CSE 4102: Programming Languages

Lectures 0-1

Spring 2026

CSE 4102. Programming Languages. (3 Credits)

Is this…

- an overview of a bunch of different programming languages?
- a survey of the design of programming languages?
- a history course?
- a theory of programming languages course?
- an overview of how PLs are implemented?

Yes.

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)

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

- Learn various PL paradigms
  - Makes it much easier to learn more PLs in the future!
  - Learn how to choose the right tool for the right problem
  - and to use paradigmatic ideas from some languages in others
- Understand the history and key ideas behind the design of programming languages

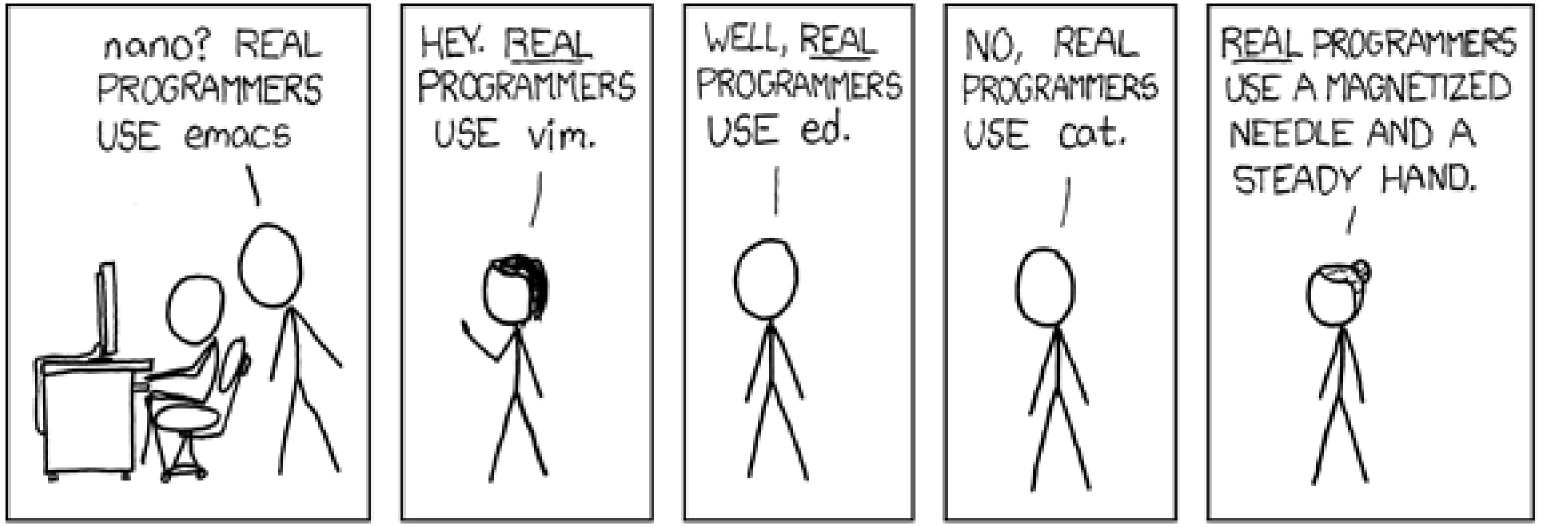
Course Non-goals

- How to write a compiler (that’s a compilers course)
    - You may get a general sense of how to begin designing a PL
- Deeply understand the theory behind PL so you can write mathematical proofs about it (that’s a graduate course)
- Learn every PL under the sun just for the sake of knowing them (that’s a parlor trick, not a course)

Today and Thursday

1. Programming Language overview/history
2. Programming Language paradigms
3. Break: administrivia
4. Programming Language implementation (compilers and interpreters)
5. OCaml introduction

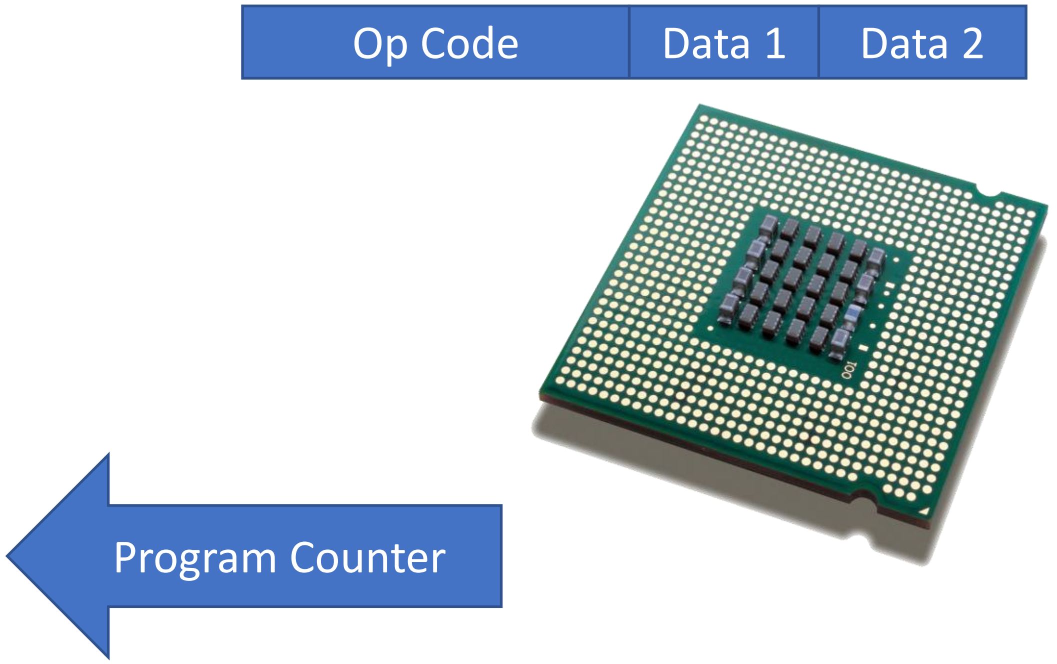
You can program without programming languages… if you really want


nano? REAL PROGRAMMERS USE emacs
HEY. REAL PROGRAMMERS USE vim.
WELL, REAL PROGRAMMERS USE ed.
NO, REAL PROGRAMMERS USE cat.
REAL PROGRAMMERS USE A MAGNETIZED NEEDLE AND A STEADY HAND.

xkcd

Computer Architecture in One Slide

Memory
01100111

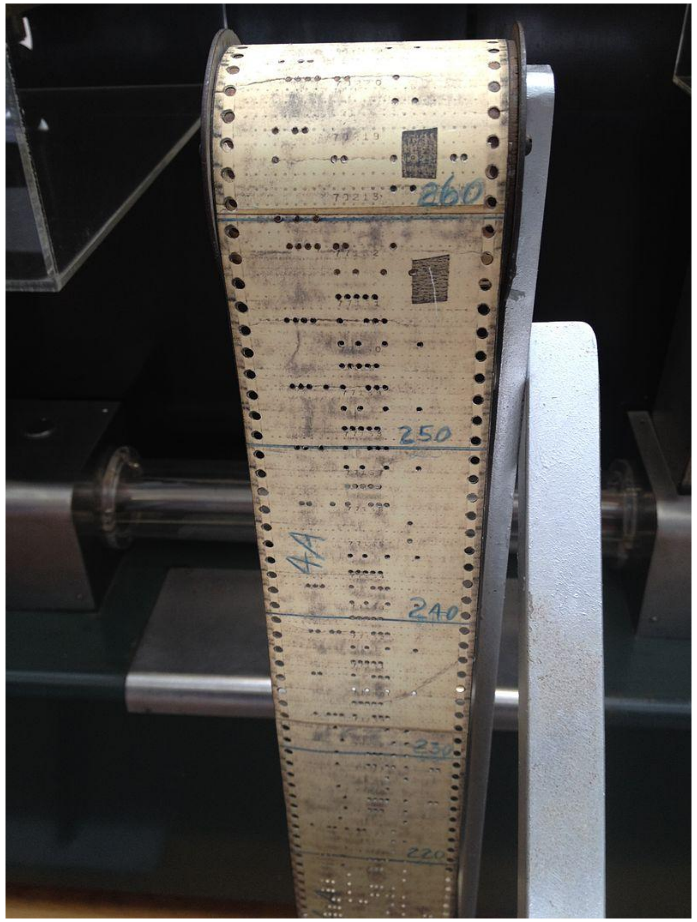


Registers
42
126
35
1
7
0
987
1024
2

You can program without programming languages… if you really want

Altair 8800
1974

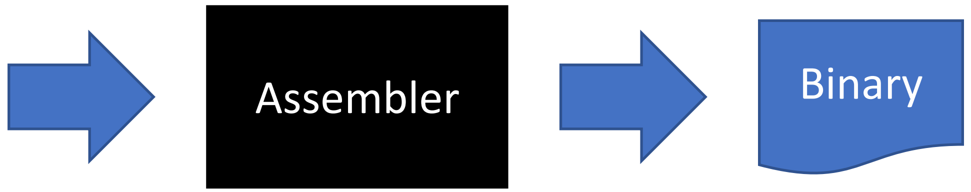
You can program without programming languages… if you really want



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
```

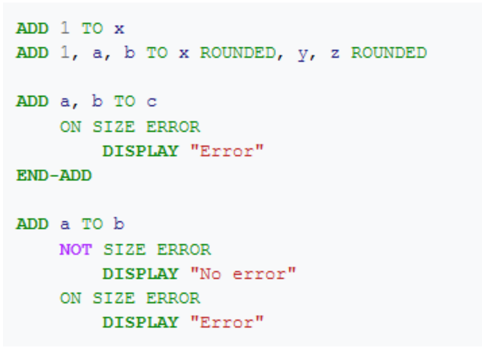


10101010100100010001001
11100101010010001000001
0111110110000110000…

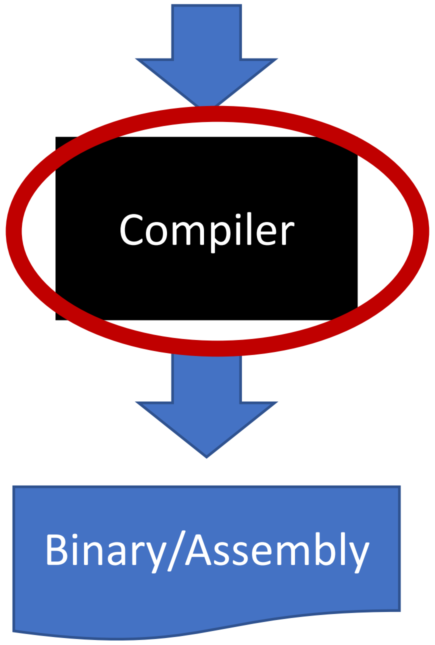
If we can turn text into binaries, why not easier-to-write text?

Rear Admiral Grace Hopper
(1906-1992)

FLOW-MATIC (1955)
⇒ 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"
```



Today and Thursday

1. Programming Language overview/history
2. Programming Language paradigms
3. Break: administrivia
4. Programming Language implementation (compilers and interpreters)
5. OCaml introduction

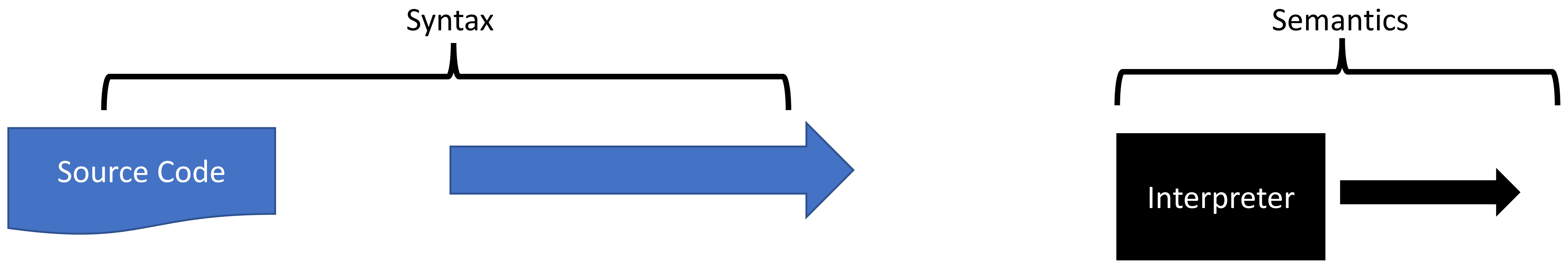
All programming languages are the same… in a deep sense

“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

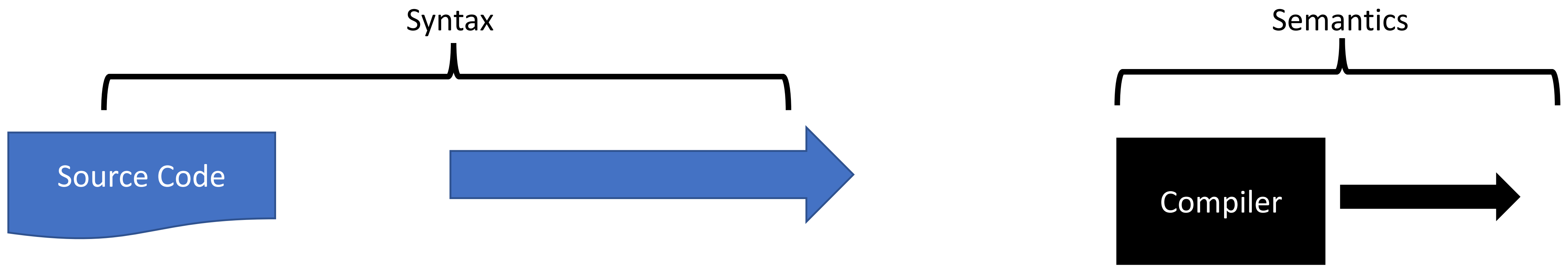


```
def func ():
  return 5 + “hello”
```

```
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

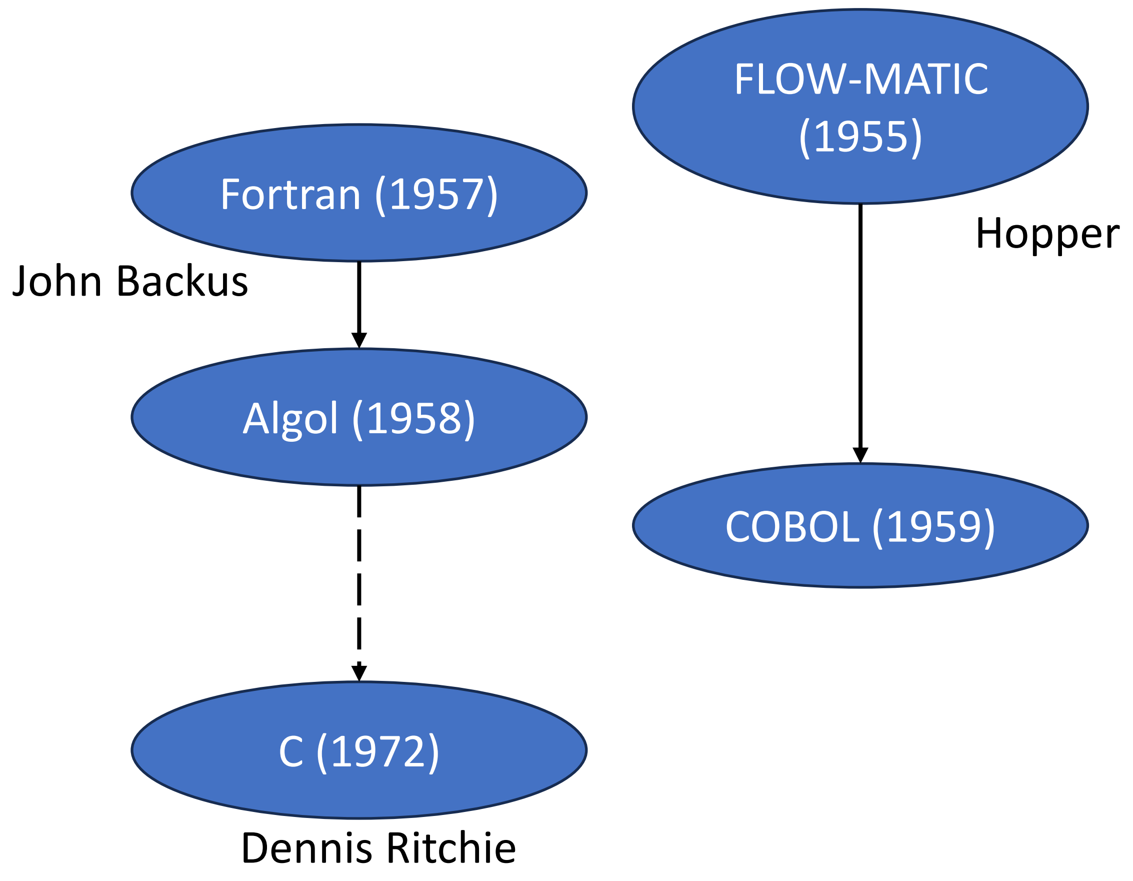
- (Weakly typed languages): types checked at compile time, but can be avoided, resulting in unexpected behavior or type errors at run time

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

- Imperative/Procedural: *tell computer what to do*
- Functional: *describe the computation mathematically*
- Object-oriented: *objects perform computation and carry data*
- Logic: *provide the constraints/requirements for the answer*
- Scripting
- Relational
- Domain-specific

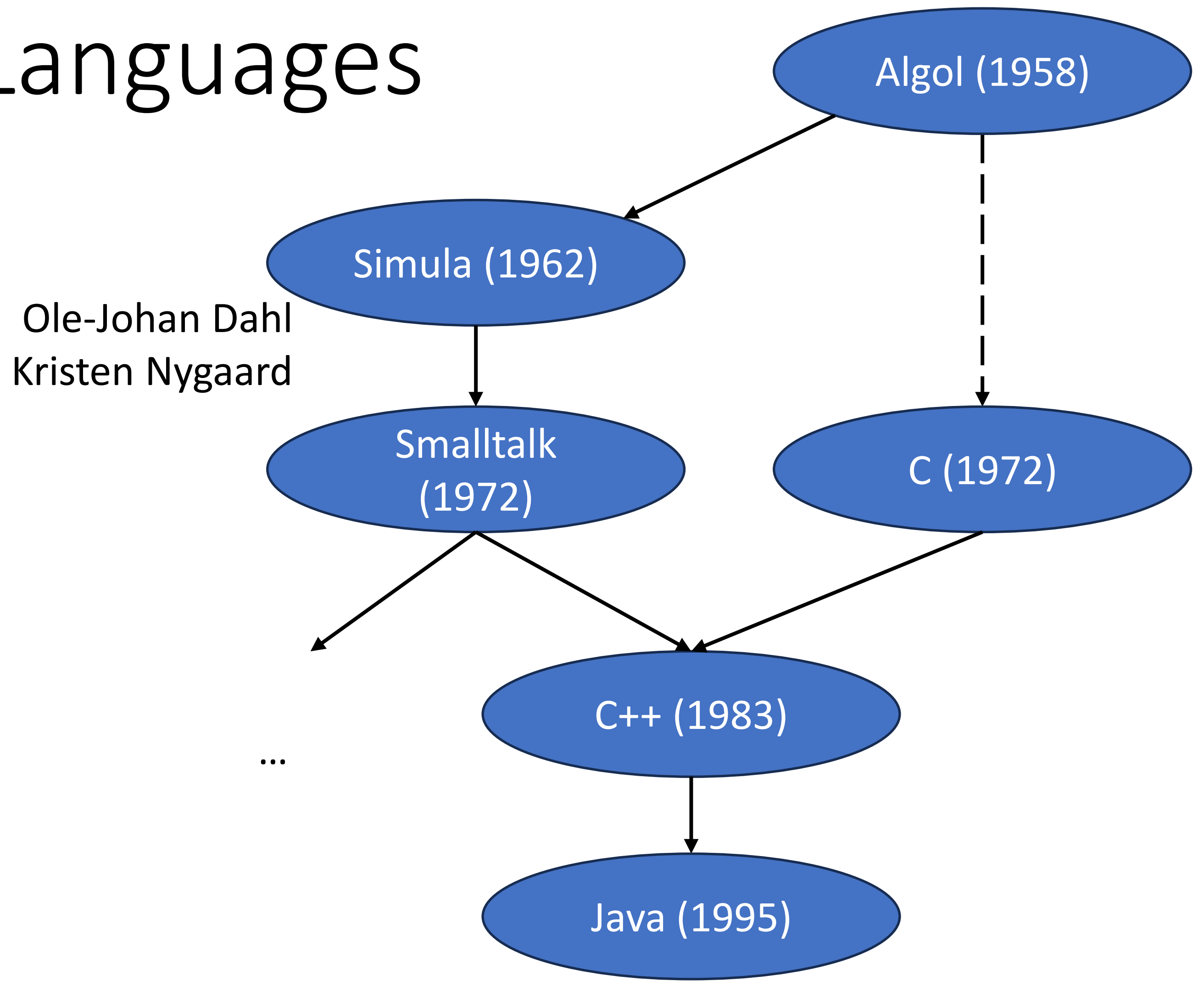
Imperative Programming Languages

- Key features:
  - Performance
  - Computation-centric
  - Works like a computer



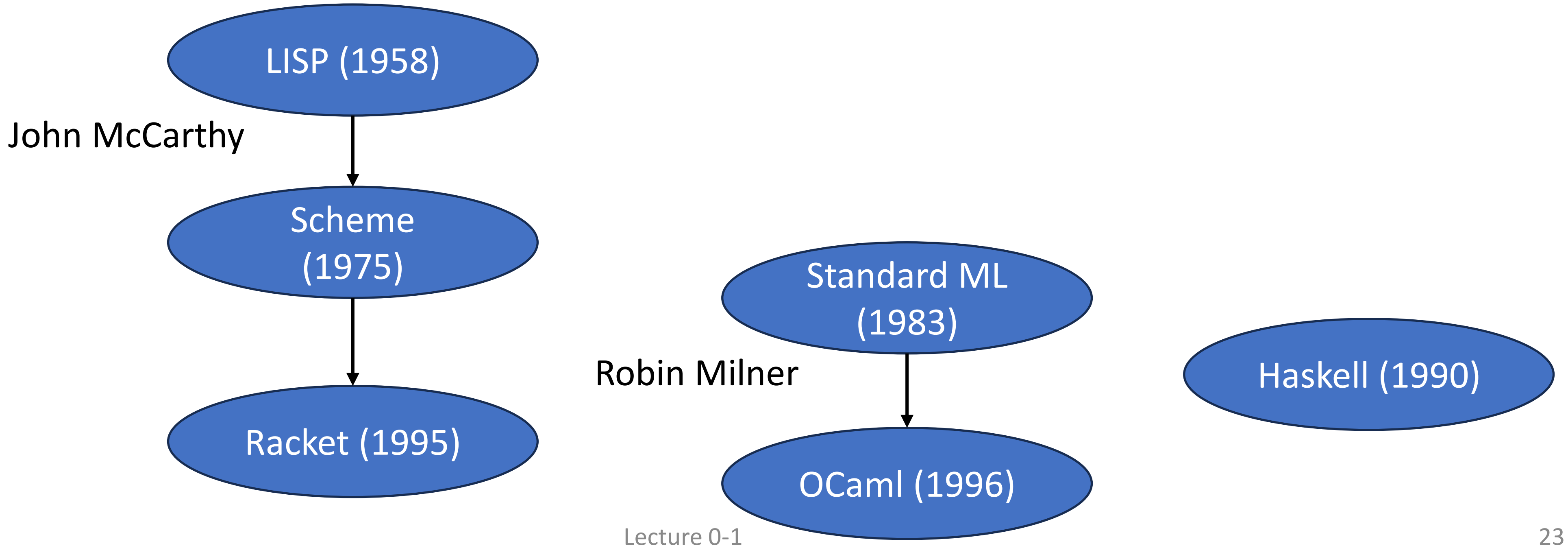
Object-Oriented Languages

- Key Features:
  - State centric
  - Hierarchy of objects
  - Good for simulation



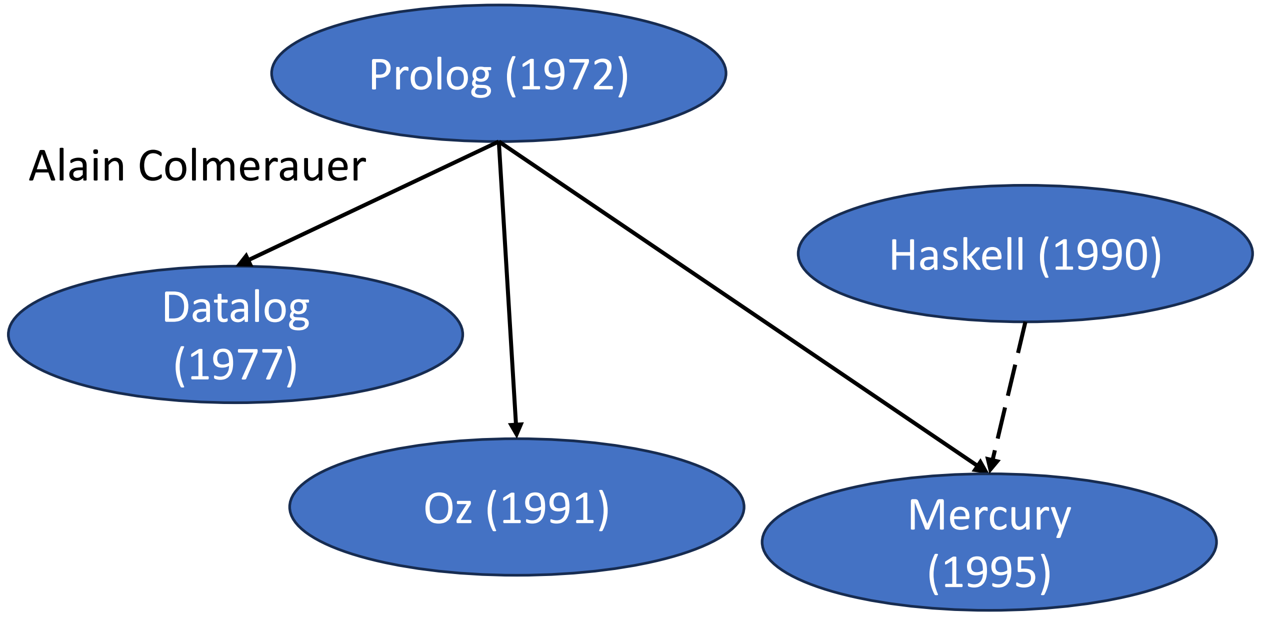
Functional Programming Languages

- Key features
  - Mathematical
  - NOT state-oriented



Logic Programming Languages

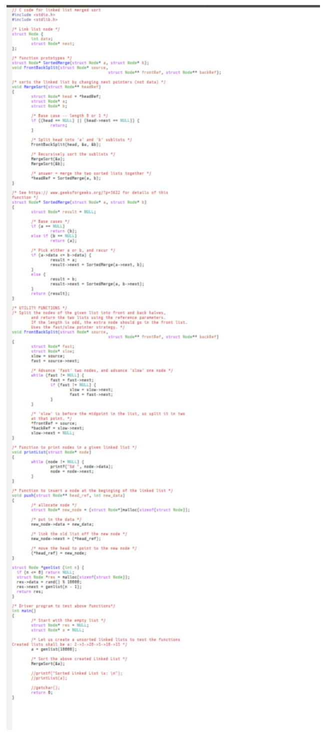
- Key Features:
  - Computation is reasoning
  - Inference
  - Non-determinism



	Static	Dynamic	Untyped
Imperative	C, Rust, .NET?	Python, MATLAB?	Assembly
Functional	Haskell, OCaml	Scheme	
Object-oriented	C++, Java, C#, Typescript, Go?	Javascript, Smalltalk	
Logic		Prolog	
Relational		SQL	

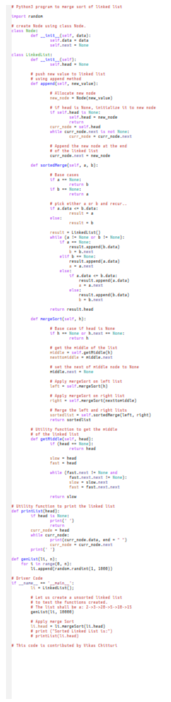
Knowing the right paradigm to use can make programming easier

Task: Sort a linked list (using merge sort)



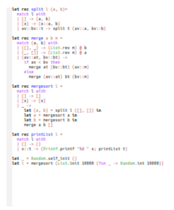
```c
// C code for linked list merged sort
#include <stdio.h>
#include <stdlib.h>

/* Link list node */
struct Node {
        int data;
        struct Node* next;
};

/* function prototypes */
struct Node* SortedMerge(struct Node* a, struct Node* b);
void FrontBackSplit(struct Node* source,
                                        struct Node** frontRef, struct Node** backRef);

/* sorts the linked list by changing next pointers (not data) */
void MergeSort(struct Node** headRef)
{
        struct Node* head = *headRef;
        struct Node* a;
        struct Node* b;

        /* Base case -- length 0 or 1 */
        if ((head == NULL) || (head->next == NULL)) {
                return;
        }

        /* Split head into 'a' and 'b' sublists */
        FrontBackSplit(head, &a, &b);

        /* Recursively sort the sublists */
        MergeSort(&a);
        MergeSort(&b);

        /* answer = merge the two sorted lists together */
        *headRef = SortedMerge(a, b);
}

/* See https:// www.geeksforgeeks.org/?p=3622 for details of this
function */
struct Node* SortedMerge(struct Node* a, struct Node* b)
{
        struct Node* result = NULL;

        /* Base cases */
        if (a == NULL)
                return (b);
        else if (b == NULL)
                return (a);

        /* Pick either a or b, and recur */
        if (a->data <= b->data) {
                result = a;
                result->next = SortedMerge(a->next, b);
        }
        else {
                result = b;
                result->next = SortedMerge(a, b->next);
        }
        return (result);
}

/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves,
        and return the two lists using the reference parameters.
        If the length is odd, the extra node should go in the front list.
        Uses the fast/slow pointer strategy. */
void FrontBackSplit(struct Node* source,
                                        struct Node** frontRef, struct Node** backRef)
{
        struct Node* fast;
        struct Node* slow;
        slow = source;
        fast = source->next;

        /* Advance 'fast' two nodes, and advance 'slow' one node */
        while (fast != NULL) {
                fast = fast->next;
                if (fast != NULL) {
                        slow = slow->next;
                        fast = fast->next;
                }
        }

        /* 'slow' is before the midpoint in the list, so split it in two
        at that point. */
        *frontRef = source;
        *backRef = slow->next;
        slow->next = NULL;
}

/* Function to print nodes in a given linked list */
void printList(struct Node* node)
{
        while (node != NULL) {
                printf("%d ", node->data);
                node = node->next;
        }
}

/* Function to insert a node at the beginging of the linked list */
void push(struct Node** head_ref, int new_data)
{
        /* allocate node */
        struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));

        /* put in the data */
        new_node->data = new_data;

        /* link the old list off the new node */
        new_node->next = (*head_ref);

        /* move the head to point to the new node */
        (*head_ref) = new_node;
}

struct Node *genlist (int n) {
  if (n <= 0) return NULL;
  struct Node *res = malloc(sizeof(struct Node));
  res->data = rand() % 10000;
  res->next = genlist(n - 1);
  return res;
}

/* Driver program to test above functions*/
int main()
{
        /* Start with the empty list */
        struct Node* res = NULL;
        struct Node* a = NULL;

        /* Let us create a unsorted linked lists to test the functions
Created lists shall be a: 2->3->20->5->10->15 */
        a = genlist(10000);

        /* Sort the above created Linked List */
        MergeSort(&a);

        //printf("Sorted Linked List is: \n");
        //printList(a);

        //getchar();
        return 0;
}
```

C



```python
# Python3 program to merge sort of linked list

import random

# create Node using class Node.
class Node:
        def __init__(self, data):
                self.data = data
                self.next = None

class LinkedList:
        def __init__(self):
                self.head = None

        # push new value to linked list
        # using append method
        def append(self, new_value):

                # Allocate new node
                new_node = Node(new_value)

                # if head is None, initialize it to new node
                if self.head is None:
                        self.head = new_node
                        return
                curr_node = self.head
                while curr_node.next is not None:
                        curr_node = curr_node.next

                # Append the new node at the end
                # of the linked list
                curr_node.next = new_node

        def sortedMerge(self, a, b):

                # Base cases
                if a == None:
                        return b
                if b == None:
                        return a

                # pick either a or b and recur..
                if a.data <= b.data:
                        result = a
                else:
                        result = b

                result = LinkedList()
                while (a != None or b != None):
                    if a == None:
                        result.append(b.data)
                        b = b.next
                    elif b == None:
                        result.append(a.data)
                        a = a.next
                    else:
                        if a.data <= b.data:
                            result.append(a.data)
                            a = a.next
                        else:
                            result.append(b.data)
                            b = b.next

                return result.head

        def mergeSort(self, h):

                # Base case if head is None
                if h == None or h.next == None:
                        return h

                # get the middle of the list
                middle = self.getMiddle(h)
                nexttomiddle = middle.next

                # set the next of middle node to None
                middle.next = None

                # Apply mergeSort on left list
                left = self.mergeSort(h)

                # Apply mergeSort on right list
                right = self.mergeSort(nexttomiddle)

                # Merge the left and right lists
                sortedlist = self.sortedMerge(left, right)
                return sortedlist

        # Utility function to get the middle
        # of the linked list
        def getMiddle(self, head):
                if (head == None):
                        return head

                slow = head
                fast = head

                while (fast.next != None and
                        fast.next.next != None):
                        slow = slow.next
                        fast = fast.next.next

                return slow

# Utility function to print the linked list
def printList(head):
        if head is None:
                print(' ')
                return
        curr_node = head
        while curr_node:
                print(curr_node.data, end = " ")
                curr_node = curr_node.next
        print(' ')

def genList(li, n):
    for i in range(0, n):
        li.append(random.randint(1, 1000))

# Driver Code
if __name__ == '__main__':
        li = LinkedList();

        # Let us create a unsorted linked list
        # to test the functions created.
        # The list shall be a: 2->3->20->5->10->15
        genList(li, 10000)

        # Apply merge Sort
        li.head = li.mergeSort(li.head)
        # print ("Sorted Linked List is:")
        # printList(li.head)

# This code is contributed by Vikas Chitturi
```

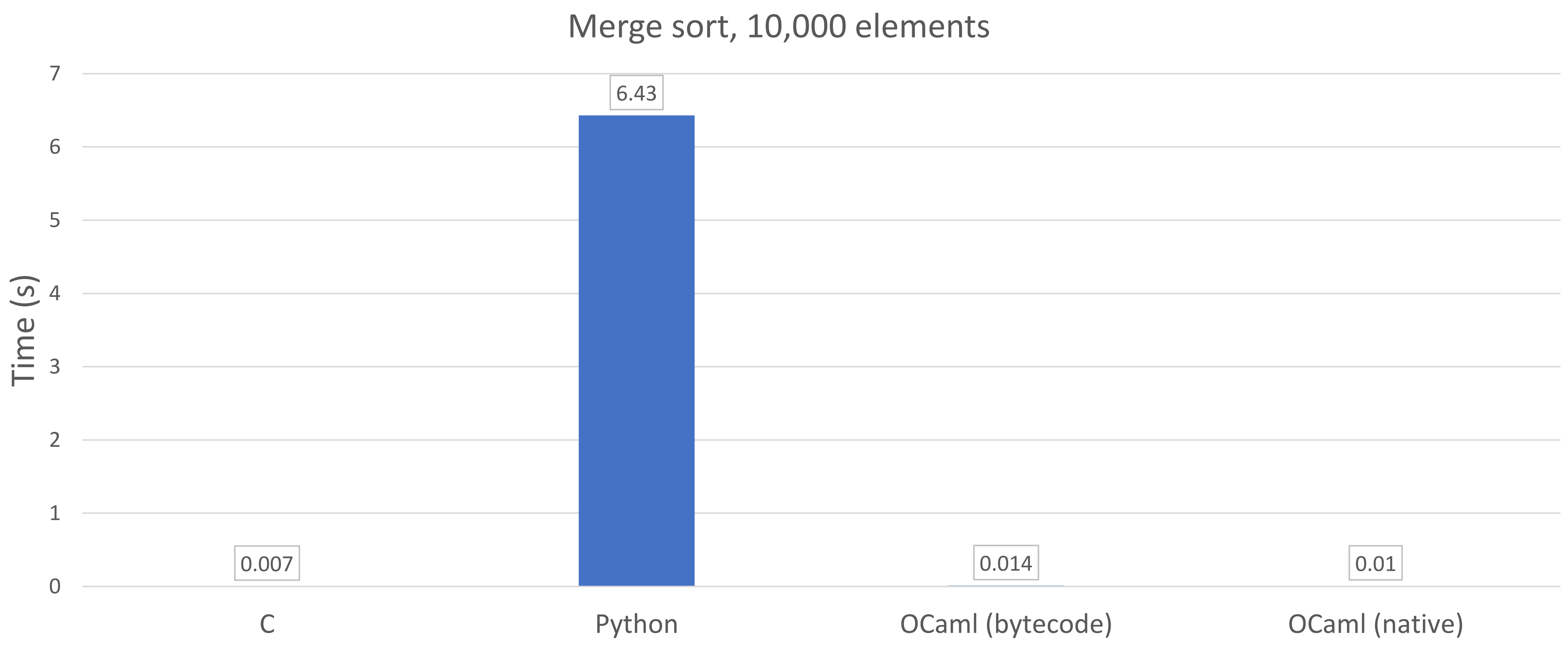
Python



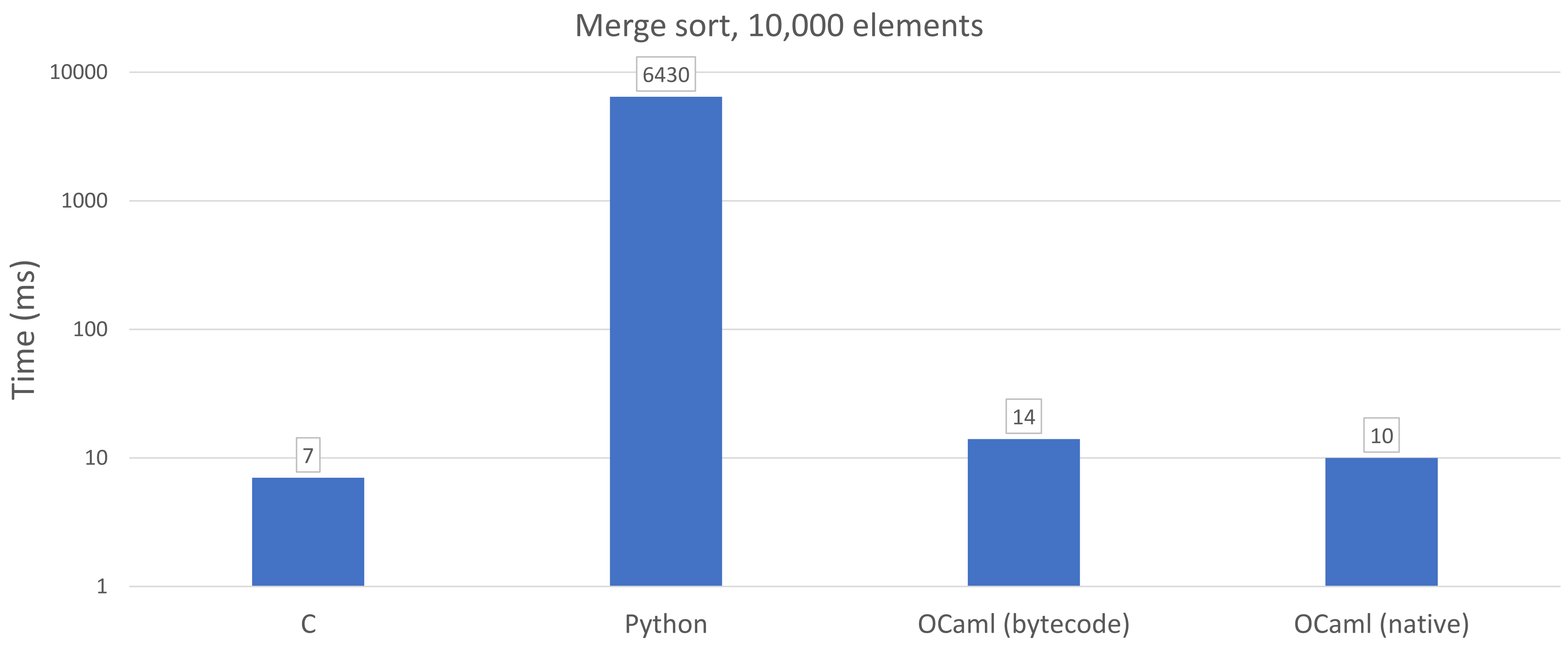
```ocaml
let rec split l (a, b)=
  match l with
  | [] -> (a, b)
  | [x] -> (x::a, b)
  | av::bv::t -> split t (av::a, bv::b)

let rec merge a b m =
  match (a, b) with
  | ([], _) -> (List.rev m) @ b
  | (_, []) -> (List.rev m) @ a
  | (av::at, bv::bt) ->
     if av < bv then
       merge at (bv::bt) (av::m)
     else
       merge (av::at) bt (bv::m)

let rec mergesort l =
  match l with
  | [] -> []
  | [x] -> [x]
  | _ ->
     let (a, b) = split l ([], []) in
     let a = mergesort a in
     let b = mergesort b in
     merge a b []

let rec printList l =
  match l with
  | [] -> ()
  | x::t -> (Printf.printf "%d " x; printList t)

let _ = Random.self_init ()
let l = mergesort (List.init 10000 (fun _ -> Random.int 10000))
```

OCaml

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



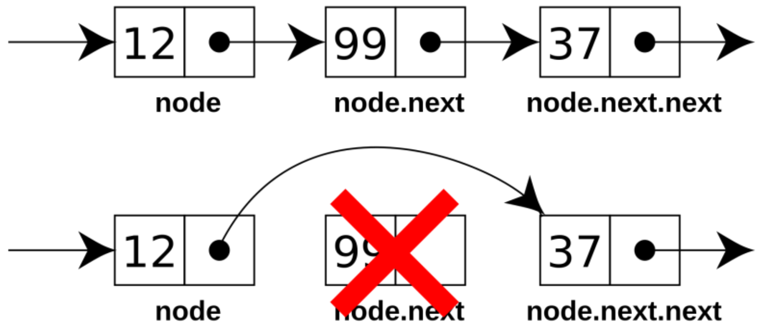
Knowing about the language and how it's translated can help you write faster code



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).
- Rocq
  ```
  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

OCaml ?

C?

Rust?

Today and Thursday

1. Programming Language overview/history
2. Programming Language paradigms
3. Break: administrivia
4. Programming Language implementation (compilers and interpreters)
5. OCaml introduction

Course Staff

- Instructor: Stefan Muller
    - Office Hours: Thur., 1-2pm (ITE 463)
- TA: Ricky Cheng
    - Office Hours: TBA

HuskyCT

- Syllabus
- Course schedule
- Important resources
- Assignments
- Assignment submission (through Gradescope)

Other ways to get help

- Discord server
  - Invitation will be sent by email

	Discord	Office Hours	Email
General questions about lectures, logistics, etc.	✓	✓	
General discussion, clarifications, about HW questions	✓	✓	
Specific questions about your HW answers		✓	
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
- Using search engines and/or generative AI to get more clarification about course material is OK.
- ALL work submitted for credit MUST BE your own individual work.
- Not allowed:
  - Working together
  - Sharing answers
  - Looking for answers on the internet
  - Asking generative AI for code/answers

This is the short version: read the details in the syllabus

Course schedule

- Intro and overview (1 week)
- Functional Programming in OCaml (~4 weeks)
- Midterm #1
- PL theory and Lambda Calculus (~3 weeks)
- Spring Break
- Midterm #2
- Logic Programming in Prolog (~2 weeks)
- Object-Oriented Programming in Smalltalk (~1 week)
- Wrap-up (1 week)

Homeworks

- 7-8 homeworks, ~2 weeks each
  - HW 0 Out ~Thursday, Due 1/29
- Written and programming
- Work individually

Late Days:

- 6 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

- 2 Midterms (tentatively Mar. 3 and Apr. 7)
- Final (finals week)

- Details TBA

- (No using late days, sorry)

Grading

- 15% Homeworks
- 50% Midterms
- 35% Final

93-100	A
90-93	A-
87-90	B+
83-87	B
80-83	B-
77-80	C+
73-77	C
70-73	C-
…	
<60	F

Textbooks

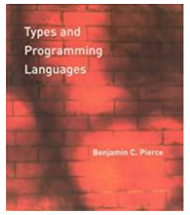
On Individual PLs:

- Clarkson. *OCaml Programming: Correct + Efficient + Beautiful*
- Blackburn, Bos, Striengnitz. *Learn Prolog Now!*

Online! See HuskyCT

For more math-y details:

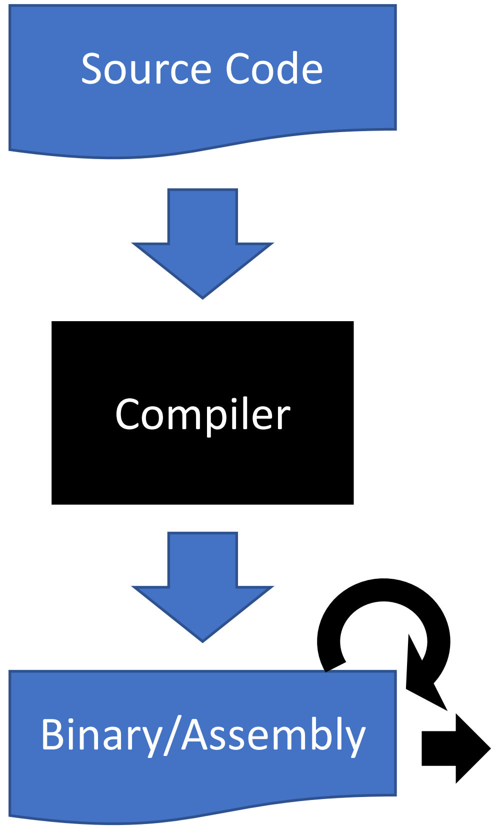
- Pierce. *Types and Programming Languages*
- Harper. *Practical Foundations for Programming Languages*



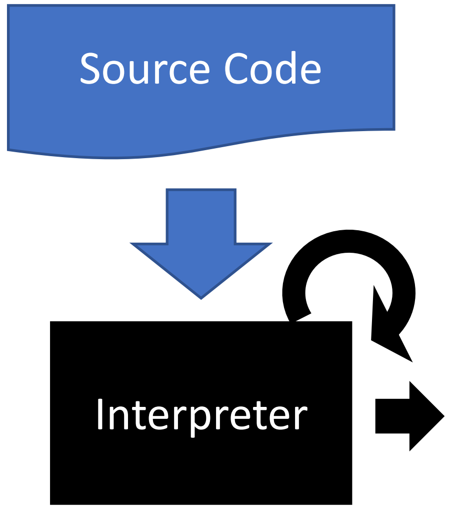
~~Today~~ and Thursday

1. Programming Language overview/history
2. Programming Language paradigms
3. Break: administrivia
4. Programming Language implementation (compilers and interpreters)
5. OCaml introduction

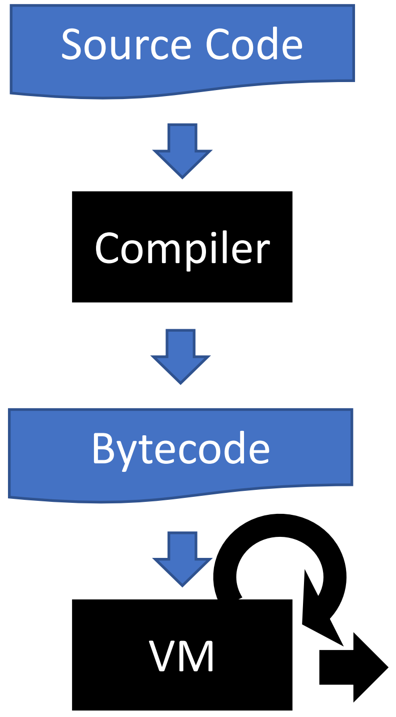
There are different ways of translating a programming language



Ex.: C, C++



Ex.: Python



Ex.: Java

“Go straight on to the roundabout; mind the lorries”

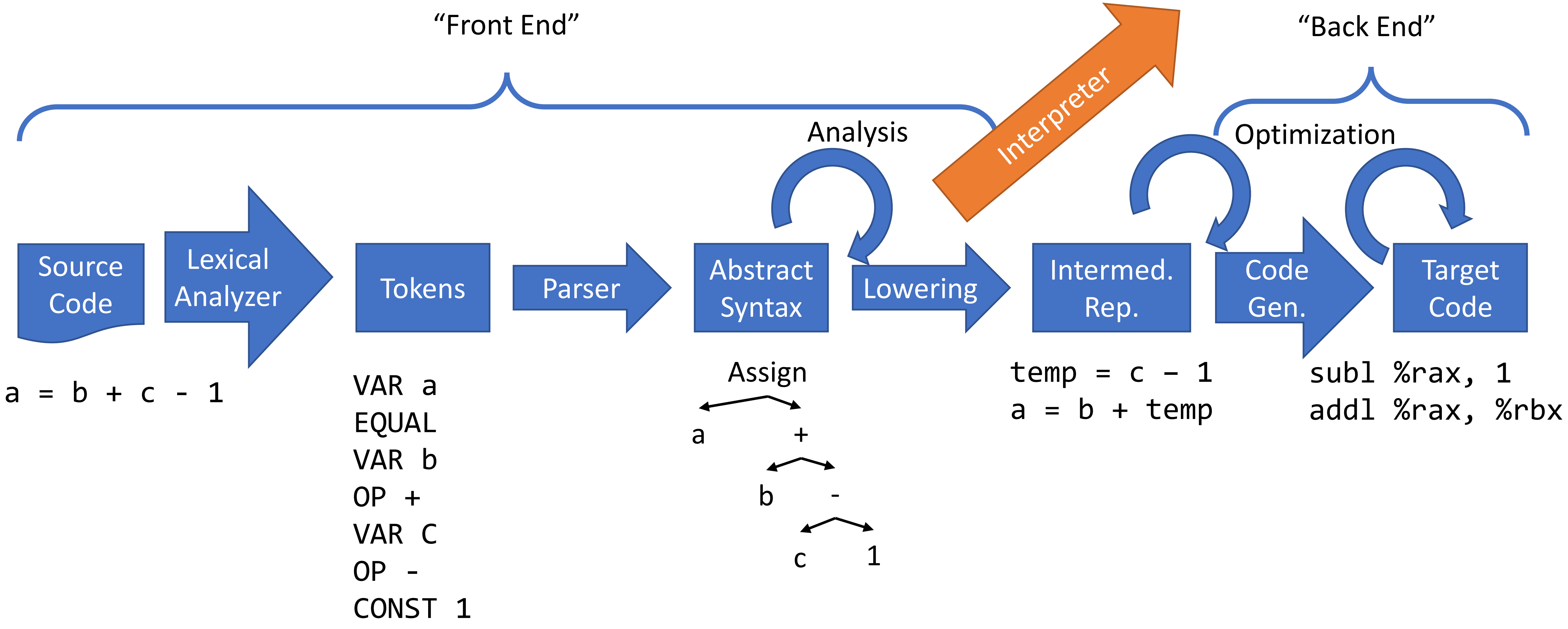
“OK, so keep going that way”;
“Here, you’re going to go straight ahead”

“It means ‘keep going until you get to this circular intersection; watch out for trucks.’”

Compilers vs. interpreters

- Compiler
  - Translates the program to a form executable by the machine (or assembly)
  - Compile, then can run the executable: compiler no longer involved
- Interpreter
  - Doesn’t translate to machine-readable format
    - Might compile to bytecode or intermediate representation
  - Runs (“interprets”) program directly
  - Can’t run without the interpreter

Compilers translate code in phases



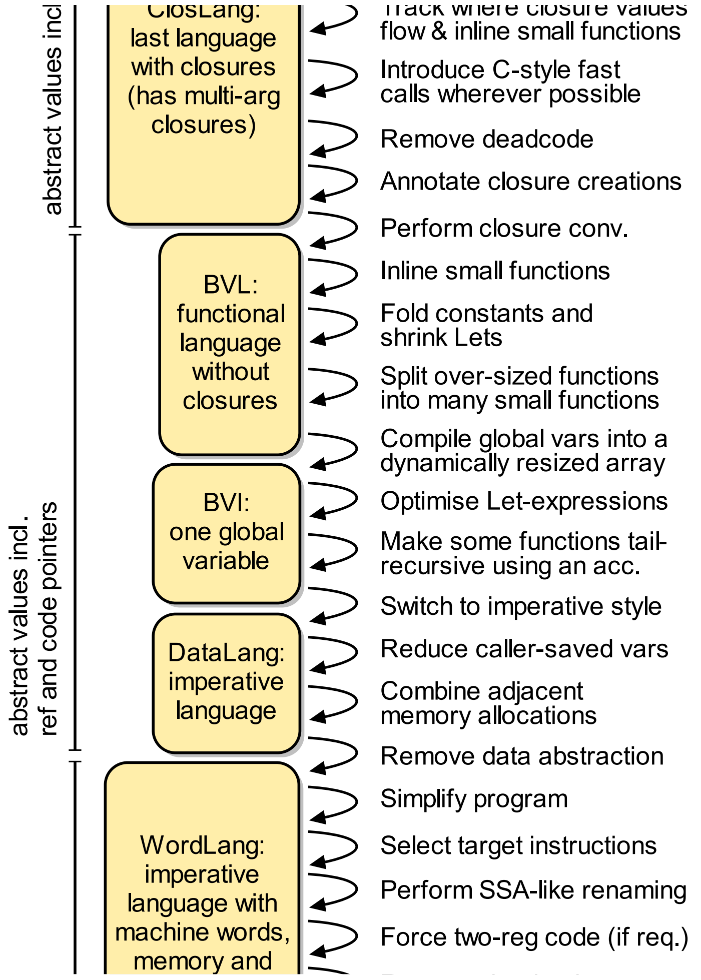
```
a = b + c - 1
```

```
VAR a
EQUAL
VAR b
OP +
VAR C
OP -
CONST 1
```

```
temp = c – 1
a = b + temp
```

```
subl %rax, 1
addl %rax, %rbx
```

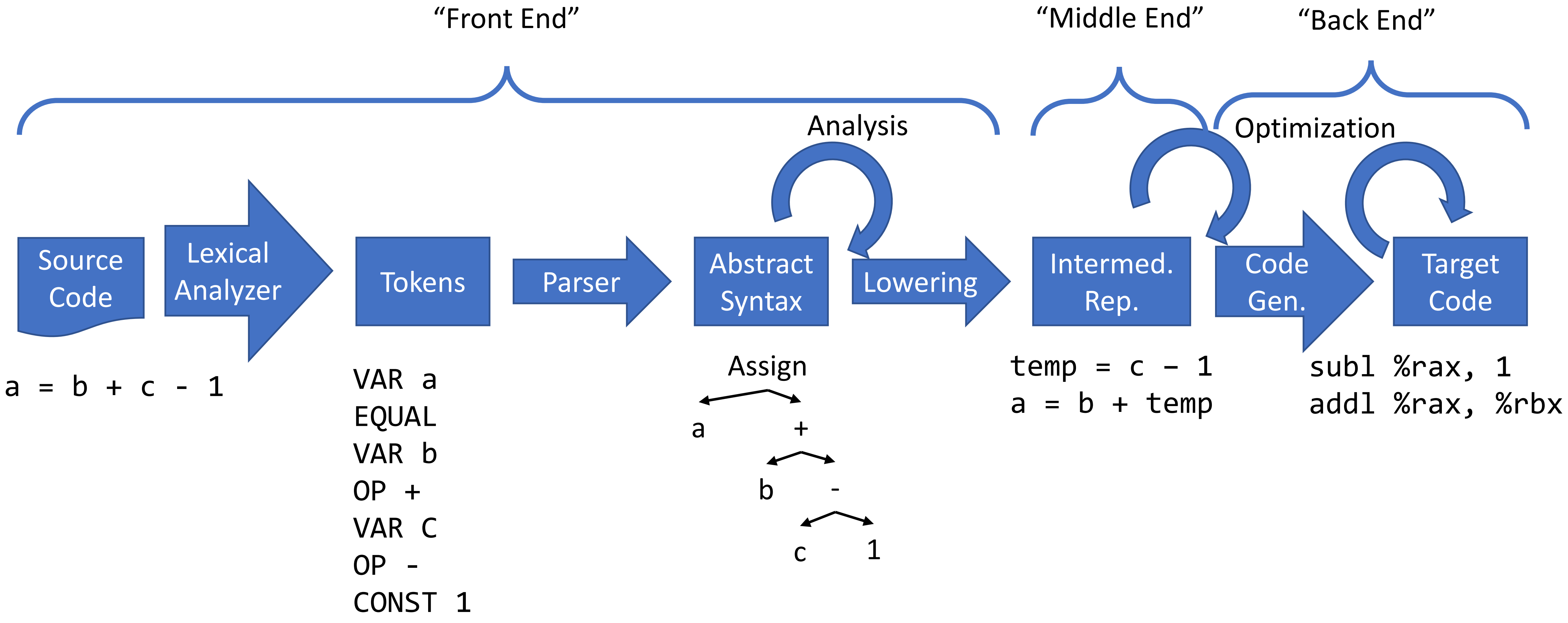
May have many more phases, several intermediate representations



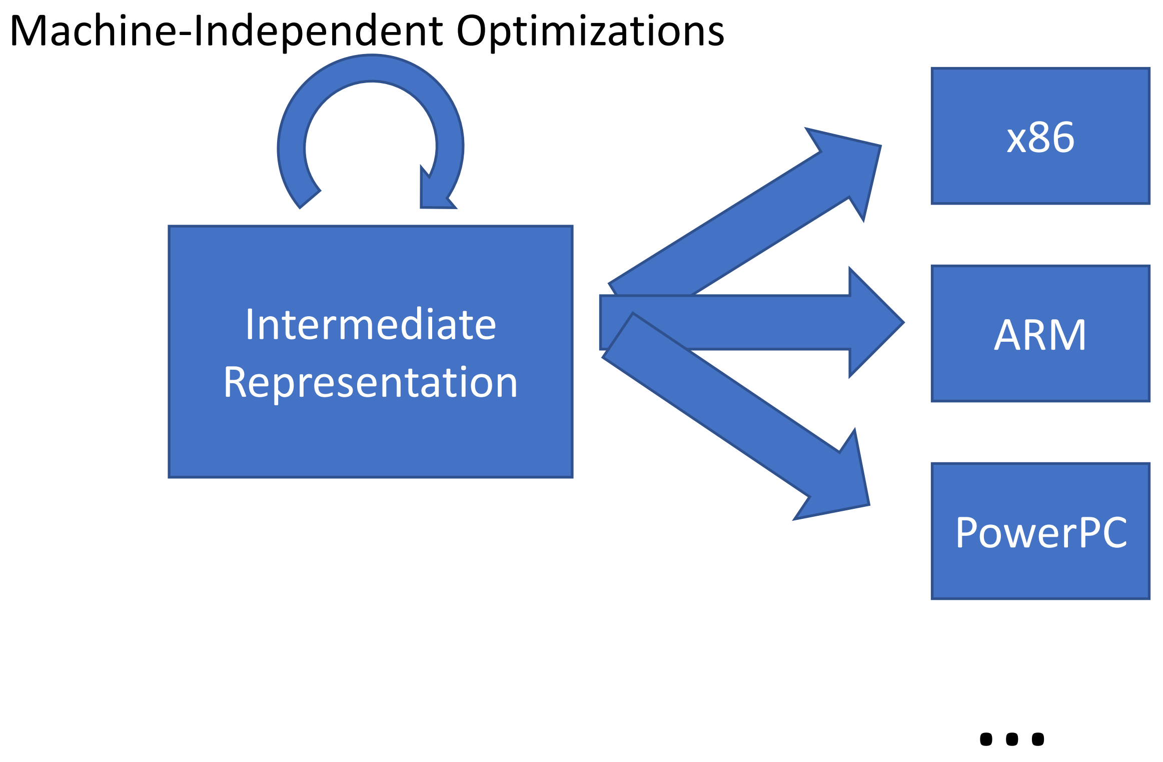
abstract values incl
ClosLang: last language with closures (has multi-arg closures)
Track where closure values flow & inline small functions
Introduce C-style fast calls wherever possible
Remove deadcode
Annotate closure creations
Perform closure conv.
BVL: functional language without closures
Inline small functions
Fold constants and shrink Lets
Split over-sized functions into many small functions
Compile global vars into a dynamically resized array
BVI: one global variable
Optimise Let-expressions
Make some functions tail-recursive using an acc.
Switch to imperative style
abstract values incl. ref and code pointers
DataLang: imperative language
Reduce caller-saved vars
Combine adjacent memory allocations
Remove data abstraction
WordLang: imperative language with machine words, memory and
Simplify program
Select target instructions
Perform SSA-like renaming
Force two-reg code (if req.)

CAKEML
A Verified Implementation of ML

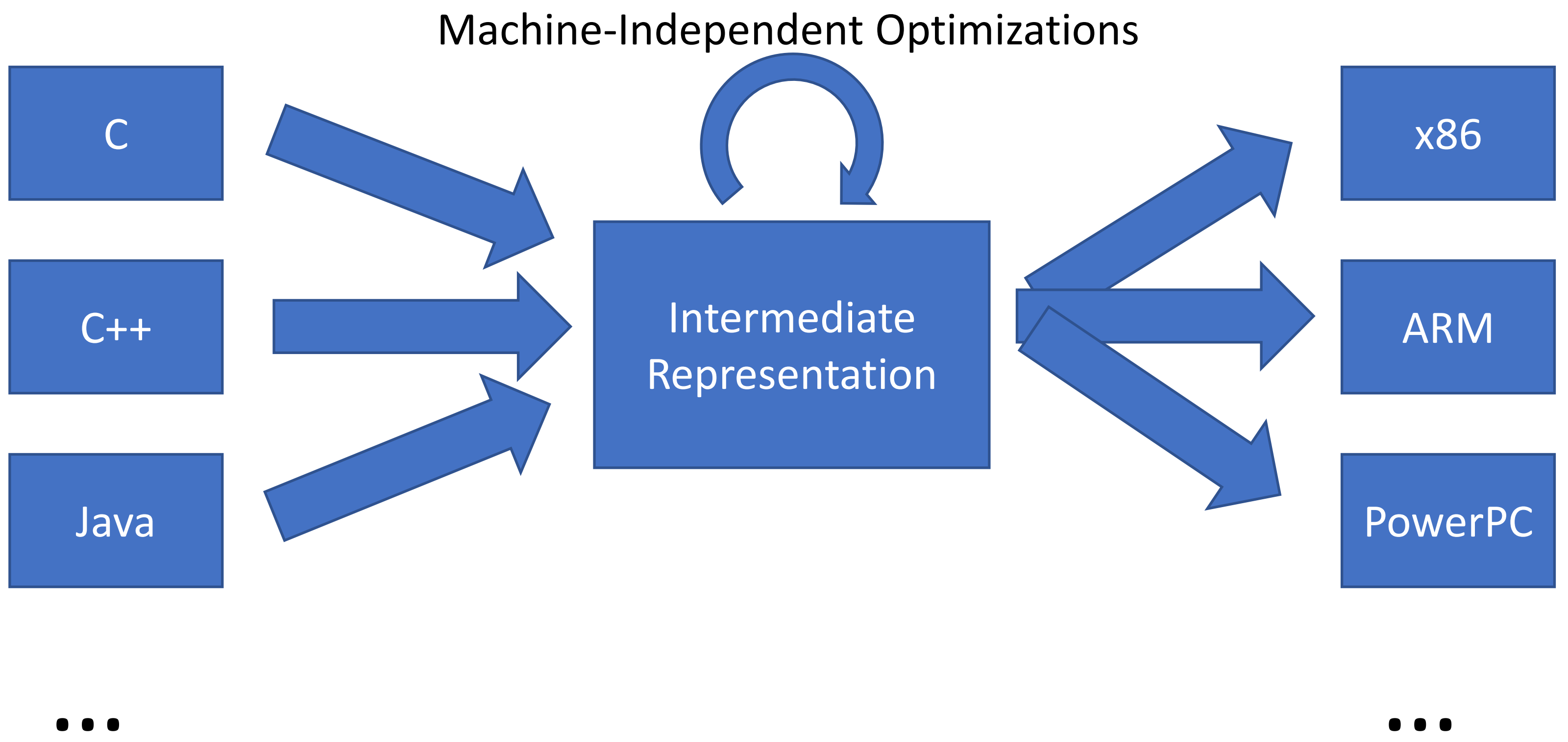
Front End is language specific
Back End is machine specific



Can (and usually do) swap out back ends to target different machines

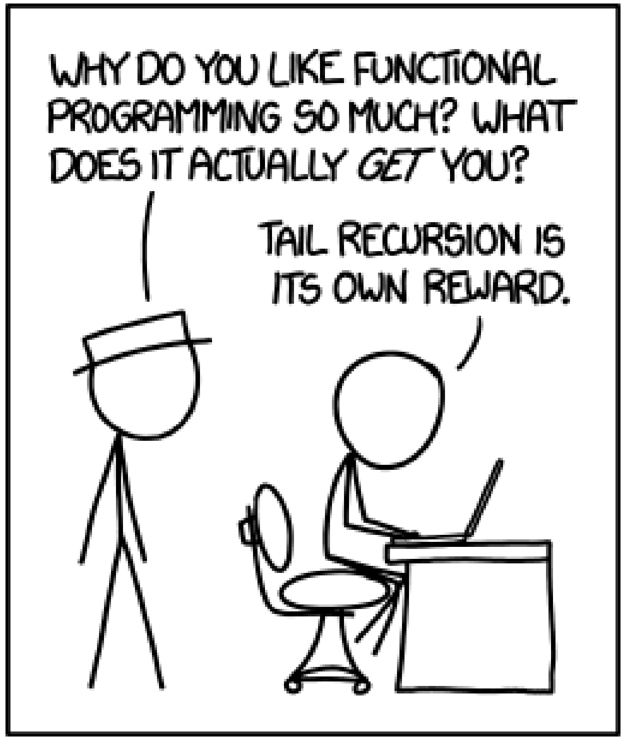

Machine-Independent Optimizations
Intermediate Representation
x86
ARM
PowerPC
…

Compiler collections also swap out front ends for different languages


Machine-Independent Optimizations
C
C++
Java
…
Intermediate Representation
x86
ARM
PowerPC
…

Functional Programming

- Strong mathematical foundations
- Very high-level
- Really elegant for expressing many algorithms


WHY DO YOU LIKE FUNCTIONAL PROGRAMMING SO MUCH? WHAT DOES IT ACTUALLY *GET* YOU?
TAIL RECURSION IS ITS OWN REWARD.

xkcd.com/1270

(Alt text: Functional programming combines the flexibility and power of abstract mathematics with the intuitive clarity of abstract mathematics)

OCaml

- Statically typed, functional
  - (also has imperative and object-oriented features)

- Strong, expressive type system
  - (makes implementing many data structures very easy)
- Type inference
  - `int x = 5;`
  - `x = 5`

OCaml

- Probably the most used functional language
- First appeared 1996
  - “ML family” of languages (Standard ML, F#) goes back to the 1970s

- Industrial-strength compiler
  - Actively maintained
  - Lots of libraries (standard and 3rd-party)

Jane Street