# CSCI 334:
# Principles of Programming Languages

## Lecture 11: Language Architecture

Instructor: Dan Barowy

Williams

## Topics

How do programs run?

Garbage collection

## Your to-dos

1. Lab 6, **due Wednesday 3/19** (partner lab)
2. Review for midterm **next week**.
3. Start reading *Gumptionology 101* and *Proof by Reduction*.

## Computer Science Movie Night: Hidden Figures


**Thursday, March 13 at 7pm Wege Auditorium**

Casual viewing. Feel free to come late, leave early, and work during.

**Benefits:**

- **Fun!**
- **Snacks!**
- Learn a little history and be inspired by the science montages!
- Did I mention snacks?!!

Co-sponsored by faculty, UnICS, and WiCS

# Announcements

- CS Colloquium this **Friday, Mar 14 @ 2:35pm in Wege Auditorium (TCL 123)**

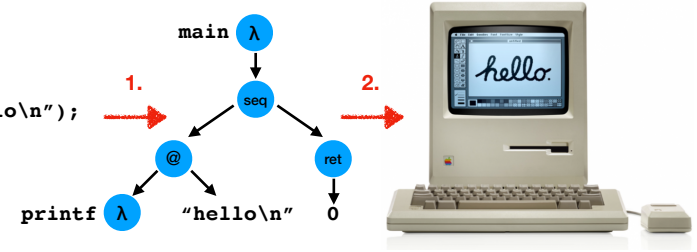
Effective Evaluations of Large Language Models for Complex Problems
Prof. Niranjan Balasubramanian (Stony Brook)

Evaluation efforts are central to understanding and improving the capabilities of Large Language Models on complex problems. In the first part of this talk, I will present a set of ideas to create effective complex reasoning tasks for a range of tasks such as question answering, theory-of-mind analyses, procedure understanding, and biomedical natural language inference. In the second half, I will present our recent work on developing a rigorous benchmark that can test automation for solving complex tasks involving commonly used apps.

# How do programs run?

# How do programs run?

```
int main() {
  printf("hello\n");
  return 0;
}
```







1. lexical analysis ("front-end")
2. evaluation ("back-end")

# Front-end: the parser

A **parser** is a **function** that takes as input a string of symbols conforming to the rules of a formal grammar. If the string is not a valid sentence in the language, the parser **rejects** the string. If the string is a valid sentence in the language, the parser **accepts** the string and outputs a data structure that **represents the meaning of the sentence**.

For programming languages, meaning is generally represented in the form of an **abstract syntax tree** (AST). In an AST, conventionally, interior nodes are operations, and leaves are data.

# Front-end: the parser

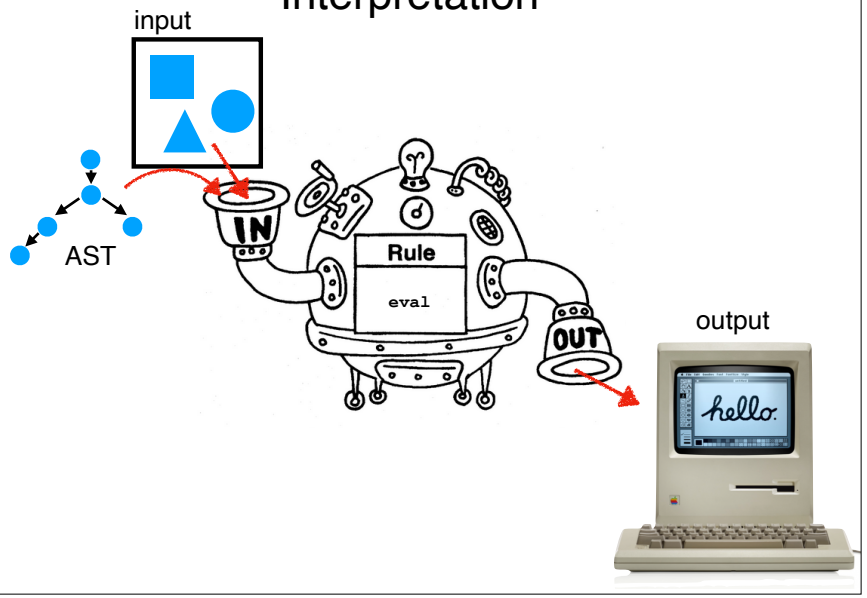
The subject of today's lesson.

# Back-end: the evaluator

There are two kinds of back-end:

1. Interpreter
2. Compiler

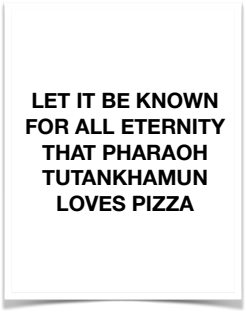
# Interpretation




# Interpretation Downsides

- Usually (very) slow

  (often 100-200x slower than compilation)



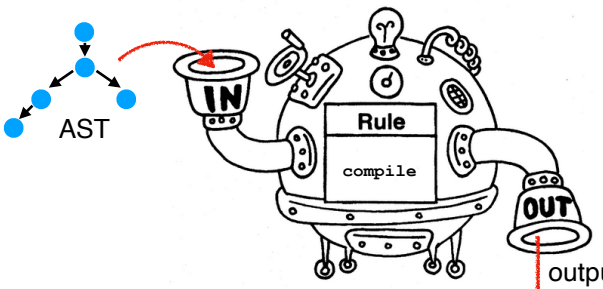
## Interpretation Advantages

- An interpreter is “just a program” so debugging a language is the same as debugging any other program.

## Some interpreted languages

- Shell (e.g., bash)
- Python
- Ruby
- MATLAB
- R
- (sort of) Java and JavaScript

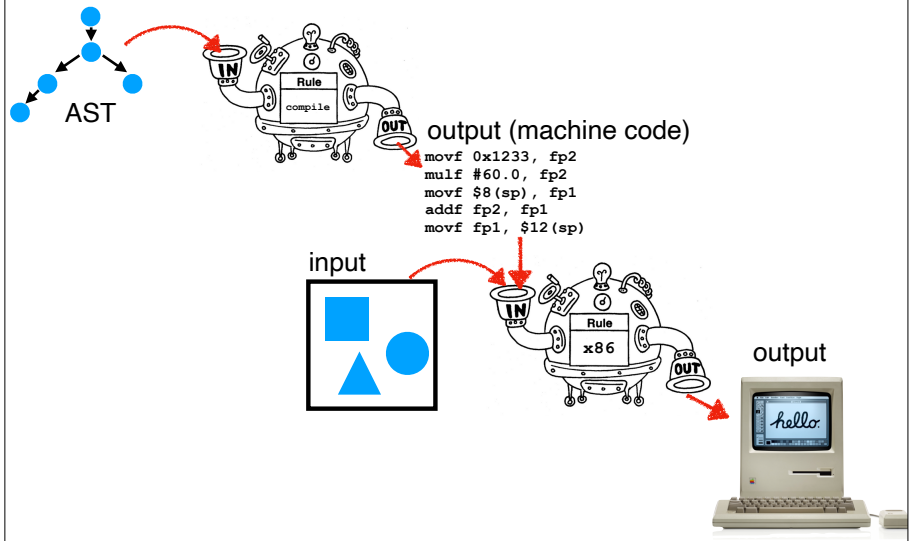
## Compilation




output (machine code)

```
movf 0x1233, fp2
mulf #60.0, fp2
movf $8(sp), fp1
addf fp2, fp1
movf fp1, $12(sp)
```

## Compilation

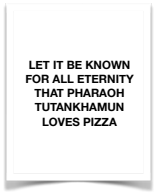


## Some compiled languages

- C
- C++
- Go
- FORTRAN
- Java (sort of)
- C# (ditto)
- F# (ditto)

## Compilation Advantages

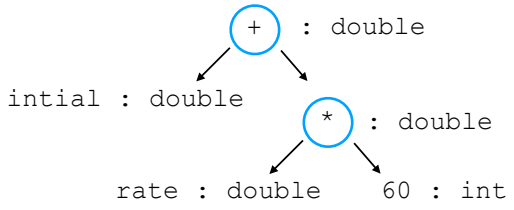
- Usually (very) fast
  (only 1.5-2X slower than hand-optimized assembly code)
- Compiled program is in machine (binary) format; difficult to debug the language itself.



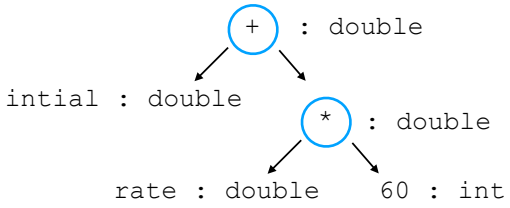

## Compilation Example

## Parsing

```
double position = initial + (rate * 60)
```



## Intemediate Representation




```
temp1 = convert_int_to_double(60)
temp2 = mult(rate, temp1)
temp3 = add(initial, temp2)
position = temp3
```

## “Optimization”

```
temp1 = convert_int_to_double(60)
temp2 = mult(rate, temp1)
temp3 = add(initial, temp2)
position = temp3
```

```
temp1 = mult(rate, 60.0)
position = add(initial, temp1)
```

## Instruction Selection

```
temp1 = mult(rate, 60.0)
position = add(initial, temp1)
```

```
movf rate, fp2
mulf #60.0, fp2
movf initial, fp1
addf fp2, fp1
movf fp1, position
```

## Compilation Downsides

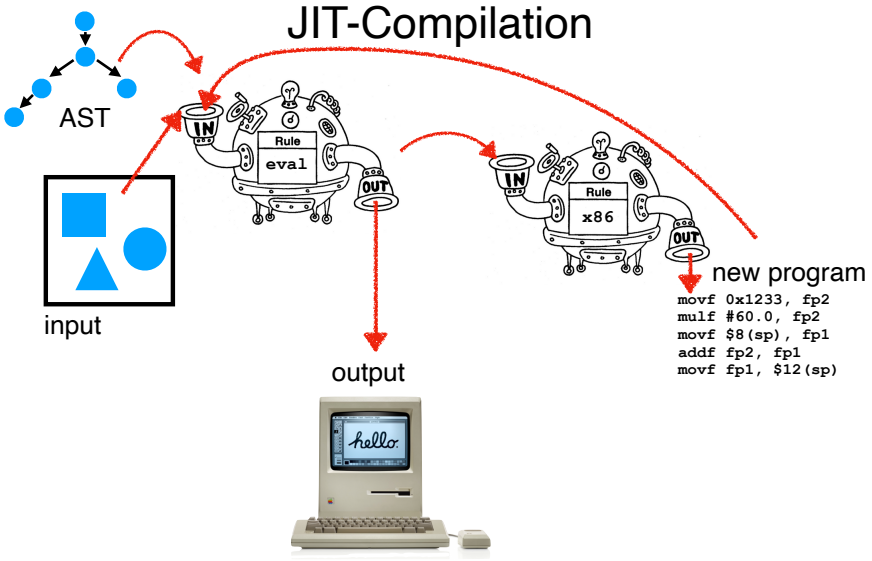
- Compilation can take a long time


- Cannot modify program without source code.
- Hard to evolve language; compilers are complex.

## Some hybrid (JIT) languages

- Java (C#, F#)
- JavaScript

## JIT-Compilation




## History

- Surprisingly, compilers were invented before interpreters.
- More obvious to early engineers.

## Compilers: History

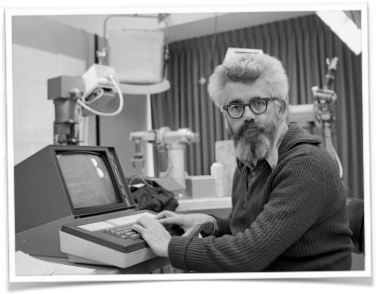
- Invented by Grace Hopper in 1952 while working on the A-0 and FLOW-MATIC languages.
- Work eventually became the COBOL programming language, still widely in use today.

# Compilers: History

I used to be a mathematics professor. At that time I found there were a certain number of students who could not learn mathematics. I then was charged with the job of making it easy for businessmen to use our computers. I found it was not a question of whether they could learn mathematics or not, but whether they would. […] They said, 'Throw those symbols out — I do not know what they mean, I have not time to learn symbols.' I suggest a reply to those who would like data processing people to use mathematical symbols that they make them first attempt to teach those symbols to vice-presidents or a colonel or admiral. I assure you that I tried it. — Grace Hopper

# Interpreters: History

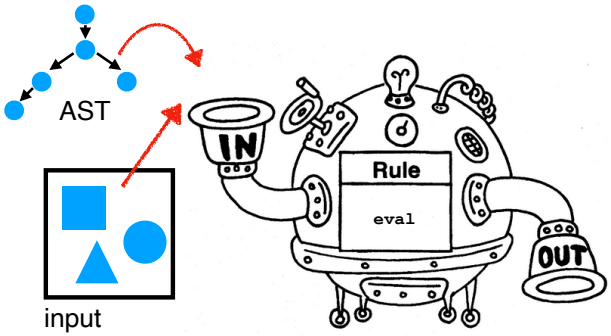
- Invented by John McCarthy in 1958 while working on LISP.
- Invented as a byproduct of McCarthy's thinking about computation from first principles.
- McCarthy wanted to build computers that could *think*!



- LISP was too resource hungry for most uses at the time.

# A PL is a program

that runs "user" programs




Remember: user programs are **data**!

When you write a PL, you have the opportunity to act on that data.

You can automate away problems like…

# Garbage collection

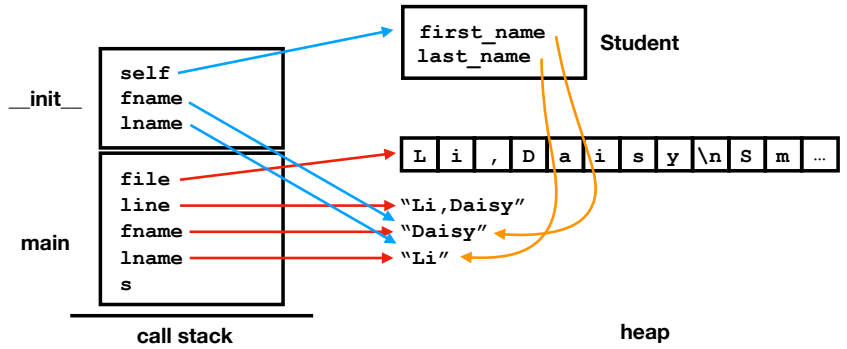
A **garbage collection algorithm** is an algorithm that determines whether the storage, occupied by a value used in a program, **can be reclaimed for future use**. Garbage collection algorithms are often tightly integrated into a programming language **runtime**.

# Garbage collection

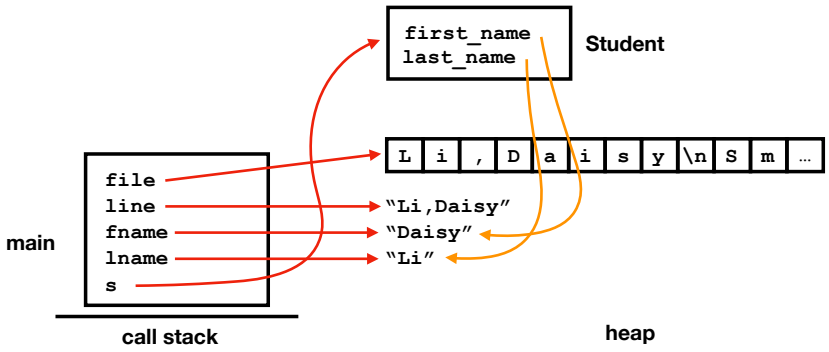
```
$ head -n5 roster.csv
Angoluan,Simon
Avila-Hernandez,Natalia
Canora,Nick
Cape,Noah
Chen,Nicholas
```

```
class Student:
    first_name: str
    last_name: str

    def __init__(self, first_name: str, last_name: str):
        self.first_name = first_name
        self.last_name = last_name

    def canonical(self) -> str:
        return self.first_name + " " + self.last_name

if __name__ == '__main__':
    with open("roster.csv") as file:
        for line in file:
            (fname,lname) = line.rstrip().split(",")
            s: Student = Student(fname, lname)
            print(s.canonical())
```

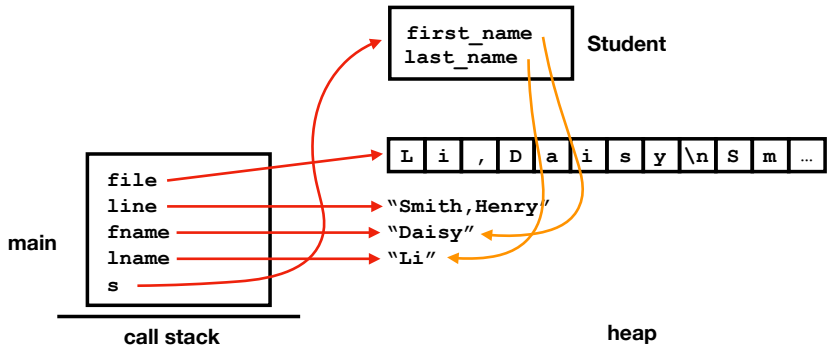
# If we did not have GC



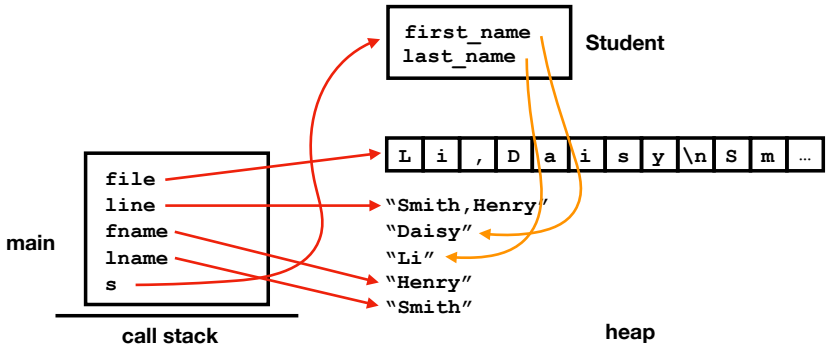

# If we did not have GC



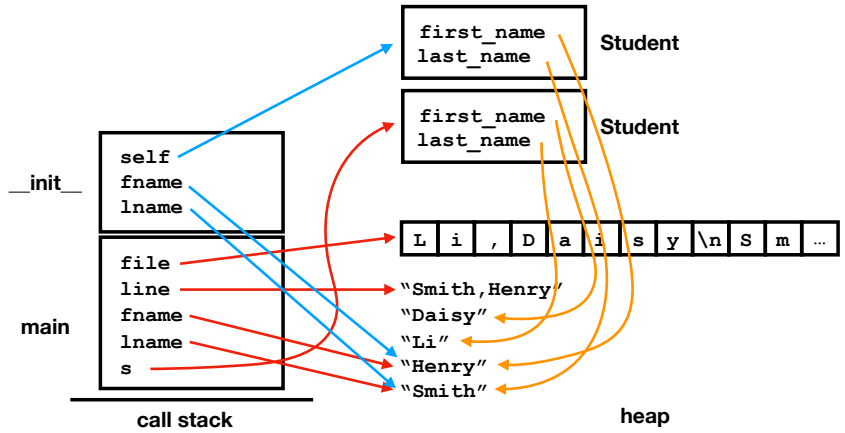

# If we did not have GC



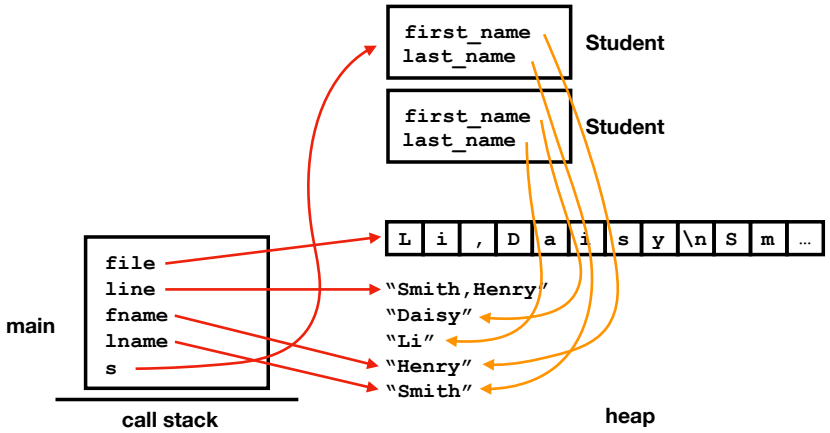
# If we did not have GC



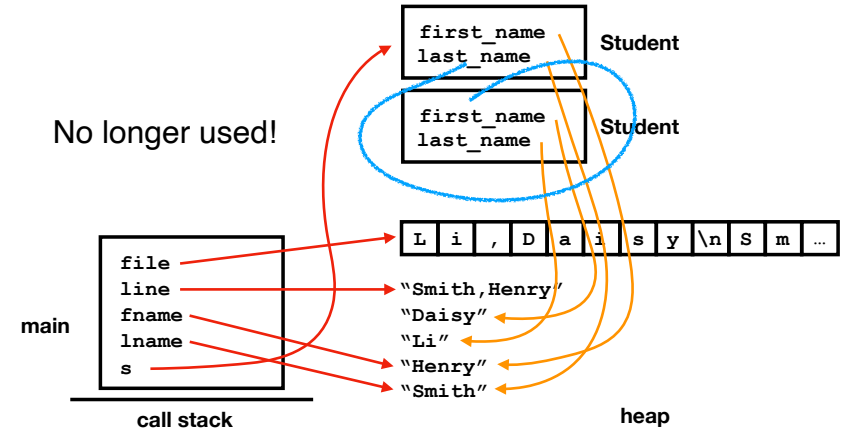

# If we did not have GC




# If we did not have GC



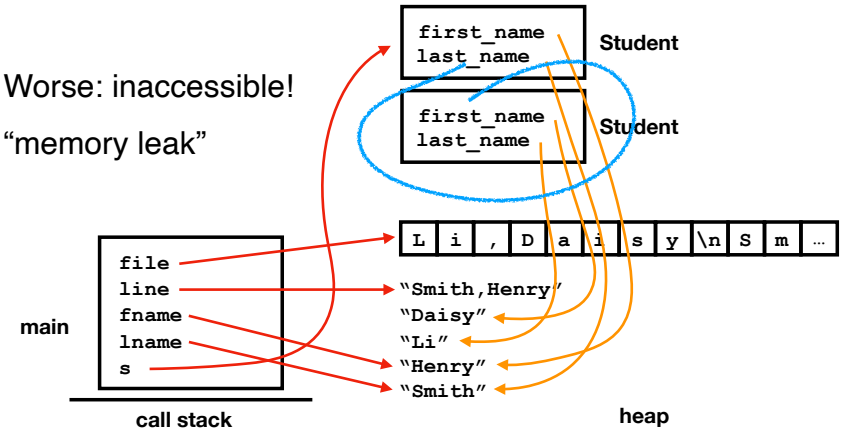

# If we did not have GC



# If we did not have GC

Worse: inaccessible!

"memory leak"

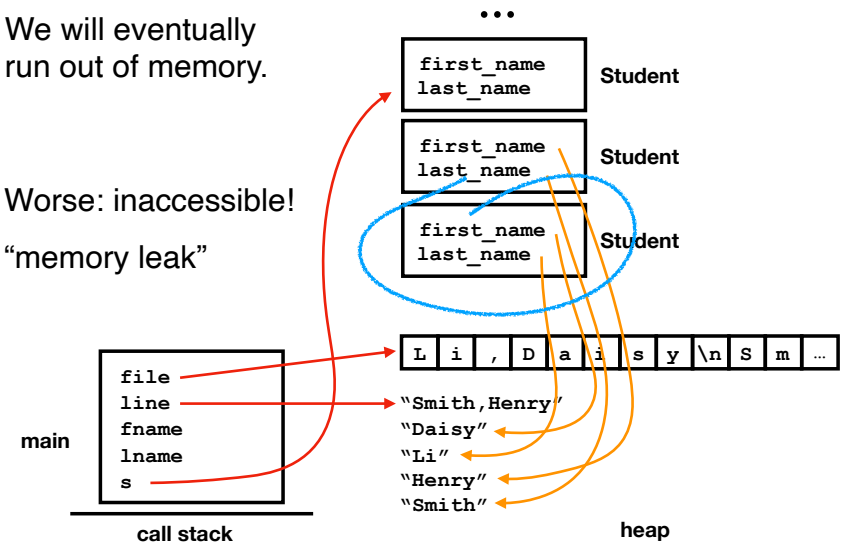



# If we did not have GC

We will eventually run out of memory.

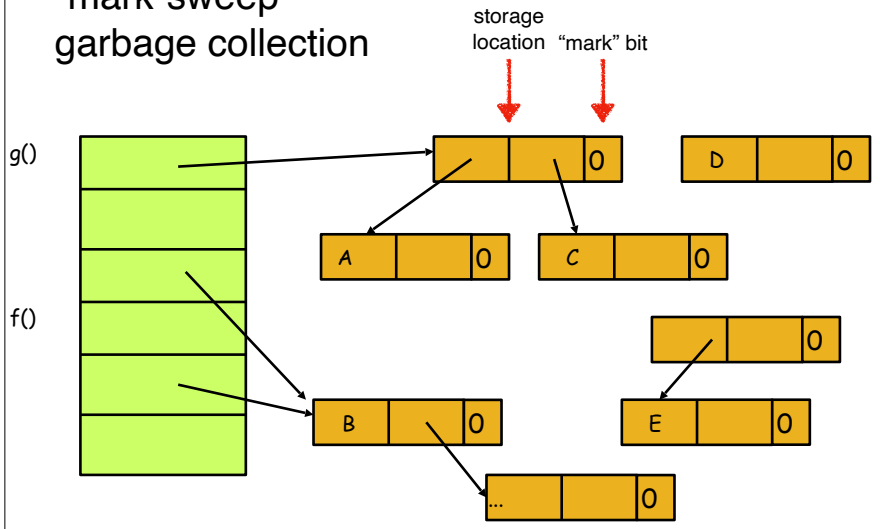
Worse: inaccessible!

"memory leak"

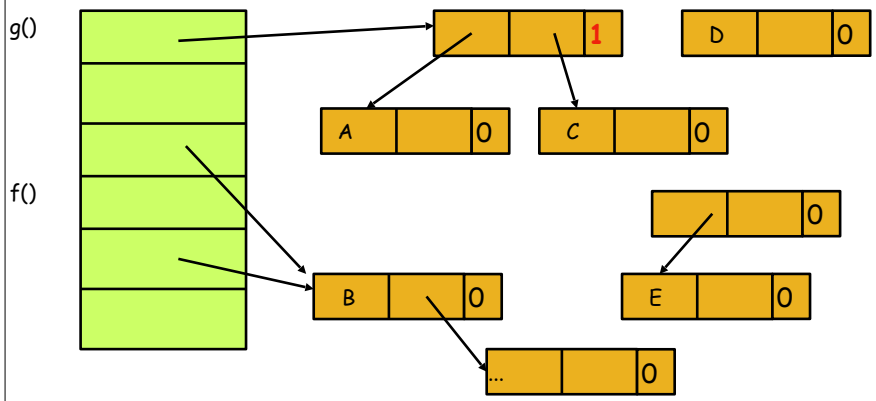



John McCarthy

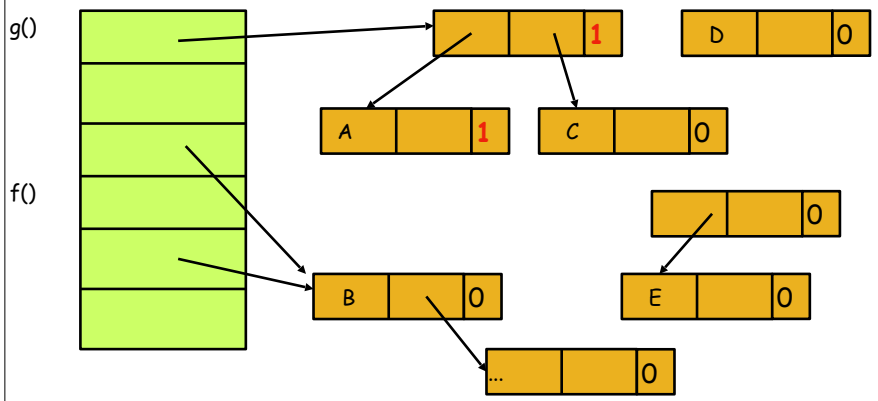
# "mark-sweep" garbage collection



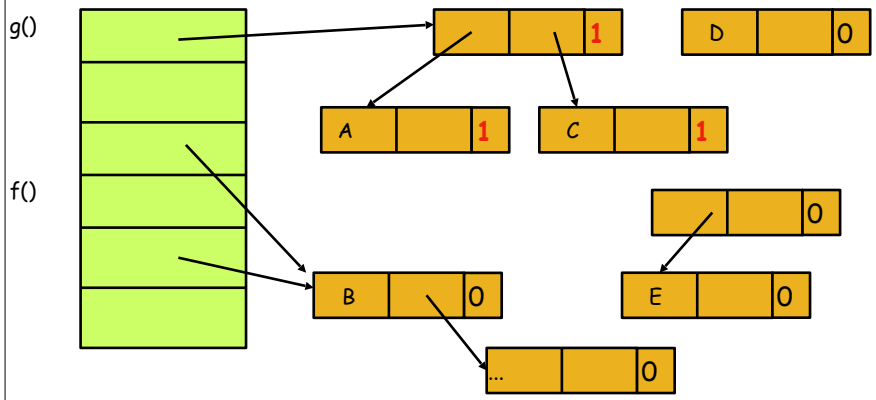
## 1. Mark reachable cells



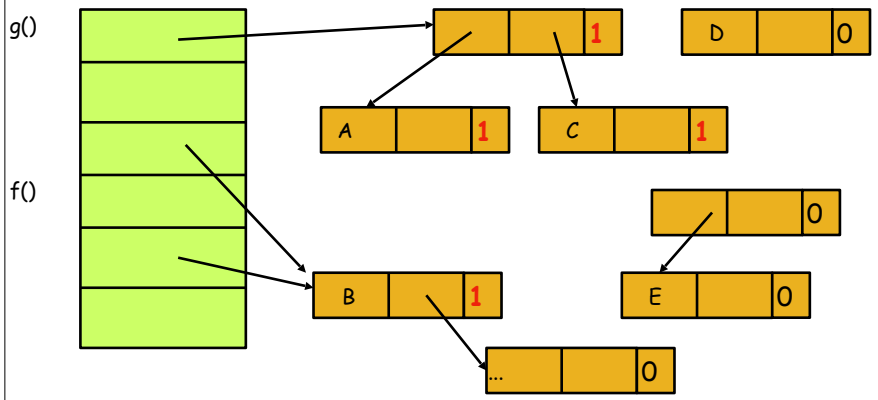

## 1. Mark reachable cells



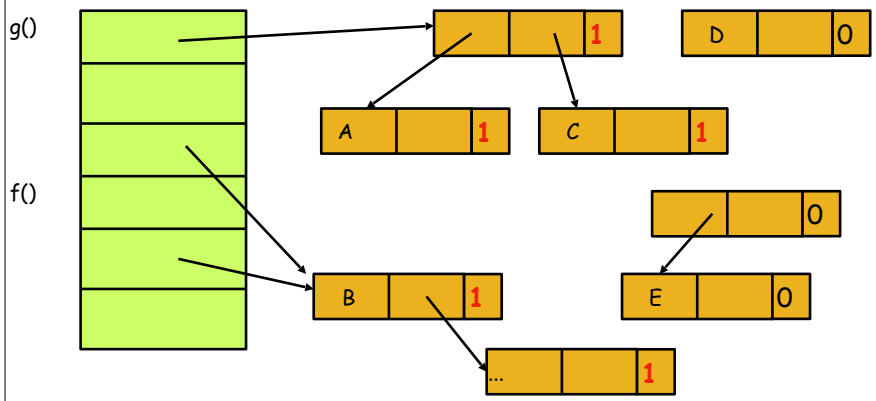

## 1. Mark reachable cells



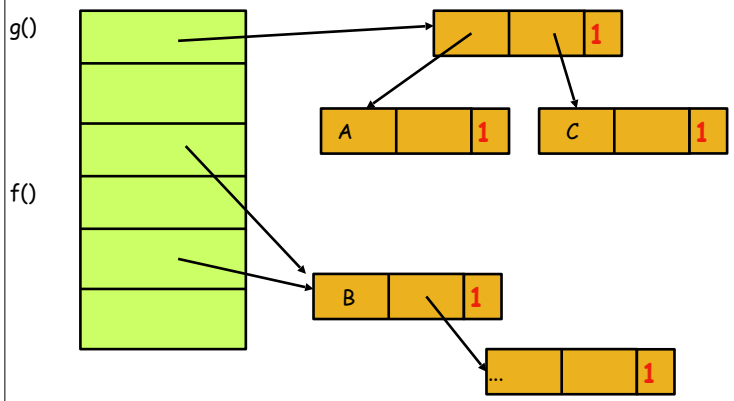

## 1. Mark reachable cells



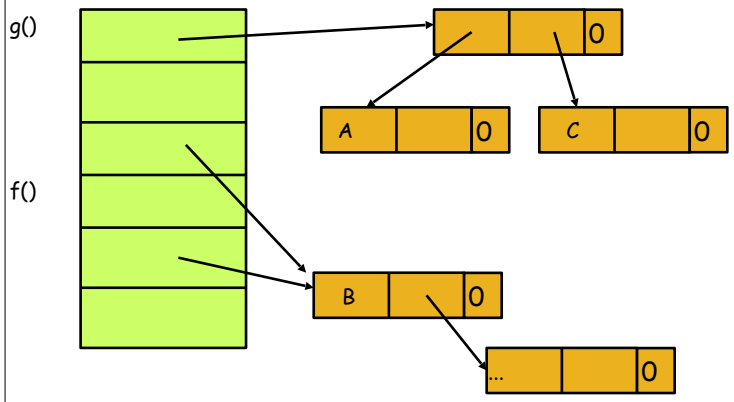
## 1. Mark reachable cells




## 2. Free (“sweep”) unreachable cells




## 3. Clear tags




## Garbage Collection

is **undecidable**.

Wait… **what**?!!!

# Recap & Next Class

## Today:

Language architecture

Garbage collection

## Next class:

Midterm review