CSCI 334: Principles of Programming Languages

Lecture 9: Computability, part 2

Instructor: Dan Barowy

Williams

Topics

Garbage collection

Halting problem

Reduction proofs

Your to-dos

1. Lab 5, **due Sunday 10/16** (partner lab)
2. Review quiz solutions if you haven't already…

Announcements

- **Field trip to WCMA**, Tuesday, Oct 18.
- **Midterm exam**, in class, Thursday, Nov 3.
- Colloquium: **What I Did Last Summer (Research)**, 2:35pm in Wege Auditorium with **cookies**.

Mountain Day, whenever that is…

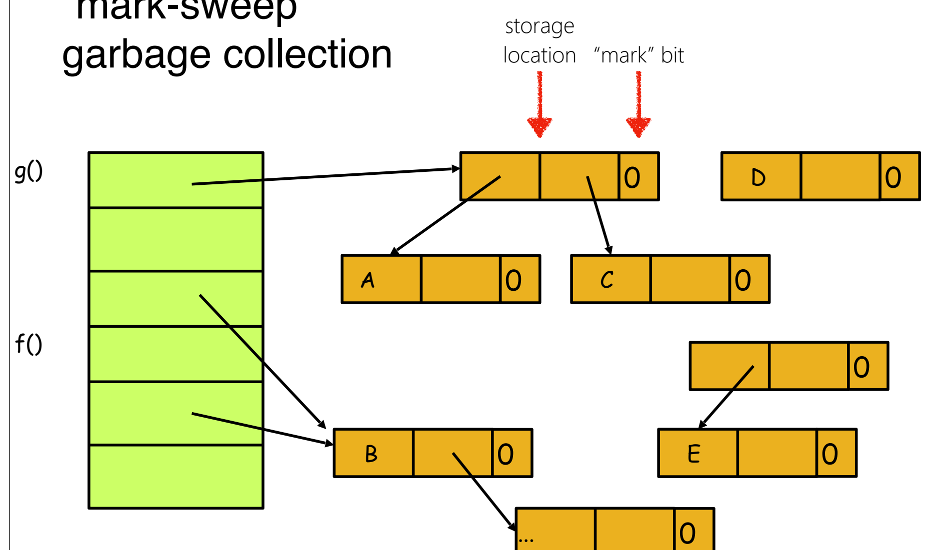
- Rescheduled office hours (faculty "retreat")

Garbage collection

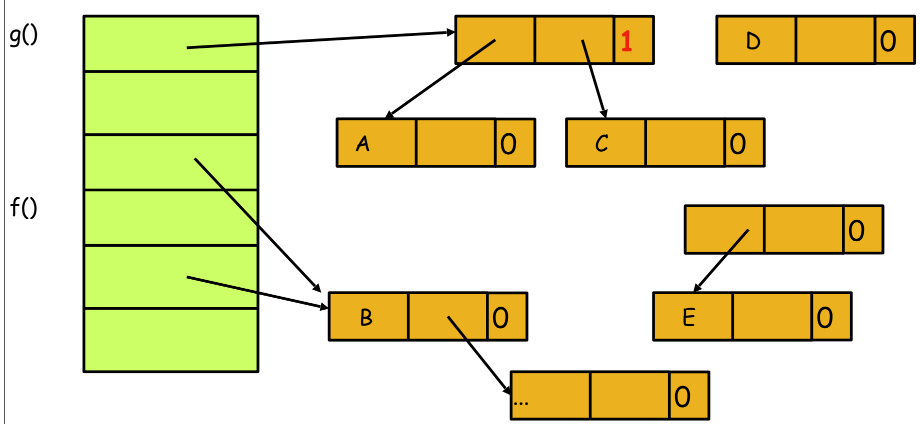
A **garbage collection algorithm** is an algorithm that determines whether the storage, occupied by a value used in a program, **can be reclaimed for future use**. Garbage collection algorithms are often tightly integrated into a programming language **runtime**.

John McCarthy

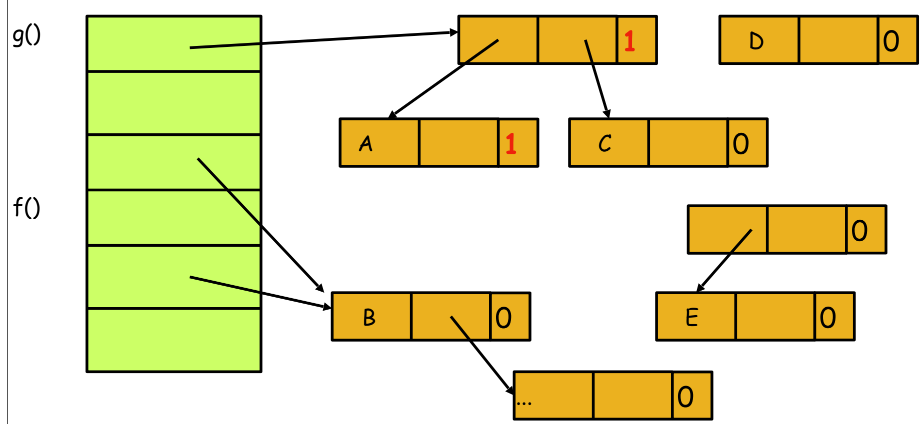
"mark-sweep" garbage collection



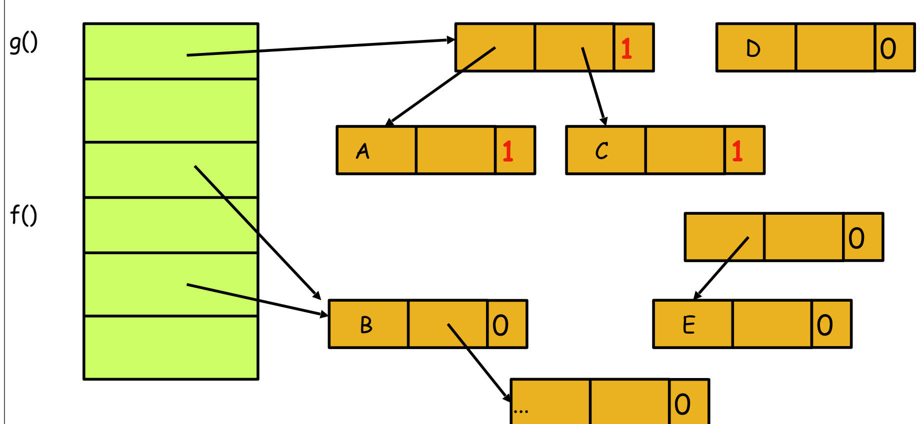
1. Mark reachable cells



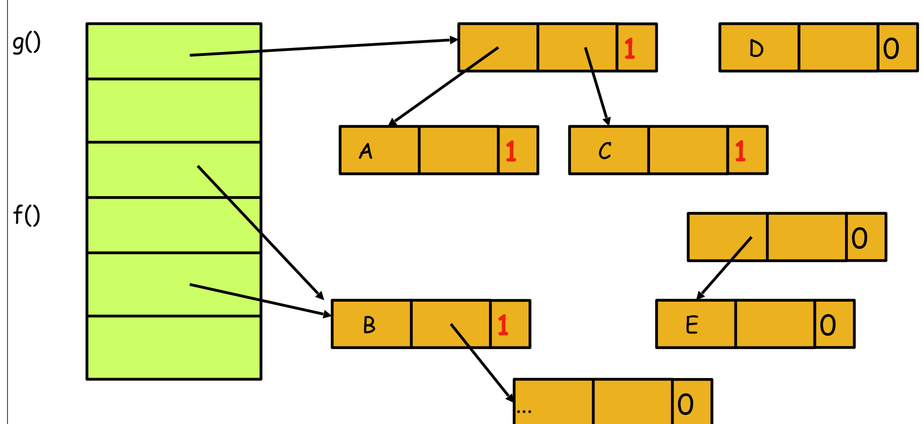
1. Mark reachable cells



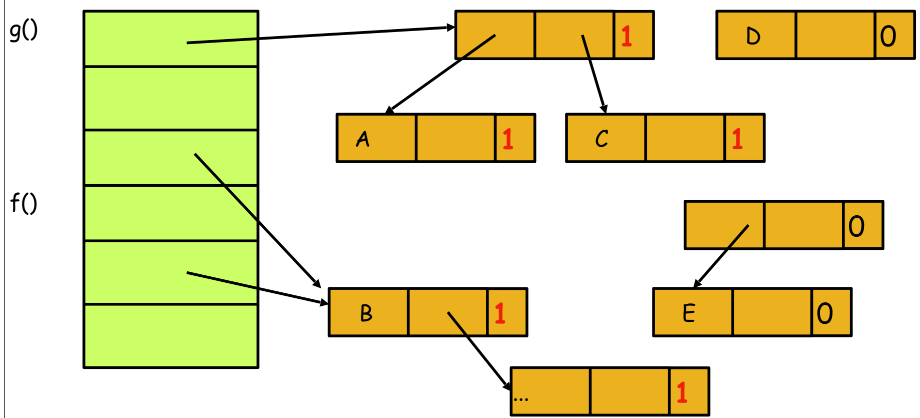
1. Mark reachable cells



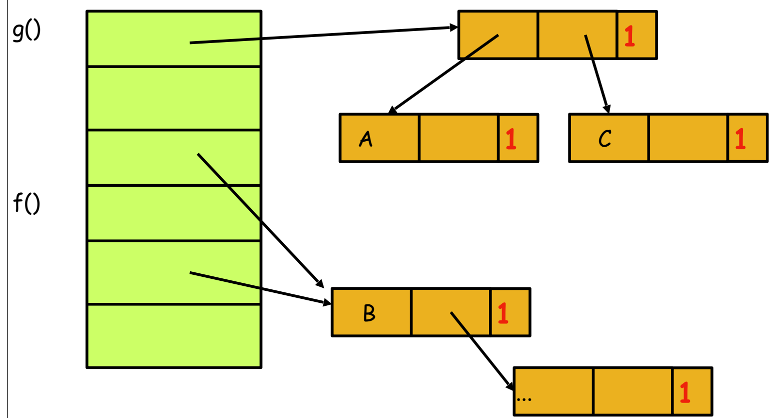
1. Mark reachable cells



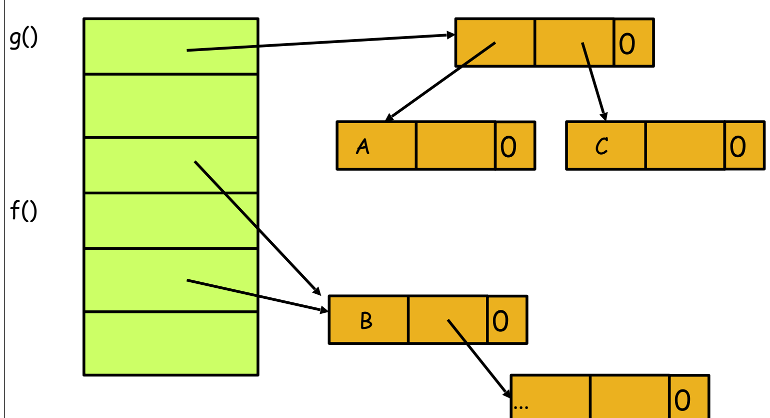
1. Mark reachable cells



2. Free ("sweep") unreachable cells



3. Clear tags



Decidability Problems

A **decidability problem** is a question with a **yes** or **no** answer about a **particular input**.

"Is x prime?"

In CS, we care about whether there is an **algorithm** for solving decidability problems.

If there is **no algorithm**, then the problem is **undecidable**.

Decidability Problems

A **decidability problem** is a question with a **yes** or **no** answer about a **particular input**.

"Is x prime?"

In CS, we care about whether there is an **algorithm** for solving decidability problems.

If there is **no algorithm**, then the problem is **undecidable**.

The Halting Problem

Decide whether program **P** halts on input **x**.

Given program P and input x,

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

How might this work?

Clarifications:

`P(x)` is the output of program `P` run on input `x`.

The type of `x` does not matter; assume `string`.

The Halting Problem

Decide whether program P halts on input x.

Given program P and input x,

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

How might this work?

Fact: it is provably impossible to write `Halt`

Notes on the proof

We utilize two key ideas:

- Function **evaluation by substitution**
- **Reductio ad absurdum** (proof form)

Notes on the proof

The *form* of the proof is ***reductio ad absurdum***.

Literally: "reduction to absurdity".

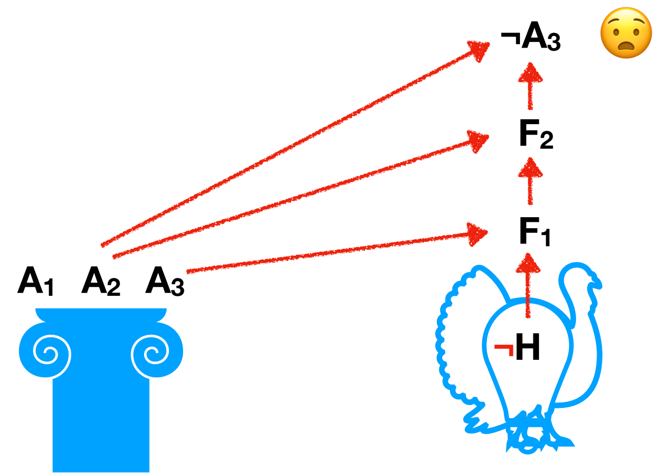
Start with **axioms** and **presuppose the outcome** we want to show.

Then, following strict rules of logic, **derive new facts**.

Finally, derive a fact that **contradicts** another fact.

Conclusion: the **presupposition must be false**.

Reductio ad Absurdum



Function Evaluation by Substitution

```
def addone(x):
   return x + 1
```

`addone(1)`	`λx.(+ x 1)1`
`[1/x]x + 1`	`[1/x](+ x 1)`
`1 + 1`	`(+ 1 1)`
`2`	`2`

The Halting Problem

Notes on the proof:

The proof relies on the kind of **substitution** that we've been using to "compute" functions in the lambda calculus.

Remember: **we are looking to produce a contradiction**.

The proof is hard to "understand" because the facts it derives **don't actually make sense**. Don't read too deeply.

The Halting Problem: Proof

Suppose:

`Halt(P,x)` =
- returns `true` if `P(x)` halts
- returns `false` otherwise

`Halt` always halts!

Construct:

`DNH` does not always halt!

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

The Halting Problem: Proof

Observations so far:

`DNH(P)` will run forever if `Halt(P,P)` is `true`.
`DNH(P)` will halt if `Halt(P,P)` is `false`.

Rewrite:

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

The Halting Problem: Proof

Observations so far:

`DNH(P)` will run forever if `Halt(P,P)` is `true`.
`DNH(P)` will halt if `Halt(P,P)` is `false`.

Rewrite:

`DNH(P)` =
- if `P(P)` halts, run forever
- returns `false` otherwise

The Halting Problem: Proof

Observations so far:

`DNH(P)` will run forever if `Halt(P,P)` is `true`.
`DNH(P)` will halt if `Halt(P,P)` is `false`.

Rewrite:

`DNH(P)` =
- if `P(P)` halts, run forever
- halt

The Halting Problem: Proof

Observations so far:

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

Rewrite:

`DNH(P)` = { if `P(P)` halts, run forever; 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.

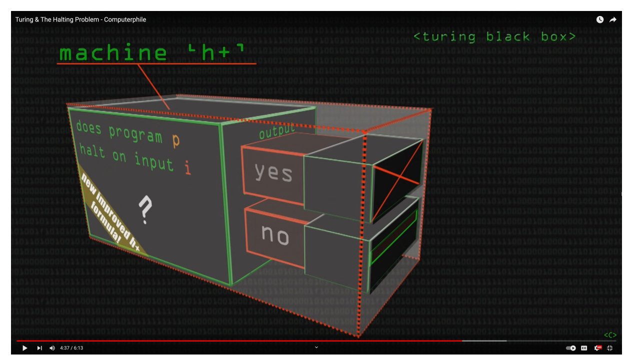
This literally makes no sense. **Contradiction!**

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

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

Need more explanation?

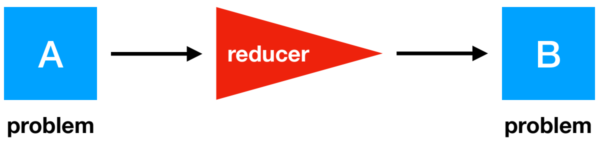
Watch this!



https://youtu.be/macM_MtS_w4

Reductions

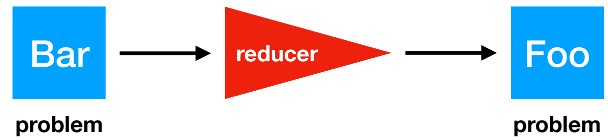
A **reduction** is an **algorithm** that transforms an instance of one problem into an instance of another. Reductions are often **employed to prove something** about a problem given a similar problem.



Reductions

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**.

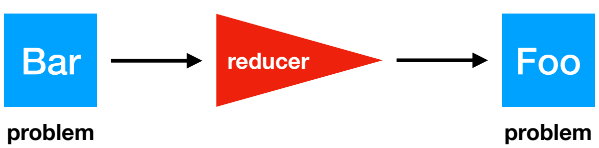


The above is a **contradiction**, meaning that **Foo is not possible**.

Reductions

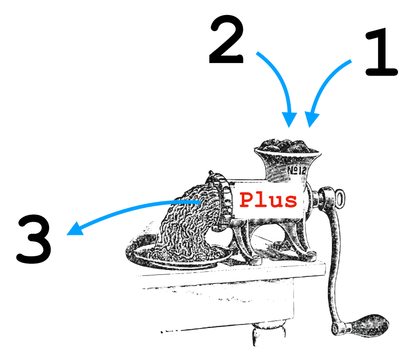
An important part of a reduction is that the reducer be an **ordinary algorithm**.

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.

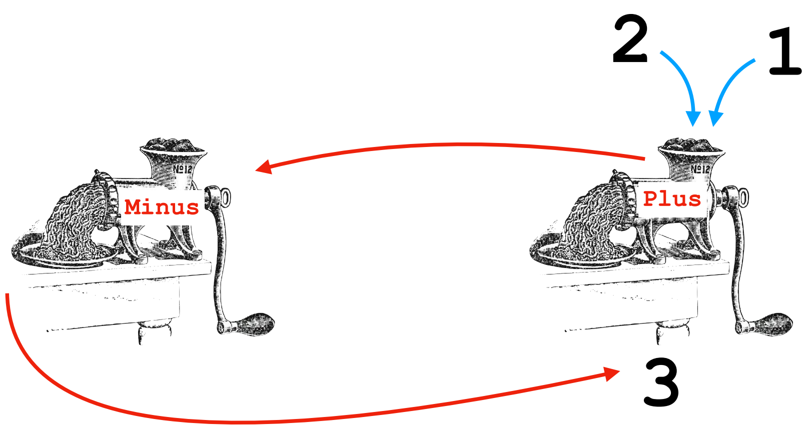
Reductions



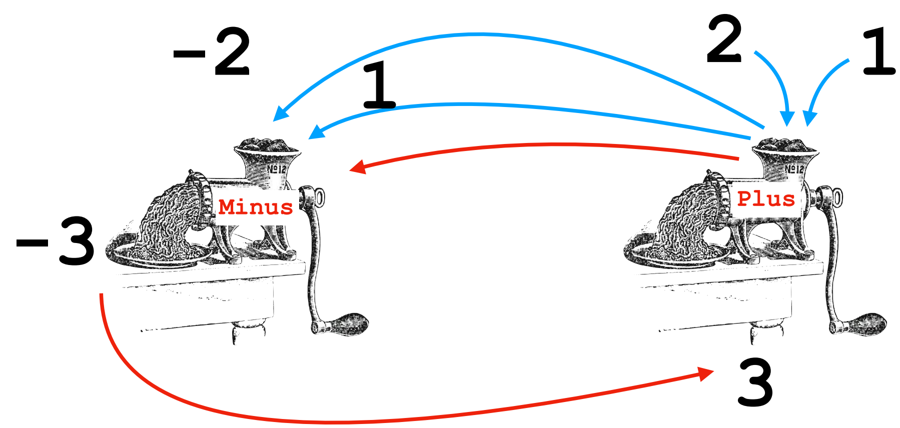
The humble algorithm.

(sorry, vegetarians)

Reductions

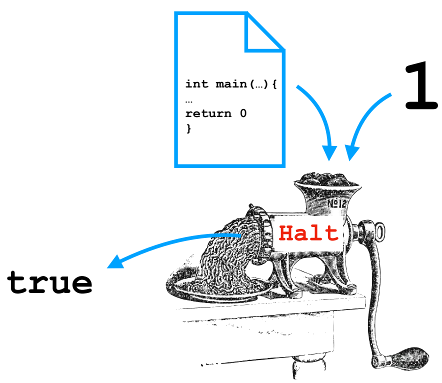


Reductions



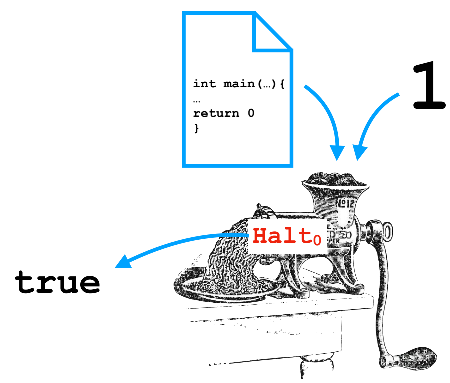

```
let reducer(x: int)(y: int) = -(-x-y)
```

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

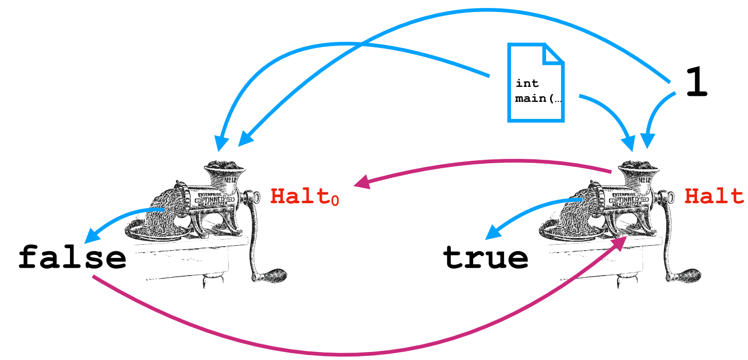
A function `f(i)` **halts not** if and only if `f` **does not halt** on input `i`.

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)

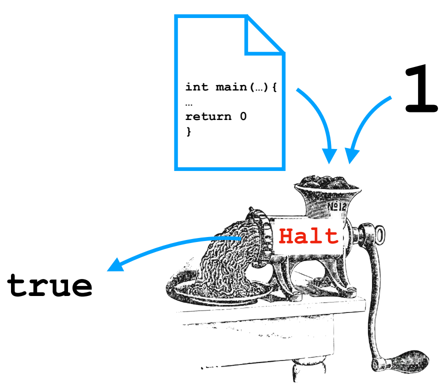
```
def halt(f, i):
  return not halt_0(f, i);
```

Reductions



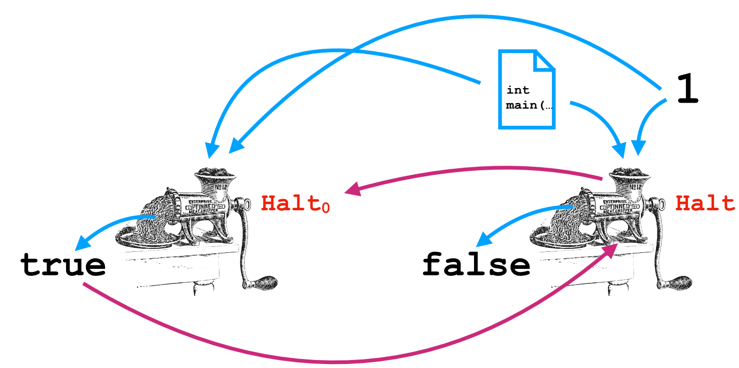
Reduction: **Construct** `Halt` using $\texttt{Halt}_0$.

Reductions



We know that `Halt` is not computable.

Reductions



If we can build this new machine,
what does that mean for $\texttt{Halt}_0$?

$\texttt{Halt}_0$ is **not computable**.

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:

WCMA