

CS440: Programming Languages and Translators

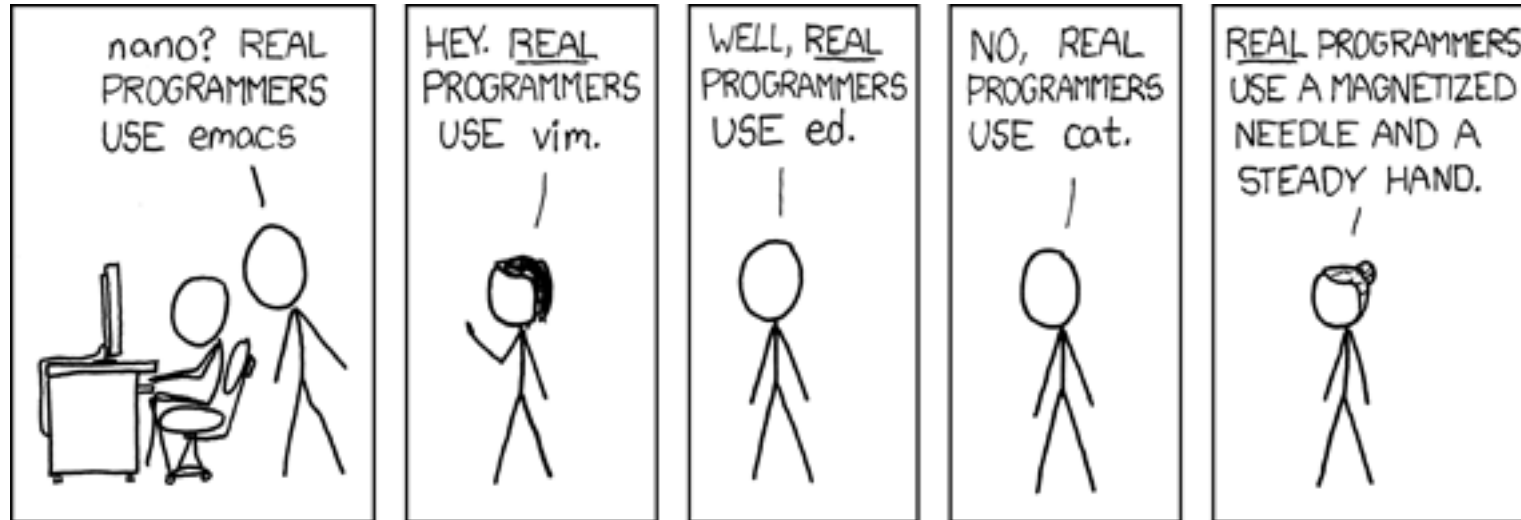
Lecture 0

Spring 2023

Outline

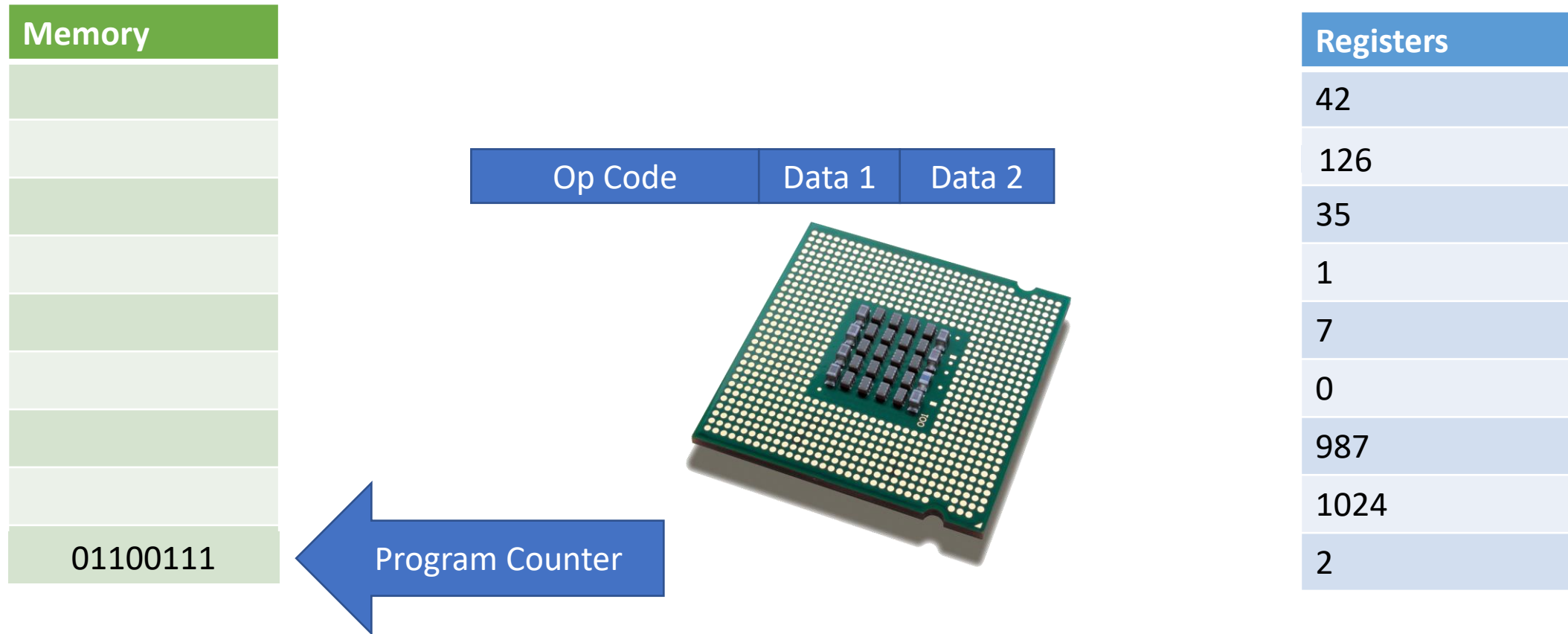
1. Programming Languages
2. Translators: Compilers and Interpreters
3. Types of Programming Languages
4. Syllabus

You can program without programming languages... if you really want



xkcd

Computer Architecture in One Slide

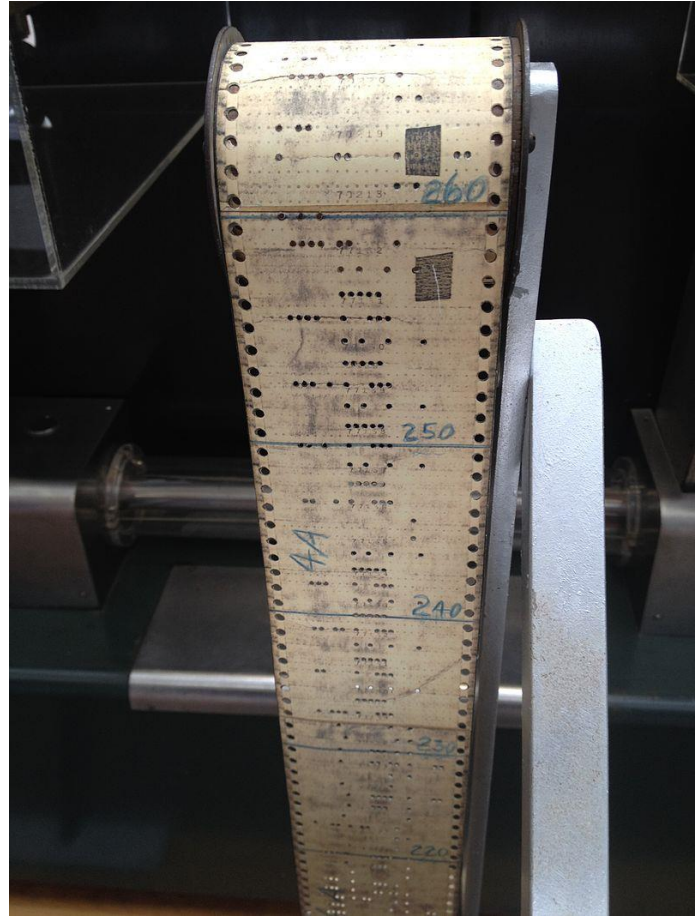


You can program without programming languages... if you really want

Altair 8800
1974



You can program without programming languages... if you really want

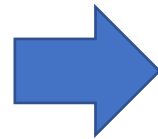


Instruction tape for Harvard Mark I
~1944

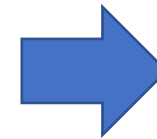
Assembly code makes instructions more human-readable

```
push    %rbp
mov     %rsp,%rbp
sub     $0x30,%rsp
mov     %rdi,-0x28(%rbp)
mov     %fs:0x28,%rax

mov     %rax,-0x8(%rbp)
xor     %eax,%eax
mov     -0x28(%rbp),%rax
mov     (%rax),%rax
mov     %rax,-0x10(%rbp)
cmpq   $0x0,-0x10(%rbp)
je      7c2 <MergeSort+0x88>
mov     -0x10(%rbp),%rax
mov     0x8(%rax),%rax
test   %rax,%rax
je      7c2 <MergeSort+0x88>
lea    -0x18(%rbp),%rdx
lea    -0x20(%rbp),%rcx
mov    -0x10(%rbp),%rax
mov    %rcx,%rsi
mov    %rax,%rdi
callq  877 <FrontBackSplit>
lea    -0x20(%rbp),%rax
mov    %rax,%rdi
callq  73a <MergeSort>
lea    -0x18(%rbp),%rax
mov    %rax,%rdi
callq  73a <MergeSort>
mov    -0x18(%rbp),%rdx
mov    -0x20(%rbp),%rax
mov    %rdx,%rsi
mov    %rax,%rdi
callq  7d9 <SortedMerge>
mov    %rax,%rdx
mov    -0x28(%rbp),%rax
```



Assembler



Binary

```
1010101010010001000100
1111001010100100010000
0111110110000110000...
```

If we can turn text into binaries, why not easier-to-write text?



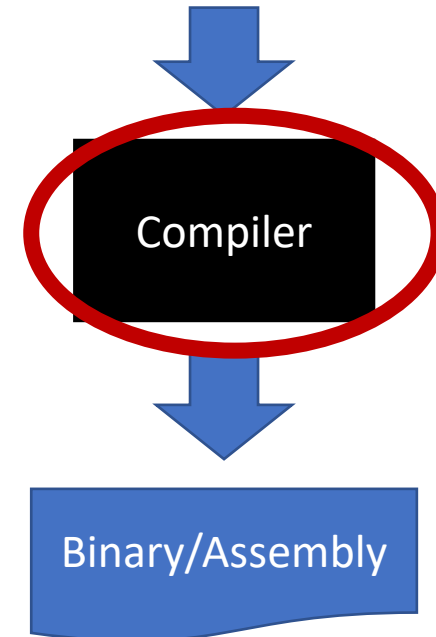
Rear Admiral Grace Hopper
(1906-1992)

COBOL
(1959)

```
ADD 1 TO x
ADD 1, a, b TO x ROUNDED, y, z ROUNDED

ADD a, b TO c
  ON SIZE ERROR
    DISPLAY "Error"
END-ADD

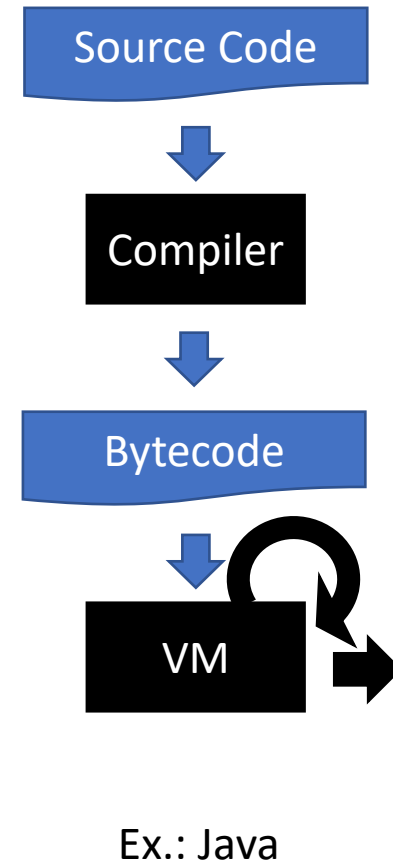
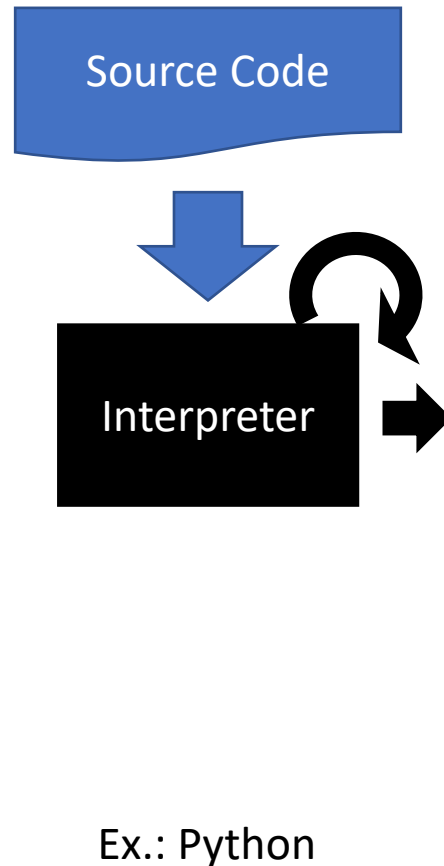
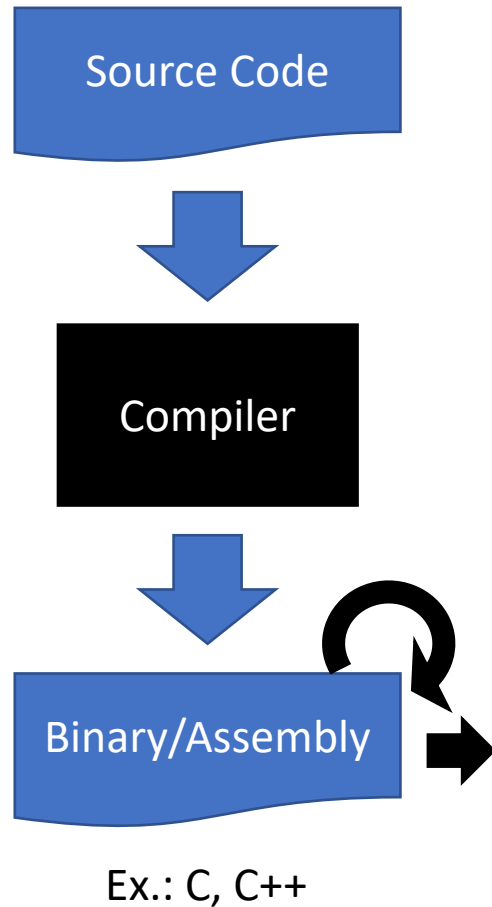
ADD a TO b
  NOT SIZE ERROR
    DISPLAY "No error"
  ON SIZE ERROR
    DISPLAY "Error"
```



Outline

1. Programming Languages
2. Translators: Compilers and Interpreters
3. Types of Programming Languages
4. Syllabus

There are different ways of translating a programming language



Outline

1. Programming Languages
2. Translators: Compilers and Interpreters
3. Types of Programming Languages
4. Syllabus

All programming languages are the same...
in a deep sense

“Turing completeness”

But the choice of language still matters in a very real sense—languages
are tools!

Programming Language =

Syntax

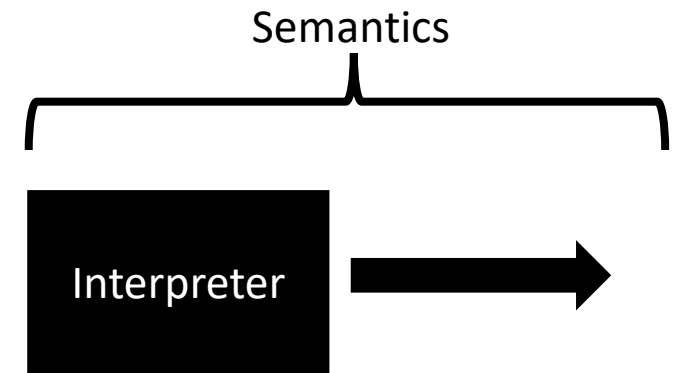
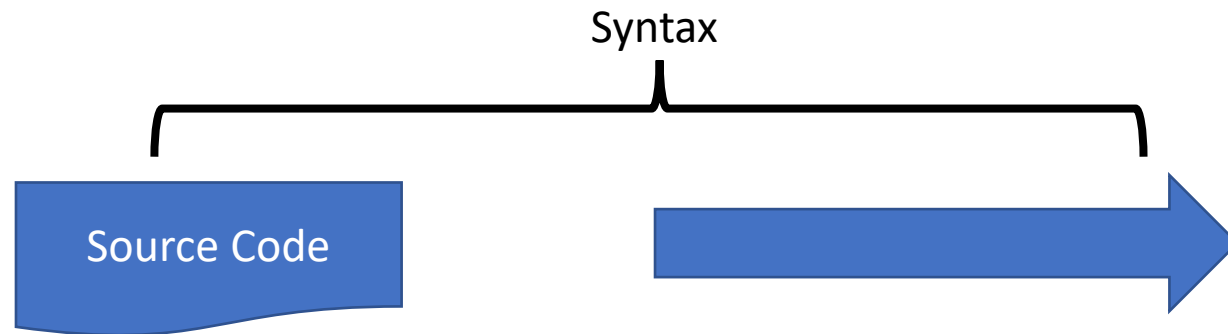
What programs *look like*

+

Semantics

What programs *mean*

Syntax vs. semantics: Python

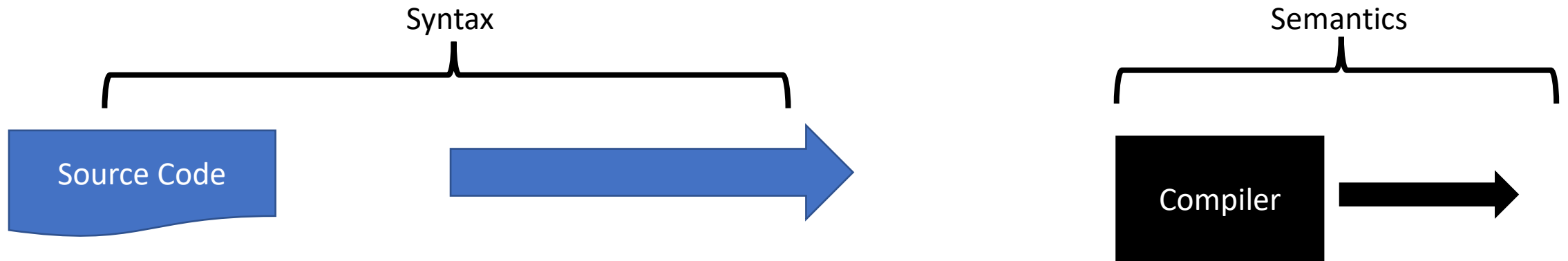


```
def func ():  
    return 5 + "hello"
```

```
File "main.py", line 1  
    def func ();  
            ^  
SyntaxError: invalid syntax
```

```
File "main.py", line 2, in func  
    return 5 + "hello"  
TypeError: unsupported operand  
type(s) for +: 'int' and 'str'
```

Syntax vs. semantics: OCaml



```
let func () = 5 + "hello"
```

Line 5, characters 18-25:
Error: This expression has
type string but an
expression was expected of
type int

Semantics =

Static

Analyzed at compile time

+

Where do we check types?

Dynamic

Happens at run time

We can divide programming languages by whether they have *static* or *dynamic* types

- Static languages: types checked at compile time: *no type errors* at runtime
- Dynamic languages: types checked at run time, can have type errors
- (Weakly typed languages): types checked at compile time, but can be avoided, resulting in unexpected behavior or type errors at run time

We can also divide programming languages based on *paradigm* (how you think about programming)

- Imperative: *tell computer what to do*
- Functional: *describe the computation mathematically*
- Object-oriented: *objects perform computation and carry data*
- Scripting
- Relational
- Domain-specific
- Logic

	Static	Dynamic
Imperative	Typescript, Pascal	Python, Javascript
Functional	Haskell, OCaml	Scheme, Racket
Object-oriented	C++, C#, Java	

Knowing the right paradigm to use can make programming easier

Task: Sort a linked list (using merge sort)

```
1 // Merge sort for linked list
2
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 // Node structure
8 struct Node {
9     int data;
10    struct Node *next;
11};
12
13 // Function to create a new node
14 struct Node* createNode(int data) {
15     struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
16     newNode->data = data;
17     newNode->next = NULL;
18     return newNode;
19}
20
21 // Function to insert a new node at the end of the list
22 void insertAtEnd(struct Node** head, int data) {
23     struct Node* newNode = createNode(data);
24     if (*head == NULL) {
25         *head = newNode;
26     } else {
27         struct Node* temp = *head;
28         while (temp->next != NULL) {
29             temp = temp->next;
30         }
31         temp->next = newNode;
32     }
33}
34
35 // Function to print the linked list
36 void printList(struct Node* head) {
37     struct Node* temp = head;
38     while (temp != NULL) {
39         printf("%d ", temp->data);
40         temp = temp->next;
41     }
42     printf("\n");
43}
44
45 // Function to sort the linked list using merge sort
46 struct Node* mergeSort(struct Node** head) {
47     if (*head == NULL || (*head)->next == NULL) {
48         return *head;
49     }
50     // Find the middle of the list
51     struct Node* middle = *head;
52     struct Node* prev = NULL;
53     while (middle->next != NULL) {
54         prev = middle;
55         middle = middle->next;
56     }
57     // Split the list into two halves
58     prev->next = NULL;
59     struct Node* left = *head;
60     struct Node* right = middle;
61     // Recursively sort both halves
62     left = mergeSort(&left);
63     right = mergeSort(&right);
64     // Merge the two sorted halves
65     struct Node* sorted = merge(left, right);
66     return sorted;
67}
68
69 // Function to merge two sorted linked lists
70 struct Node* merge(struct Node* left, struct Node* right) {
71     struct Node* sorted = NULL;
72     while (left != NULL || right != NULL) {
73         struct Node* temp = NULL;
74         if (left == NULL) {
75             temp = right;
76             right = right->next;
77         } else if (right == NULL) {
78             temp = left;
79             left = left->next;
80         } else if (left->data < right->data) {
81             temp = left;
82             left = left->next;
83         } else {
84             temp = right;
85             right = right->next;
86         }
87         sorted = merge(sorted, temp);
88     }
89     return sorted;
90}
91
92 // Driver program to test above functions
93 int main() {
94     struct Node* head = NULL;
95     insertAtEnd(&head, 1);
96     insertAtEnd(&head, 3);
97     insertAtEnd(&head, 2);
98     insertAtEnd(&head, 4);
99     insertAtEnd(&head, 5);
100    printList(head);
101    head = mergeSort(&head);
102    printList(head);
103    return 0;
104}
```

C

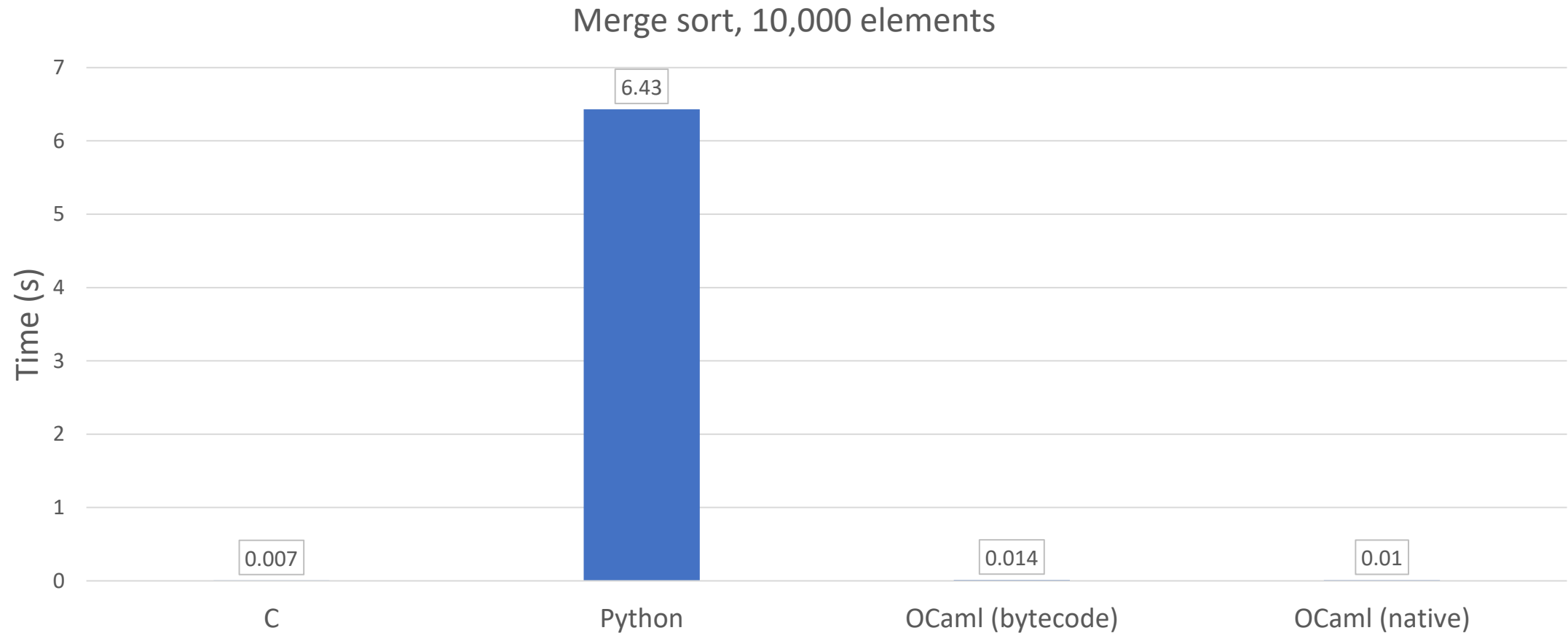
```
1 # Python program to merge sort of linked list
2
3 # Node class
4 class Node:
5     def __init__(self, data):
6         self.data = data
7         self.next = None
8
9 # Function to insert a new node at the end of the list
10 def insertAtEnd(head, data):
11     new_node = Node(data)
12     if head is None:
13         head = new_node
14     else:
15         temp = head
16         while temp.next is not None:
17             temp = temp.next
18         temp.next = new_node
19
20 # Function to print the linked list
21 def printList(head):
22     temp = head
23     while temp:
24         print(temp.data, end=" ")
25         temp = temp.next
26     print()
27
28 # Function to sort the linked list using merge sort
29 def mergeSort(head):
30     if head is None or head.next is None:
31         return head
32     # Find the middle of the list
33     middle = head
34     prev = None
35     while middle.next is not None:
36         prev = middle
37         middle = middle.next
38     # Split the list into two halves
39     prev.next = None
40     left = head
41     right = middle
42     # Recursively sort both halves
43     left = mergeSort(left)
44     right = mergeSort(right)
45     # Merge the two sorted halves
46     sorted = merge(left, right)
47     return sorted
48
49 # Function to merge two sorted linked lists
50 def merge(left, right):
51     sorted = None
52     while left is not None or right is not None:
53         temp = None
54         if left is None:
55             temp = right
56             right = right.next
57         elif right is None:
58             temp = left
59             left = left.next
60         elif left.data < right.data:
61             temp = left
62             left = left.next
63         else:
64             temp = right
65             right = right.next
66         sorted = merge(sorted, temp)
67     return sorted
68
69 # Driver program to test above functions
70 if __name__ == '__main__':
71     head = None
72     insertAtEnd(head, 1)
73     insertAtEnd(head, 3)
74     insertAtEnd(head, 2)
75     insertAtEnd(head, 4)
76     insertAtEnd(head, 5)
77     printList(head)
78     head = mergeSort(head)
79     printList(head)
```

Python

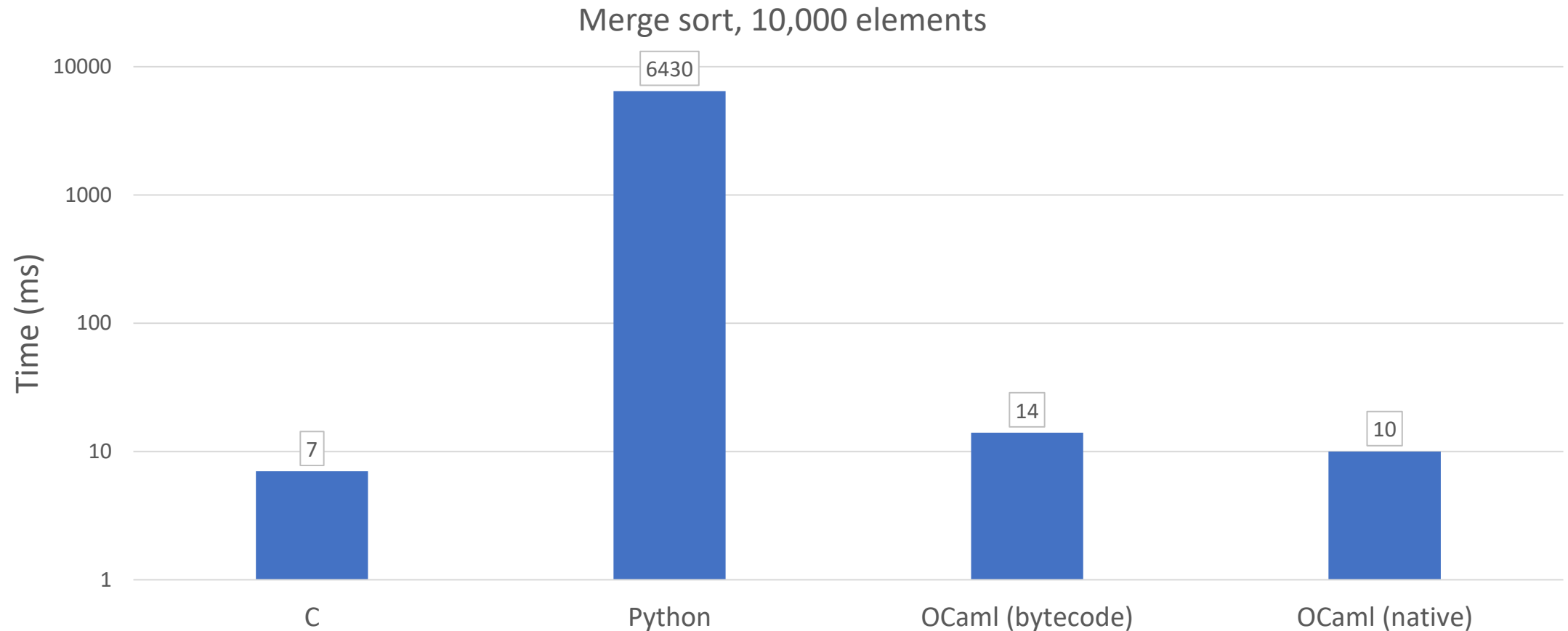
```
1 # OCaml program to merge sort of linked list
2
3 # Node structure
4 type Node = {
5   data : int;
6   next : Node option;
7 };
8
9 # Function to insert a new node at the end of the list
10 let rec insertAtEnd head data =
11   let new_node = { data; next = None } in
12   if head = None then
13     new_node
14   else
15     let temp = head in
16     let () = while temp.next = None do temp <- temp.next end in
17     temp.next <- Some new_node
18
19 # Function to print the linked list
20 let rec printList head =
21   let () = while head <= None do head <- head.next end in
22   print_int head.data;
23   let () = while head <= None do head <- head.next end in
24   print_string "\n"
25
26 # Function to sort the linked list using merge sort
27 let rec mergeSort head =
28   if head = None || head.next = None then
29     head
30   else
31     let middle = head in
32     let prev = None in
33     while middle.next <= None do
34       prev <- middle;
35       middle <- middle.next
36     end
37     prev.next <- None
38     let left = head in
39     let right = middle in
40     let () = (left, right) <- (mergeSort left, mergeSort right) in
41     merge left right
42
43 # Function to merge two sorted linked lists
44 let rec merge left right =
45   let sorted = None in
46   while left <= None || right <= None do
47     let temp = None in
48     if left = None then
49       temp <- right;
50       right <- right.next
51     else if right = None then
52       temp <- left;
53       left <- left.next
54     else if left.data < right.data then
55       temp <- left;
56       left <- left.next
57     else
58       temp <- right;
59       right <- right.next
60     sorted <- merge sorted temp
61   end
62   sorted
63
64 # Driver program to test above functions
65 let () =
66   let head = None in
67   insertAtEnd head 1;
68   insertAtEnd head 3;
69   insertAtEnd head 2;
70   insertAtEnd head 4;
71   insertAtEnd head 5;
72   printList head;
73   head <- mergeSort head;
74   printList head
```

OCaml

Knowing about the language and how it's translated can help you write faster code



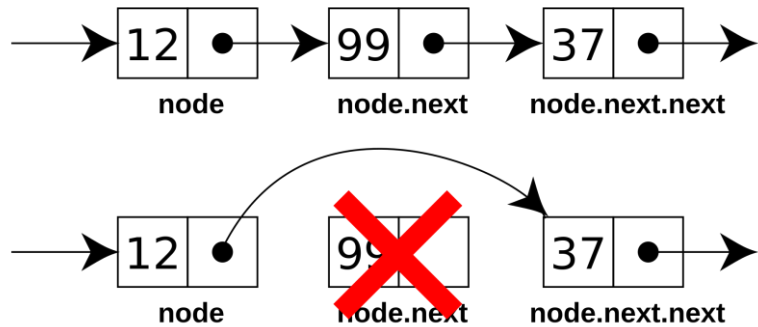
Knowing about the language and how it's translated can help you write faster code



Type systems can express different levels of guarantees

- C `node *mergesort(node *list)`
 - Takes a pointer to a node and returns a pointer to a node.
- OCaml `mergesort : int list -> int list`
 - Takes an integer list and returns an integer list.
- Haskell `mergesort :: IO ([int] -> [int])`
 - Takes an integer list, returns an integer list and performs I/O (e.g., printing).
- Coq `mergesort : forall (l1 : list int), exists (l2: int list),
Sorted l2 /\ Permutation l1 l2`
 - Takes an integer list and returns a sorted permutation of it.

Different languages are up to different tasks



 OCaml ?

C?

 Rust?

Outline

1. Programming Languages
2. Translators: Compilers and Interpreters
3. Types of Programming Languages
4. **Syllabus**

Course Goals

- Learn to evaluate and discuss programming languages
 - Learn the lingo (impressing people with jargon isn't the point, but is a side effect)

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

- Reason precisely about what programs mean
 - Inject mathematical rigor into programming
- Become a *creator* of PLs, not just a *consumer*

Sections/Attendance

- Section 01: In-person
 - Attendance not recorded, but attendance/participation may be used to “break ties”
- Section 02: Online
 - Lecture videos will be posted to Blackboard after each lecture

Course Staff

- Instructor: Stefan Muller
 - Office Hours: Mon., 11am-12pm (Online – link to come)
Thur., 2-3pm (SB 218E)
- TA: Xincheng Yang
 - Office Hours: Tue., 2-4pm (Online – appt. link to come)
Wed., 2-4pm (SB 004)

Course Website: <http://cs.iit.edu/~cs440/> Important info, notes, etc.

CS440: Programming Languages and Translators, Spring 2023

Instructor: Stefan Muller, smuller2@iit.edu
Office Hours: TBA
TA: TBA
Lectures:

Section 01: Tue, Thur 10:00-11:15 AM, SB 104
Section 02: Online only

[Schedule](#) [Resources](#) [Policies](#)

Schedule

Note: this schedule is tentative and subject to change.

For the readings posted:

"PDB" = "Purple Dragon Book" (Aho et al.)
"FPO" = "Functional Programming in OCaml" (linked below)
"TAPL" = *Types and Programming Languages* (Pierce)
"PFPL" = *Practical Foundations for Programming Languages* (Harper)

		Topic	Readings	Notes
January	10	Intro	Languages and course overview	
	12		Compiler structure, interpreters, OCaml	
	17	OCaml	OCaml evaluation, types, expressions	
	19		Functions and recursion	
	24		More on types	
	26		Lists and tail recursion	
	31		Records and algebraic data types	
February	2		Higher-order functions	
	7		Higher-order functions	
	9		Side effects	
	14	Interpreters	Building an interpreter	
	16		Closures	



Other ways to get help

- Discord: IIT CS server, #cs440 channel
 - If you're not on it, we'll send an invitation
- Academic Resource Center (ARC): www.iit.edu/arc
 - FREE subject matter tutoring and academic coaching

	Discord	Office Hours	Email	ARC
General questions about lectures, logistics, etc.	✓	✓		
General discussion, clarifications, about HW questions	✓	✓		
Specific questions about your HW answers		✓		✓
More in-depth personal tutoring				✓
Personal matters (accommodations, other requests, etc.)			✓	

Collaboration and Academic Honesty

- Discussing general concepts is encouraged
- Discussing broad strategies for doing lab tasks is OK – don't discuss actual answers or code
 - If, after your discussion, you don't take any notes/pictures and write up your code/solutions by yourself, you're probably OK
 - Cite collaborators and any other resources in your write-up
- Not allowed:
 - Working together
 - Sharing answers
 - Looking for answers on the internet

This is the short version: read the details on the course website

(Tentative) Schedule

- Intro (1 week – you are here)
- Learn OCaml (~4 weeks)
- Interpreters (~2 weeks)
- *Midterm*
- Type checking (~2 weeks)
- *Spring break*
- Formal semantics (~2 weeks)
- Formal type systems (~2 weeks)
- Other topics and wrap-up (~3 weeks)

Labs/Projects/Homeworks/Problem sets

- 6-7 homeworks, ~2 weeks each
 - Lab 0 Out ~Thursday, Due 1/26
- Written and programming
- Work individually

Late Days:

- 7 per student, extend deadline 24 hours
- No more than 2 per assignment
- If no more late days, 10% late penalty per day
- No work accepted >48 hours late

Exams

- Midterm (tentatively Mar. 2)
- Final (finals week)

- Details TBA

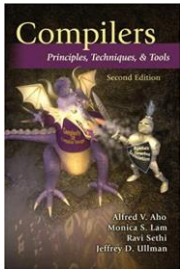
- (No using late days, sorry)

Grading

- 50% Homeworks
- 20% Midterm
- 30% Final

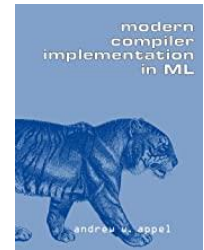
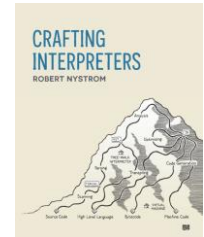
≥ 90	A
80-90	B
70-80	C
60-70	D
< 60	E

Textbooks

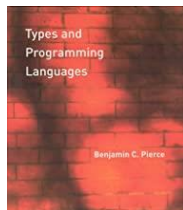


On Compilers/interpreters:

- “Purple Dragon Book”: Aho et al. *Compilers: Principles, Techniques and Tools (2nd ed.)*
- Appel. *Modern Compiler Implementation in ML*
- Nystrom. *Crafting Interpreters*



For more math-y details:



- Pierce. *Types and Programming Languages*
- Harper. *Practical Foundations for Programming Languages*



For Thursday: Bring laptops if you can!