CSCI 334:
Principles of Programming Languages

Lecture 10: Computability, part 2

Instructor: Dan Barowy

Williams

Topics

Garbage collection

Halting problem

Reduction proofs

Your to-dos

1. Lab 5, **due Sunday 10/15** (partner lab)
2. As a **part of lab 5**, read *How to Fix a Motorcycle.*

Announcements

- **Midterm exam**, in class, Thursday, Oct 19.
- **Field trip to WCMA**, Thursday, Nov 2.
- Colloquium: **What I Did Last Summer (Research Edition)**, 2:35pm in Wege Auditorium.

Announcements

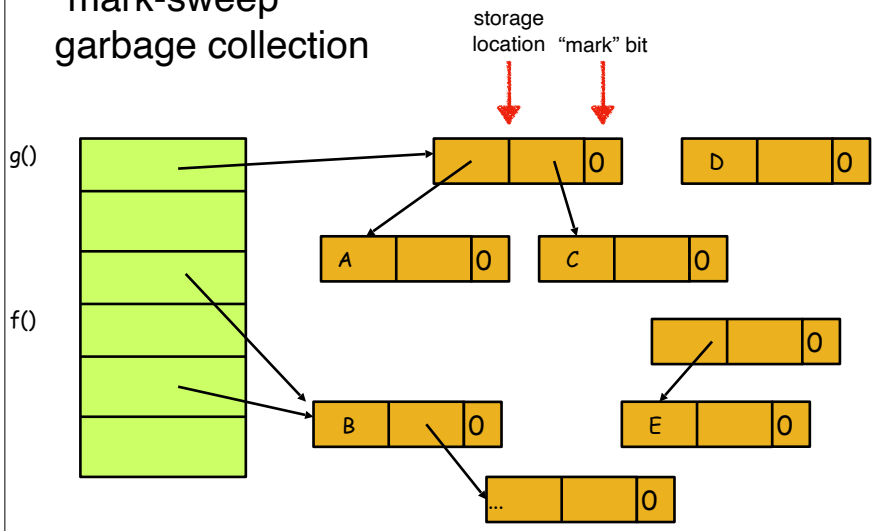
- **TA Applications** due Friday, Oct 27.
- **TA Evaluation** forms due Friday, Oct 27.
-

Garbage collection

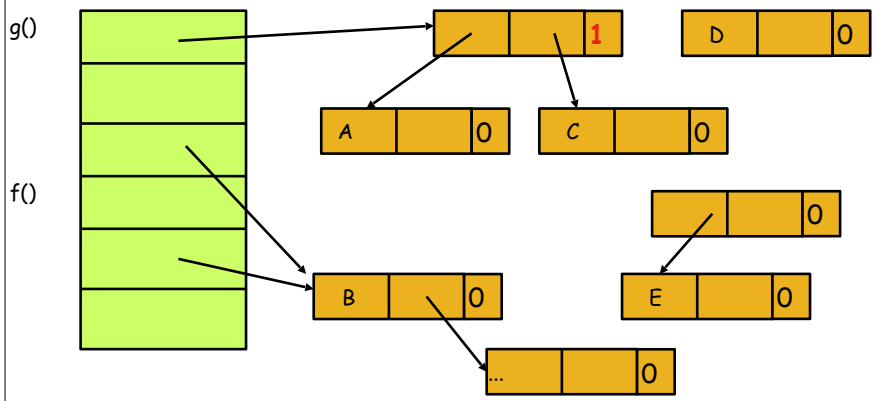
A **garbage collection algorithm** is an algorithm that determines whether the storage, occupied by a value used in a program, **can be reclaimed for future use**. Garbage collection algorithms are often tightly integrated into a programming language **runtime**.

John McCarthy

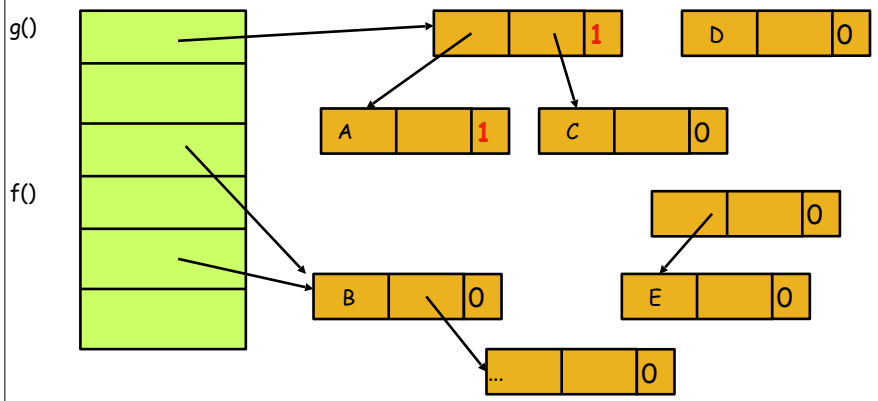
"mark-sweep" garbage collection



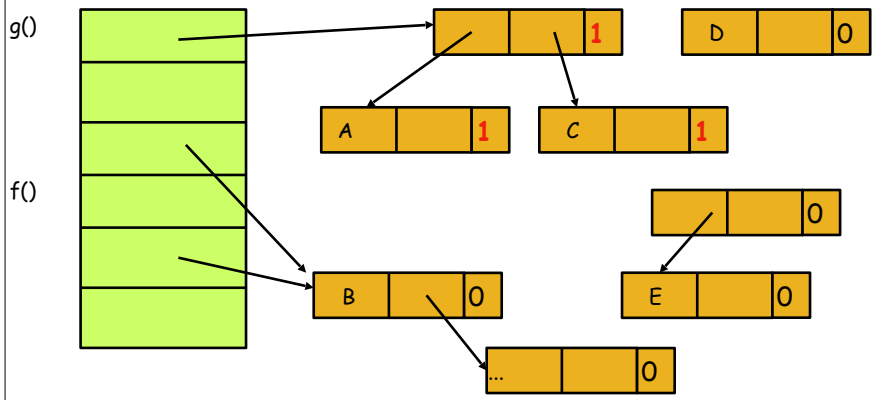
1. Mark reachable cells



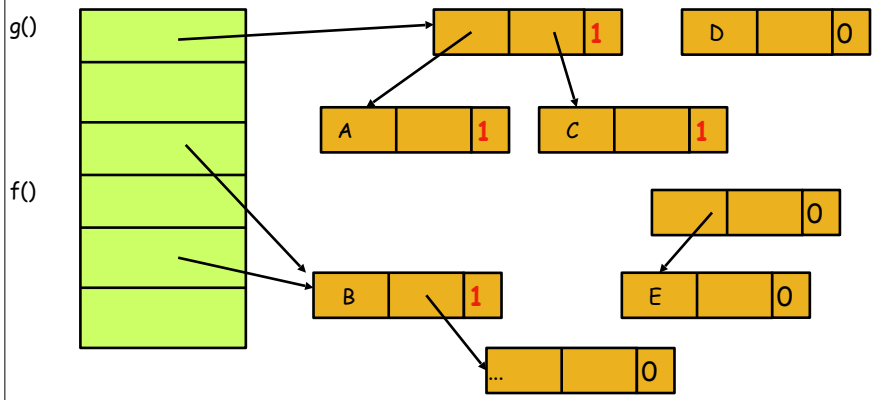
1. Mark reachable cells



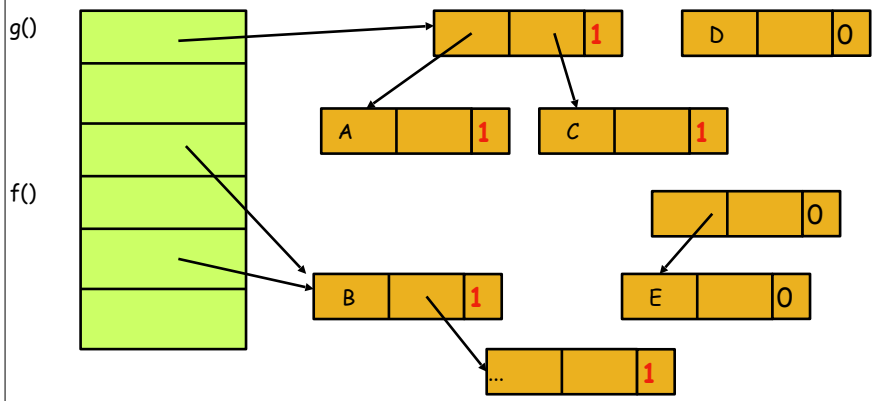
1. Mark reachable cells



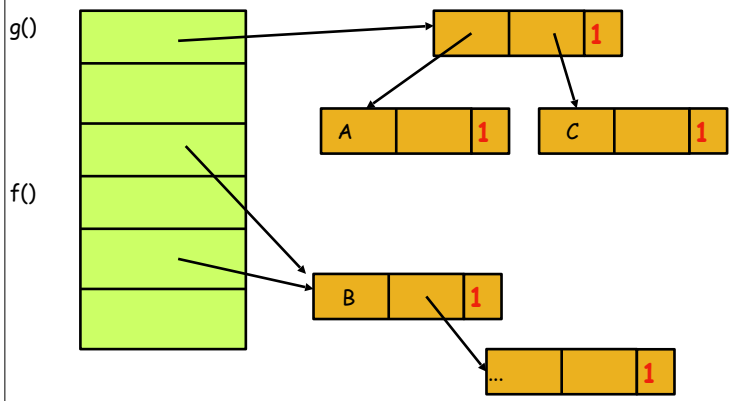
1. Mark reachable cells



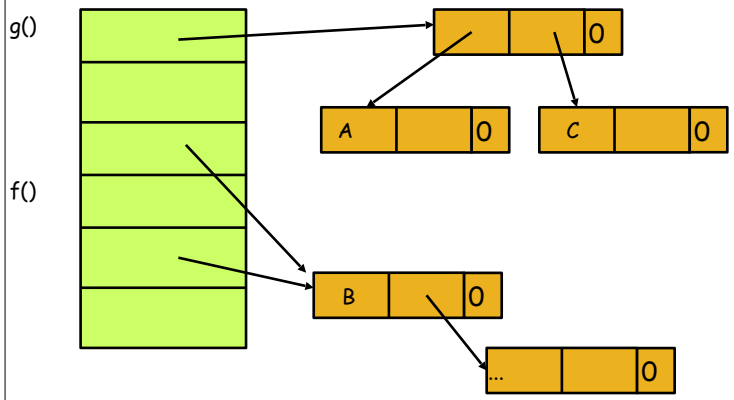
1. Mark reachable cells



2. Free ("sweep") unreachable cells



3. Clear tags



The Halting Problem: Proof

Suppose:

`Halt(P,x)` = { returns `true` if `P(x)` halts; returns `false` otherwise } — `Halt` always halts!

Construct:

`DNH(P)` = { if `Halt(P,P)` is `true`, `while(1){}`; returns `false` otherwise }

`DNH` does not always halt!

The Halting Problem

Isn't `DNH` itself a program?

What happens if we call `DNH(DNH)`?

`P = DNH`

`DNH( P )` will run forever if `P ( P )` halts.
`DNH( P )` will halt if `P ( P )` runs forever.

The Halting Problem

Isn't `DNH` itself a program?

What happens if we call `DNH(DNH)`?

`P = DNH`

`DNH(DNH)` will run forever if `DNH(DNH)` halts.
`DNH(DNH)` will halt if `DNH(DNH)` runs forever.

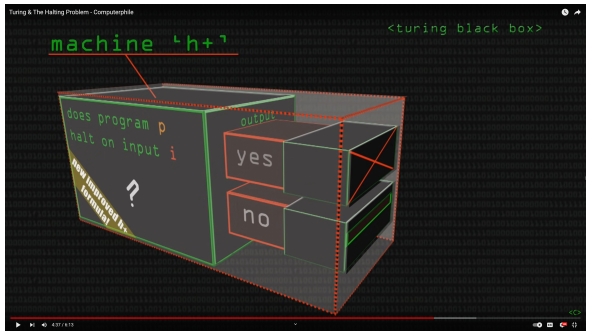
This literally makes no sense. **Contradiction!**

What was our one assumption? `Halt` **exists**.

Therefore, the `Halt` function **cannot exist**.

Need more explanation?

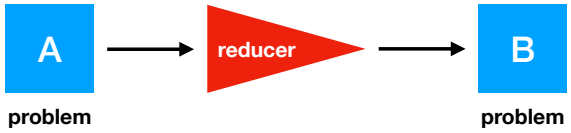
Watch this!



https://youtu.be/macM_MtS_w4

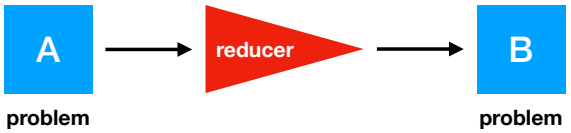
Reductions

A **reduction** is an **algorithm** that transforms an instance of one problem into an instance of another. Reductions are often **employed to prove something** about a problem given a similar problem.



Reductions

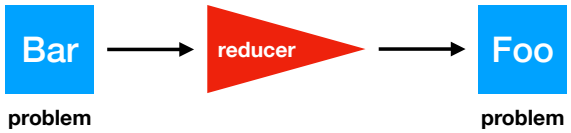
In this class, we will focus on proving things about **impossibility**, but reductions are much more general. In other cases, we prove things about **complexity**.



Reductions

Reductions are often used in a **counterintuitive** way.

For example, if we **want to know whether problem Foo is impossible**, we assume Foo is possible, and then use that fact to show that problem Bar (which **we already know** to be impossible) **appears to be possible**.

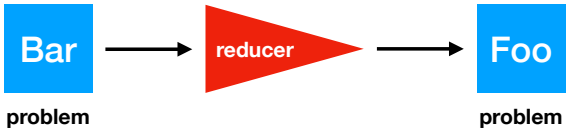


The above is a **contradiction**, meaning that **Foo is not possible**.

Reductions

Observe the **direction** of the reduction.

For computability proofs, we reduce the **problem we already know something about** to the **problem we want to understand**.

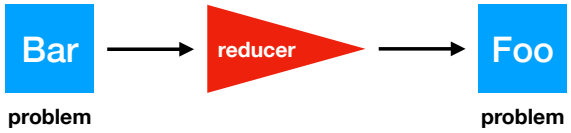


If the reduction is possible, it means that Foo is impossible **because we already know that Bar is impossible**.

Reductions

This direction **seems backward** to most people.

However, this counterintuitive direction is a feature common to many reductions.

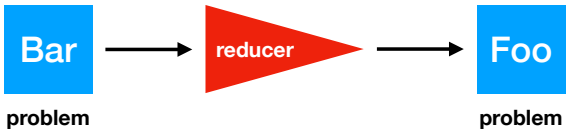


See the reading *Proof by Reduction* to understand the logic in more detail.

Reductions

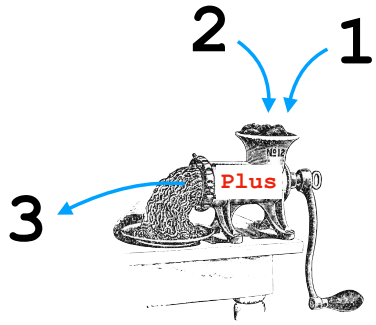
An important part of a reduction is that the reducer be an **ordinary algorithm**.

The reducer **should not solve the problem**. A reducer just converts problems from one form to another.



You will get **a lot** more exposure to reductions in CSCI 361.

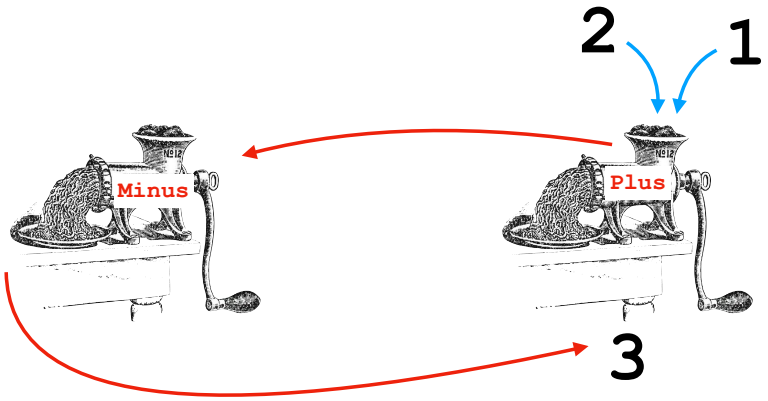
Reductions



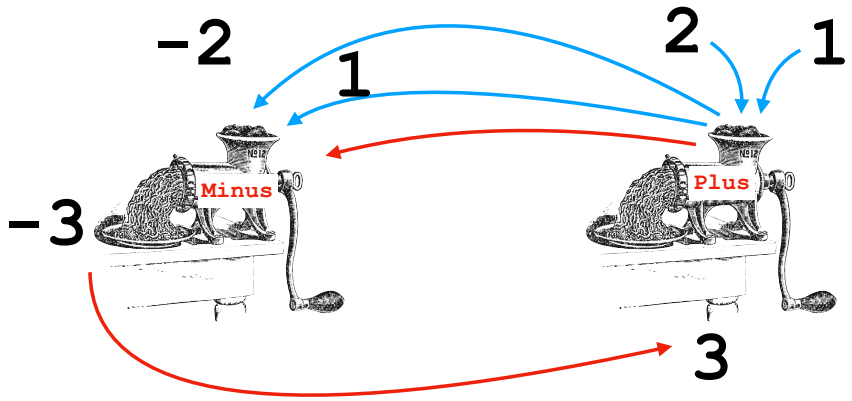
The humble algorithm.

(sorry, vegetarians)

Reductions

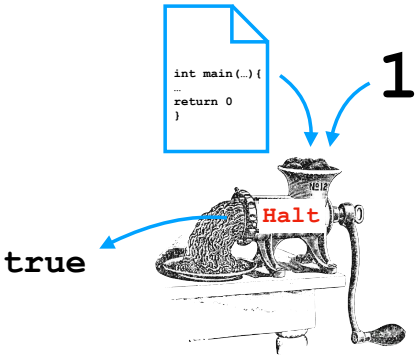


Reductions



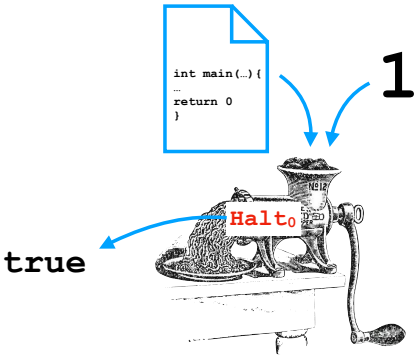

```
let reducer(x: int)(y: int) = 0-(0-x-y)
```

Reductions



We **know** that `Halt` is not computable.

Reductions



A function `f(i)` **halts not** if and only if `f` **does not halt** on input `i`.

Is $\texttt{Halt}_0$ computable?

Reductions

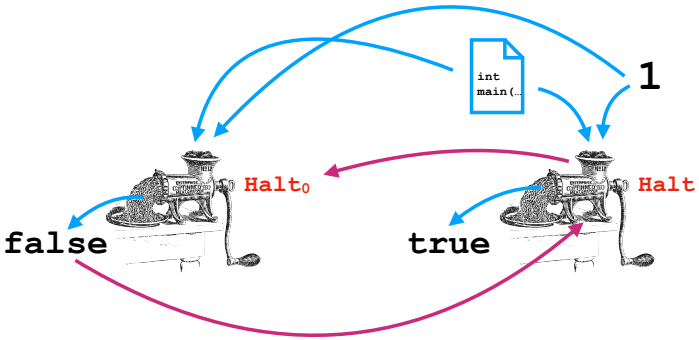
A function `f(i)` **halts not** if and only if `f` **does not halt** on input `i`.

If $\texttt{Halt}_0$ is computable, couldn't we do this?

Assume that $\texttt{Halt}_0$ is computable.
(e.g., it's in your standard library)

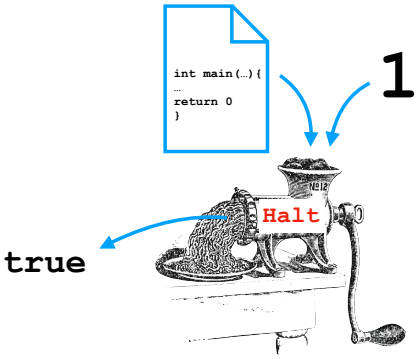
```
def halt(f, i):
  return not halt₀(f, i);
```

Reductions



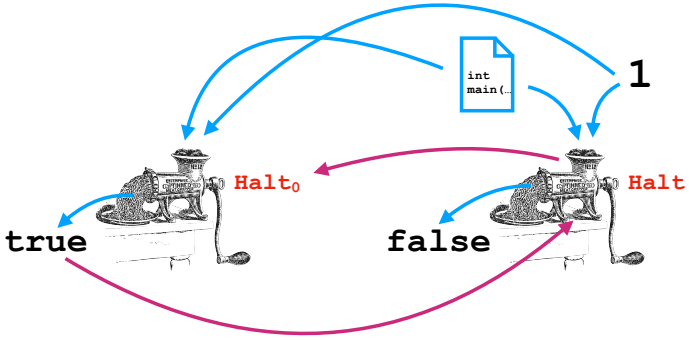
Reduction: **Construct** $\texttt{Halt}$ using $\texttt{Halt}_0$.

Reductions



We know that **Halt** is not computable.

Reductions



If we can build this new machine, what does that mean for **Halt₀**?

Halt₀ is **not computable**.

Activity

Prove that the Halting-No-Input problem is **undecidable**.

Problem: given a program P that requires no input, does P halt?

The Halting Problem

… helps us to understand the difficulty of many other problems.

Reductions

We can use the Halting Problem to show that other problems cannot be solved **by reduction** to the Halting Problem.

We cannot tell, in general…

- … if a program will **run forever**.
- … if a program will **eventually produce an error**.
- … if a program **is done using a variable**.
- … if a program **is a virus**!

Generality

```
def myprog(x):
  return 0

def Halt(f,i):
  if(f = "def myprog(x):\n\treturn 0"):
    return true
  else
    return false
```

The Halting Problem is about **an arbitrary program**.

Recap & Next Class

Today:

Halting problem

Reduction proofs

Next class:

Midterm review