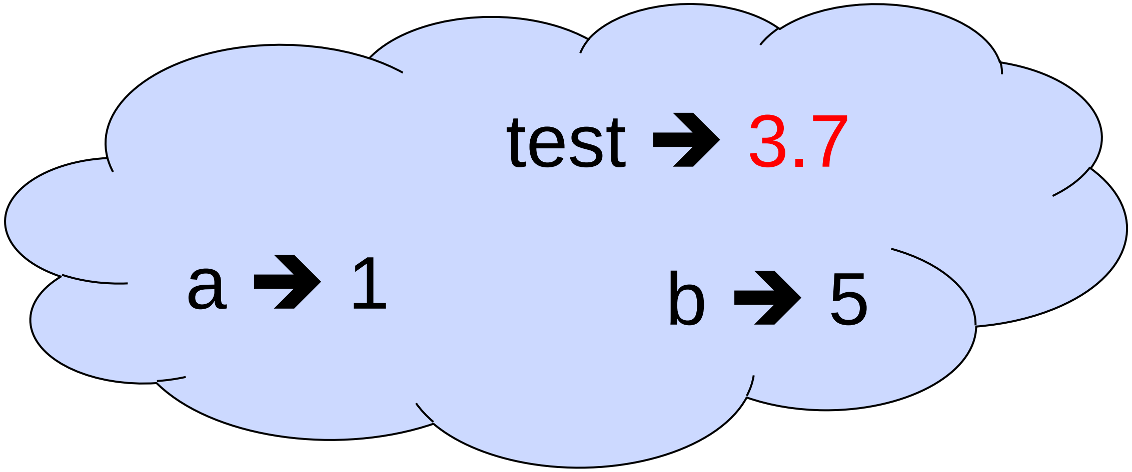
// $\rho_2 = \{b \rightarrow 5, a \rightarrow 1, test \rightarrow false\}$

```
let test = 3.7;;
```

- What is the environment after this declaration?

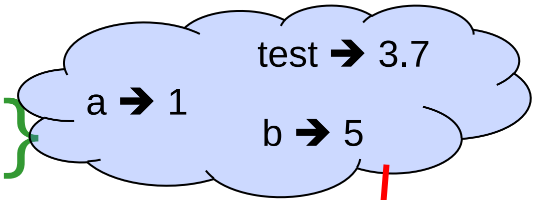
// $\rho_3 = \{test \rightarrow 3.7, a \rightarrow 1, b \rightarrow 5\}$

Environments



Local Variable Creation

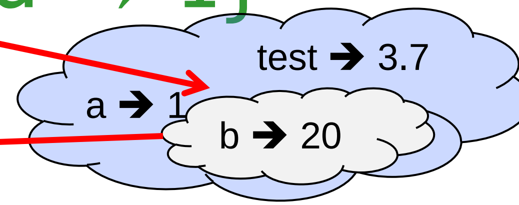
// ρ_3 = {test → 3.7, a → 1, b → 5}



let b = 5 * 4

// ρ_4 = {b → 20, test → 3.7, a → 1}

in 2 * b;;

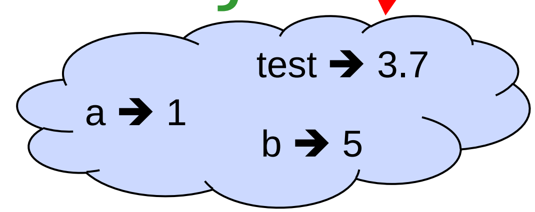


- : int = 40

// ρ_5 = ρ_3= {test → 3.7, a → 1, b → 5}

b;;

- : int = 5

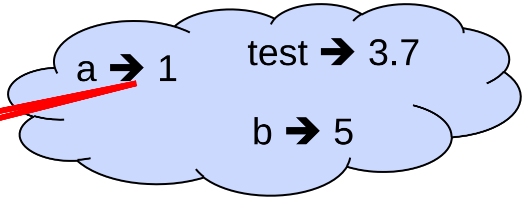


Local let binding

// ρ_5 = {test $\rightarrow$ 3.7, a $\rightarrow$ 1, b $\rightarrow$ 5}

```
# let c =
    let b = a + a



    in b * b;;



# b;;
```

Local let binding



// ρ_5 = {test → 3.7, a → 1, b → 5}

\# let c =

let b = a + a

// ρ_6 = {b → 2} + ρ_5

// = {b → 2, test → 3.7, a → 1}

in b * b;;

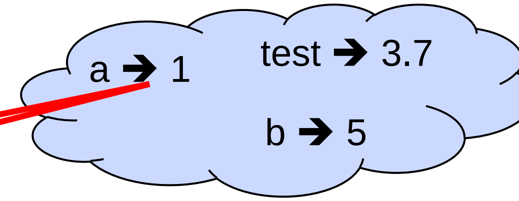
val c : int = 4

// ρ_7 = {c → 4, test → 3.7, a → 1, b → 5}

\# b;;

\- : int = 5

Local let binding

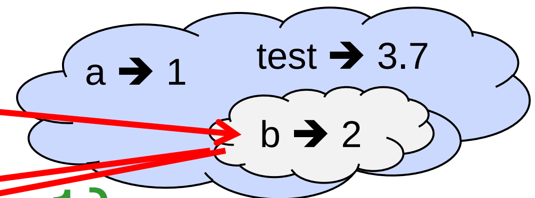


// $\rho_5 = \{test \rightarrow 3.7,\ a \rightarrow 1,\ b \rightarrow 5\}$

```
# let c =
    let b = a + a
```

// $\rho_6 = \{b \rightarrow 2\} + \rho_5$

// $\quad = \{b \rightarrow 2,\ test \rightarrow 3.7,\ a \rightarrow 1\}$

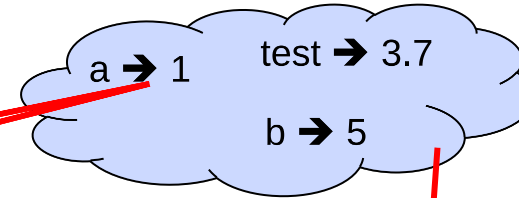



```
    in b * b;;
val c : int = 4
```

// $\rho_7 = \{c \rightarrow 4,\ test \rightarrow 3.7,\ a \rightarrow 1,\ b \rightarrow 5\}$

```
# b;;
- : int = 5
```

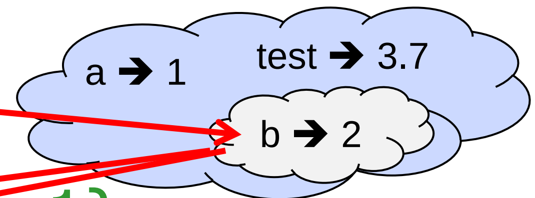
Local let binding



// ρ_5 = {test → 3.7, a → 1, b → 5}

let c =

    let b = a + a

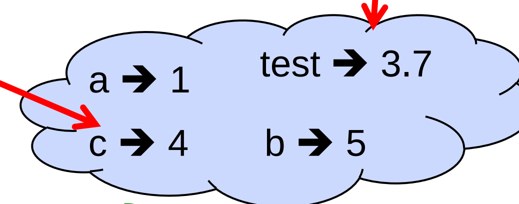


// ρ_6 = {b → 2} + ρ_5

//   = {b → 2, test → 3.7, a → 1}

    in b * b;;

val c : int = 4



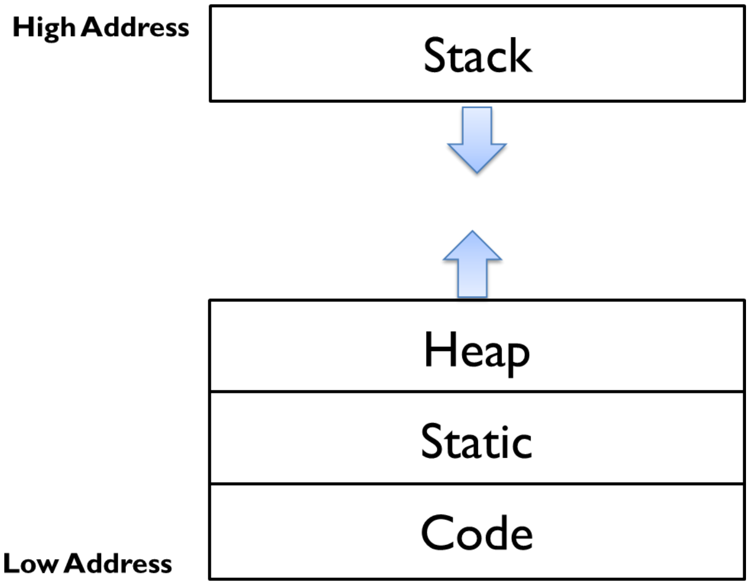
// ρ_7 = {c → 4, test → 3.7, a → 1, b → 5}

b;;

- : int = 5

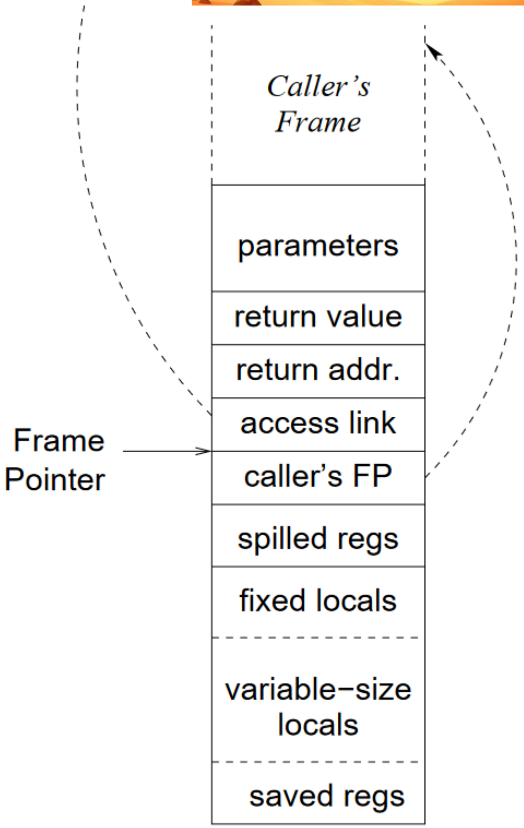
Concrete Environments: System Memory

- In systems programming, memory is often divided in different parts:



Stack Frame (reminder)

- In x86 stack frame contains:
  - At run-time, each procedure invocation has an associated **local storage**.
  - For many languages, this local storage can live on the stack, and then they are also called **stack frames**.
  - Variable names map to the offsets from the frame pointer (FP)



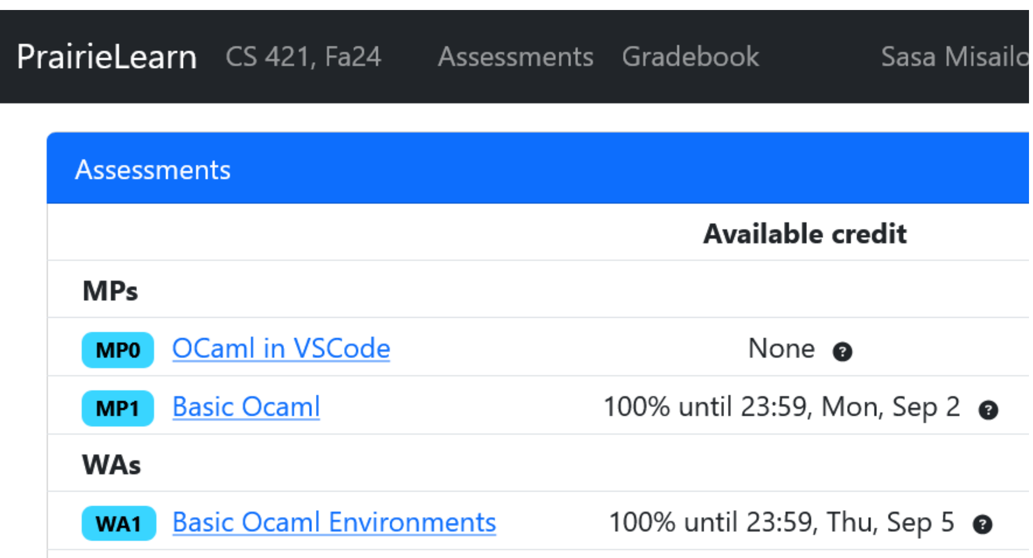
Environment vs Stack Frame

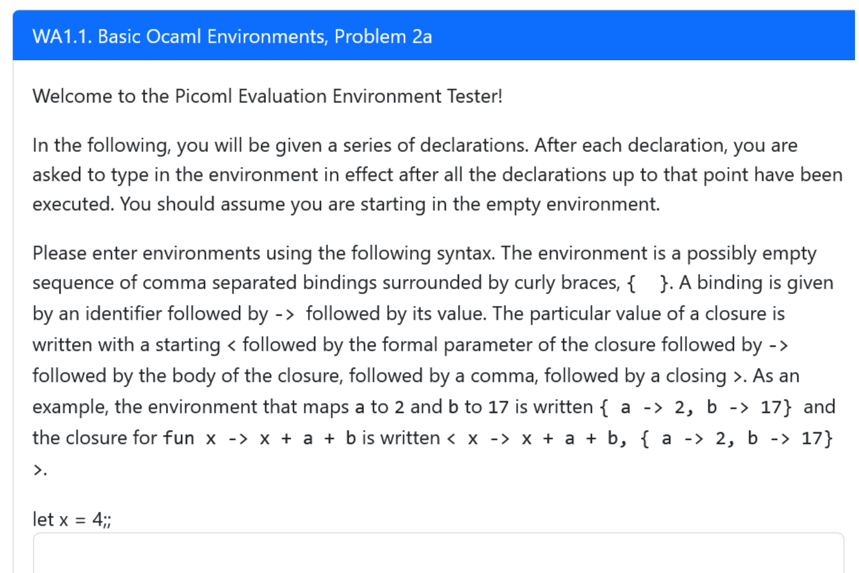
- Environment is abstraction (language level) Stack frame is implementation (system level)
- Semantics of Environment: copied after the statements; only updated variables changed
  - Becomes more interesting when we need to think about functions
- Semantics of Stack frame: updated in-place

Let’s think about high-level expressivity, not efficiency at the moment!

Now it's your turn

You should be able to do WA1-IC Problem 1 , parts (* 1 *) - (* 3 *)





Functions

```
# let plus_two n = n + 2;;
val plus_two : int -> int = <fun>

# plus_two 17;;
- : int = 19
```

Functions

```
let plus_two n = n + 2;;

plus_two 17;;

- : int = 19
```

Nameless Functions (aka Lambda Terms)

```
fun n -> n + 2;;

(fun n -> n + 2) 17;;
- : int = 19
```

Functions

```
# let plus_two n = n + 2;;
val plus_two : int -> int = <fun>
# plus_two 17;;
- : int = 19

# let plus_two = fun n -> n + 2;;
val plus_two : int -> int = <fun>
# plus_two 14;;
- : int = 16
```

First definition syntactic sugar for second

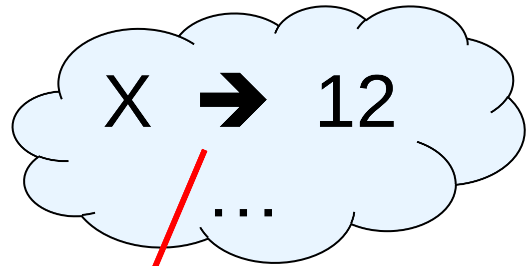
Using a nameless function

```
# (fun x -> x * 3) 5;;    (* An application *)
- : int = 15

# ((fun y -> y +. 2.0), (fun z -> z * 3));;
  (* As data *)
- : (float -> float) * (int -> int) = (<fun>, <fun>)
```

Note: in `fun v -> expression(v)`, the scope of variable is only the body `expression(v)`

Values fixed at declaration time

```
# let x = 12;;
val x : int = 12
```




```
# let plus_x y = y + x;;
val plus_x : int -> int = <fun>

# plus_x 3;;
```

What is the result?

Values fixed at declaration time

```
# let x = 12;;
val x : int = 12

# let plus_x y = y + x;;
val plus_x : int -> int = <fun>

# plus_x 3;;
- : int = 15
```

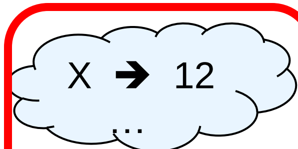
Values fixed at declaration time

```
...
# let x = 7;;
      (* New declaration, not an update *)
val x : int = 7


# plus_x 3;;
```

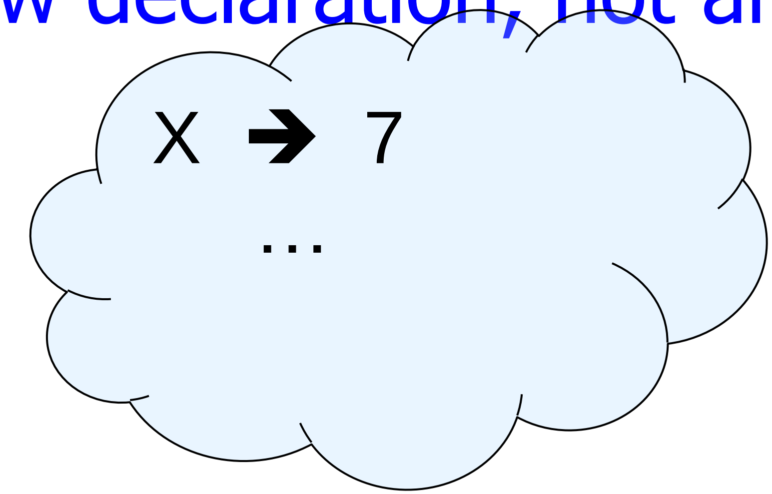
What is the result this time?

Values fixed at declaration time

```
# let x = 7;;   (* New declaration, not an
  update *)
val x : int = 7
```




```
# plus_x 3;;
```



What is the result this time?

Values fixed at declaration time

```
# let x = 7;;   (* New declaration, not an update *)
val x : int = 7

# plus_x 3;;
- : int = 15
```

Question

- Observation: Functions are first-class values in this language
- Question: What value does the environment record for a function variable?
- **Answer: a closure (let's see!)**

Save the Environment!

- A *closure* is a pair of an environment and an association of a formal parameter (the input variables)* with an expression (the function body), written:

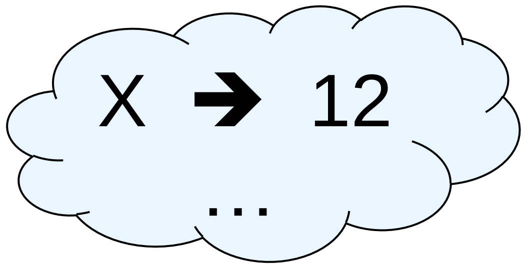
$$f \rightarrow < (v1,\ldots,vn) \rightarrow exp, \rho_f >$$

- Where ρ_f is the environment in effect when f is defined (if f is a simple function)

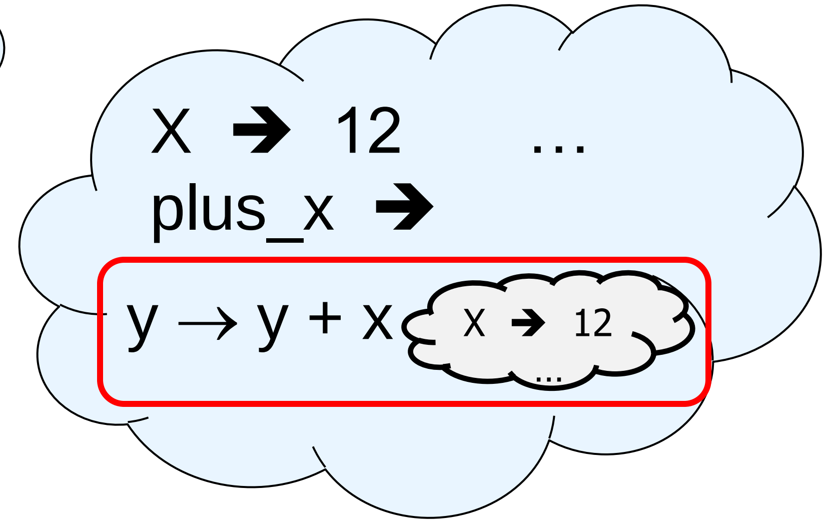
* Will come back to the "formal parameter"

Recall: let plus_x = fun x => y + x

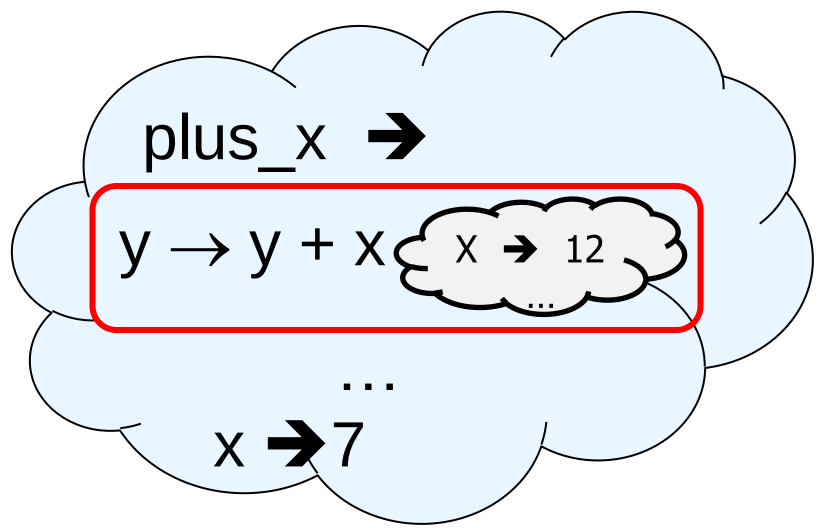
let x = 12



let plus_x = fun y -> y + x



let x = 7



Closure for plus_x

- When plus_x was defined, had environment:

$$\rho_{plus\_x} = \{\ldots, x \to 12, \ldots\}$$

- Recall: let plus_x y = y + x

  is really let plus_x = fun y -> y + x

- Closure for fun y -> y + x:

$$< y \to y + x, \rho_{plus\_x} >$$

- Environment just after plus_x defined:

$$\{plus\_x \to < y \to y + x, \rho_{plus\_x} >\} + \rho_{plus\_x}$$

Similar to set union!
(but subtle differences; new decl. replaces old)

Now it's your turn

You should be able complete ACT1

Functions with more than one argument

```
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>

# let t = add_three 6 3 2;;
val t : int = 11

# let add_three =
   fun x -> (fun y -> (fun z -> x + y + z));;
val add_three : int -> int -> int -> int = <fun>
```

Again, first syntactic sugar for second

Functions with more than one argument

```
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
```

- What is the value of add_three?
- Let ρ_{add_three} be the environment before the declaration
- Remember:

```
let add_three =
 fun x -> (fun y -> (fun z -> x + y + z));;
```

Value:

<x ->fun y -> (fun z -> x + y + z), ρ_{add_three} >

Partial application of functions

```
let add_three x y z = x + y + z;;
```

```
# let h = add_three 5 4;;
val h : int -> int = <fun>

# h 3;;
- : int = 12

# h 7;;
- : int = 16
```

Partial application also called ***sectioning***

Functions as arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>

# let g = thrice plus_two;;
val g : int -> int = <fun>

# g 4;;
- : int = 10

# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

150 minutes

Tuples as Values

```
//  ρ0 = {c → 4, a → 1, b → 5}
# let s = (5,"hi",3.2);;
val s : int * string * float = (5, "hi", 3.2)

//  ρ = {s → (5, "hi", 3.2), c → 4, a → 1, b → 5}
```

Pattern Matching with Tuples

```
//  ρ = {s → (5, "hi", 3.2), a → 1, b → 5, c → 4}

# let (a,b,c) = s;;        (* (a,b,c) is a pattern *)
val a : int = 5
val b : string = "hi"
val c : float = 3.2

# let (a, _, _) = s;;
val a : int = 5

# let x = 2, 9.3;;      (* tuples don't require parens in Ocaml *)
val x : int * float = (2, 9.3)
```

Nested Tuples

```
#  (*Tuples can be nested *)
# let d = ((1,4,62),("bye",15),73.95);;
val d : (int * int * int) * (string * int) * float =
  ((1, 4, 62), ("bye", 15), 73.95)

#  (*Patterns can be nested *)
# let (p, (st,_), _) = d;;
                 (* _ matches all, binds nothing *)
val p : int * int * int = (1, 4, 62)
val st : string = "bye"
```

150 minutes

Match Expressions

```
# let triple_to_pair triple =
  match triple
  with (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
```

- Each clause: pattern on left, expression on right
- Each x, y has scope of only its clause
- Use first matching clause

```
val triple_to_pair :
      int * int * int -> int * int = <fun>
```

Functions on tuples

```
# let plus_pair (n,m) = n + m;;
val plus_pair : int * int -> int = <fun>

# plus_pair (3,4);;
- : int = 7

# let twice x = (x,x);;
val twice : 'a -> 'a * 'a = <fun>

# twice 3;;
- : int * int = (3, 3)

# twice "hi";;
- : string * string = ("hi", "hi")
```

Curried vs Uncurried

- Recall

```
# let add_three u v w = u + v + w;;
val add_three : int -> int -> int -> int = <fun>
```

- How does it differ from

```
# let add_triple (u,v,w) = u + v + w;;
val add_triple : int * int * int -> int = <fun>
```

- add_three is **curried**;
- add_triple is **uncurried**

Curried vs Uncurried

```
# add_three 6 3 2;;
- : int = 11

# add_triple (6,3,2);;
- : int = 11

# add_triple 5 4;;
Characters 0-10:  add_triple 5 4;;
                       ^^^^^^^^^^
This function is applied to too many arguments,
maybe you forgot a `;'

# fun x -> add_triple (5,4,x);;
: int -> int = <fun>
```

Extras

Evaluating declarations

- Evaluation uses an environment ρ
- To evaluate a (simple) declaration let $x = e$
  - Evaluate expression e in ρ to value v
  - Update ρ with x v: $\{x \to v\} + \rho$
- Update: $\rho_1 + \rho_2$ has all the bindings in ρ_1 and all those in ρ_2 that are not rebound in ρ_1

$\{x \to 2, y \to 3, a \to \text{“hi”}\} + \{y \to 100, b \to 6\}$

$= \{x \to 2, y \to 3, a \to \text{“hi”}, b \to 6\}$

Evaluating expressions in OCaml

- Evaluation uses an environment ρ
- A constant evaluates to itself, including primitive operators like + and =
- To evaluate a variable, look it up in ρ: $\rho(v)$
- To evaluate a tuple $(e_1,\ldots,e_n)$,
  - Evaluate each e_i to v_i, right to left for Ocaml
  - Then make value $(v_1,\ldots,v_n)$

Evaluating expressions in OCaml

- To evaluate uses of +, _ , etc, eval args, then do operation
- Function expression evaluates to its closure
- To evaluate a local dec: let x = e1 in e2
  - Eval e1 to v, then eval e2 using $\{x \to v\} + \rho$
- To evaluate a conditional expression: if b then e1 else e2
  - Evaluate b to a value v
  - If v is True, evaluate e1
  - If v is False, evaluate e2

Evaluation of Application with Closures

- Given application expression $f\ e$
- In Ocaml, evaluate e to value v
- In environment ρ, evaluate left term to closure, $c = <(x_1,\ldots,x_n) \rightarrow b, \rho'>$
  - $(x_1,\ldots,x_n)$ variables in (first) argument
  - v must have form $(v_1,\ldots,v_n)$
- Update the environment ρ' to

  $\rho'' = \{x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n\} + \rho'$
- Evaluate body b in environment ρ''