CSCI 334:
Principles of Programming Languages

Lecture 8: PL Fundamentals IV

Instructor: Dan Barowy

Williams

Announcements

Announcements

No class on Tuesday (reading period)

Announcements

No class on Tuesday (reading period)

Midterm exam on Thursday

Announcements

No class on Tuesday (reading period)

Midterm exam on Thursday

Midterm review Q&A:
Here, Tuesday, normal time
(optional!)

Announcements

No class on Tuesday (reading period)

Midterm exam on Thursday

Midterm review Q&A:
Here, Tuesday, normal time
(optional!)

Will post study guide shortly after class

Another way to find redexes

```
(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2
```

Another way to find redexes

```
(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2
```

Normal order: `(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`

Another way to find redexes

`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`

Normal order: `(λa.(λz.(+ x z))((λz.(+ x z)) a ))` `2`

Applicative order: `(λa.(λz.(+ x z))(` `(λz.(+ x z))` `a` `)) 2`

Another way to find redexes

`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`

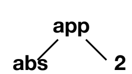
Normal order: `(λa.(λz.(+ x z))((λz.(+ x z)) a ))` `2`

Applicative order: `(λa.(λz.(+ x z))(` `(λz.(+ x z))` `a` `)) 2`

Neither: `(λa.` `(λz.(+ x z))` `((λz.(+ x z)) a )` `) 2`

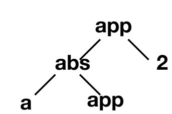
Another way to find redexes

`(λa.(λz.(+ x z))((λz.(+ x z)) a ))` `2`



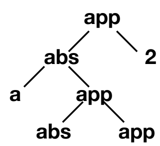
Another way to find redexes

`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`



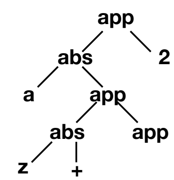
Another way to find redexes

`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`



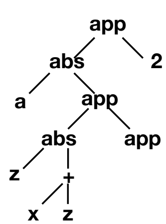
Another way to find redexes

`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`



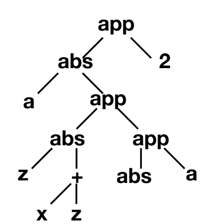
Another way to find redexes

`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`



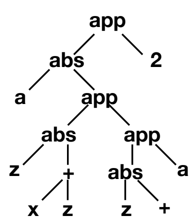
Another way to find redexes

`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`



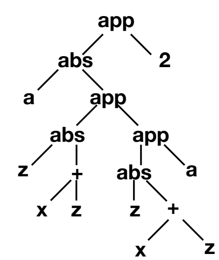
Another way to find redexes

(λa.(λz.(+ x z))((λz.(+ x z)) a)) 2



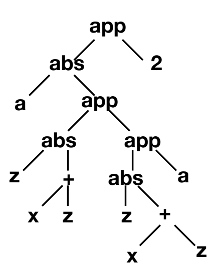
Another way to find redexes

(λa.(λz.(+ x z))((λz.(+ x z)) a)) 2



Another way to find redexes

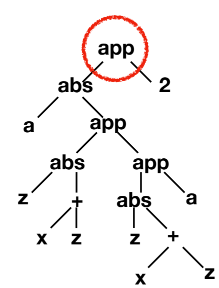
(λa.(λz.(+ x z))((λz.(+ x z)) a)) 2



Normal order: “leftmost outermost” application

Another way to find redexes

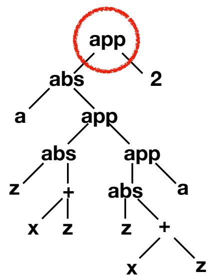
(λa.(λz.(+ x z))((λz.(+ x z)) a)) 2



Normal order: “leftmost outermost” application

Another way to find redexes

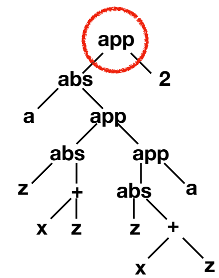
`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`



Normal order: "leftmost outermost" application

Another way to find redexes

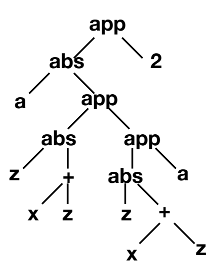
`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`



Normal order: "leftmost outermost" application

Another way to find redexes

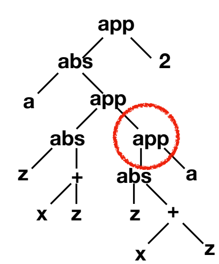
`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`



Applicative order: "leftmost innermost" application

Another way to find redexes

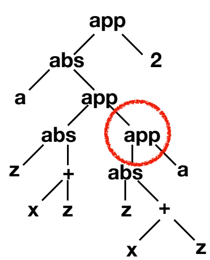
`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`



Applicative order: "leftmost innermost" application

Another way to find redexes

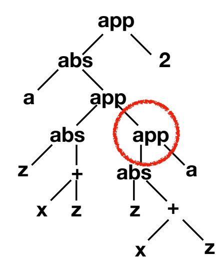
`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`



Applicative order: “leftmost innermost” application

Another way to find redexes

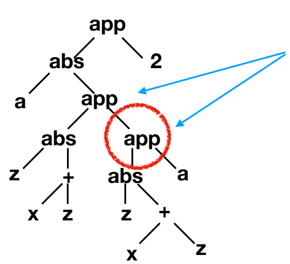
`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`



Applicative order: “leftmost innermost” application

Another way to find redexes

`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`

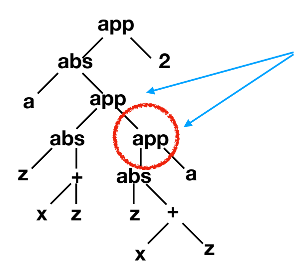


Q: Wait!
Isn’t there a
“more left” app?

Applicative order: “leftmost innermost” application

Another way to find redexes

`(λa.(λz.(+ x z))((λz.(+ x z)) a )) 2`



Q: Wait!
Isn’t there a
“more left” app?

A: Yes, but it is not the *innermost*. “Leftmost” is used to break ties among multiple innermost apps.

Applicative order: “leftmost innermost” application

The Halting Problem

The Halting Problem

The Halting Problem

Decide whether program P halts on input x.

The Halting Problem

Decide whether program P halts on input x.

Given program P and input x,

$$\texttt{Halt(P,x)} = \begin{cases} \text{returns } \texttt{true} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

The Halting Problem

Decide whether program P halts on input x.

Given program P and input x,

$$\texttt{Halt(P,x)} = \begin{cases} \text{returns } \texttt{true} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

<u>Clarifications:</u>

The Halting Problem

Decide whether program P halts on input x.

Given program P and input x,

$$\texttt{Halt(P,x)} = \begin{cases} \text{returns } \texttt{true} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

<u>Clarifications:</u>

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

The Halting Problem

Decide whether program P halts on input x.

Given program P and input x,

$$\texttt{Halt(P,x)} = \begin{cases} \text{returns } \texttt{true} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

<u>Clarifications:</u>

`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,

$$\texttt{Halt(P,x)} = \begin{cases} \text{returns } \texttt{true} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

How might this work?

<u>Clarifications:</u>

`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,

$$\texttt{Halt(P,x)} = \begin{cases} \text{returns } \texttt{true} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

How might this work?

The Halting Problem

Decide whether program P halts on input x.

Given program P and input x,

$$\texttt{Halt(P,x)} = \begin{cases} \text{returns } \texttt{true} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

How might this work?

Fact: it is provably impossible to write `Halt`

Notes on the proof

Notes on the proof

We utilize two key ideas:

Notes on the proof

We utilize two key ideas:

- Function evaluation by substitution

Notes on the proof

We utilize two key ideas:

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

Function Evaluation by Substitution

Function Evaluation by Substitution

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

Function Evaluation by Substitution

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

```
addone(1)
```

Function Evaluation by Substitution

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

```
addone(1)

[1/x]x + 1
```

Function Evaluation by Substitution

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

```
addone(1)

[1/x]x + 1

1 + 1
```

Function Evaluation by Substitution

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

```
addone(1)

[1/x]x + 1

1 + 1

2
```

Function Evaluation by Substitution

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

```
addone(1)         λx.(+ x 1)1

[1/x]x + 1

1 + 1

2
```

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

2
```

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
```

Function Evaluation by Substitution

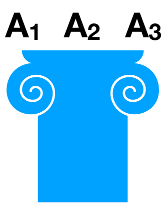
```
def addone(x):
   return x + 1
```

```
addone(1)         λx.(+ x 1)1

[1/x]x + 1        [1/x](+ x 1)

1 + 1             (+ 1 1)

2                 2
```

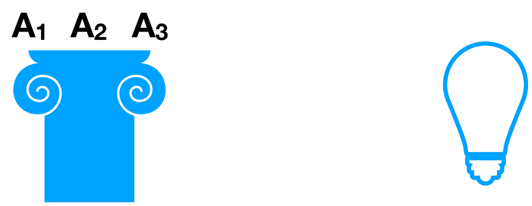
Reductio ad Absurdum

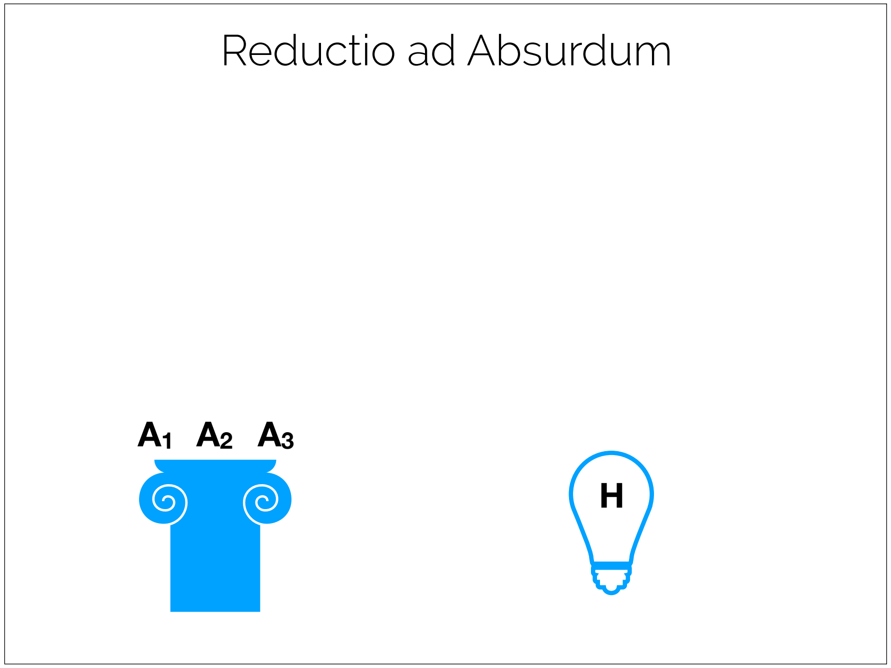
Reductio ad Absurdum

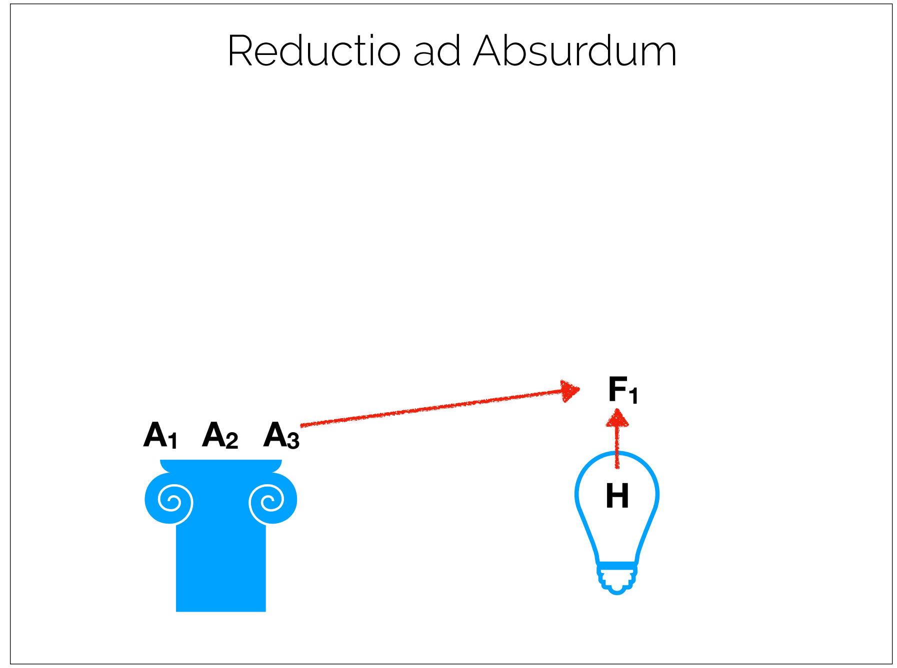
Reductio ad Absurdum

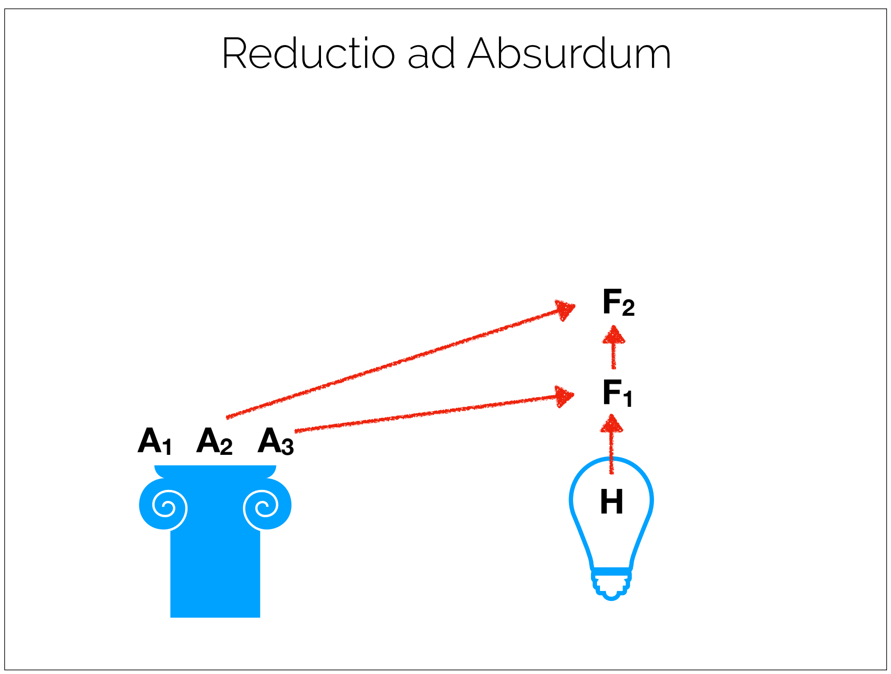


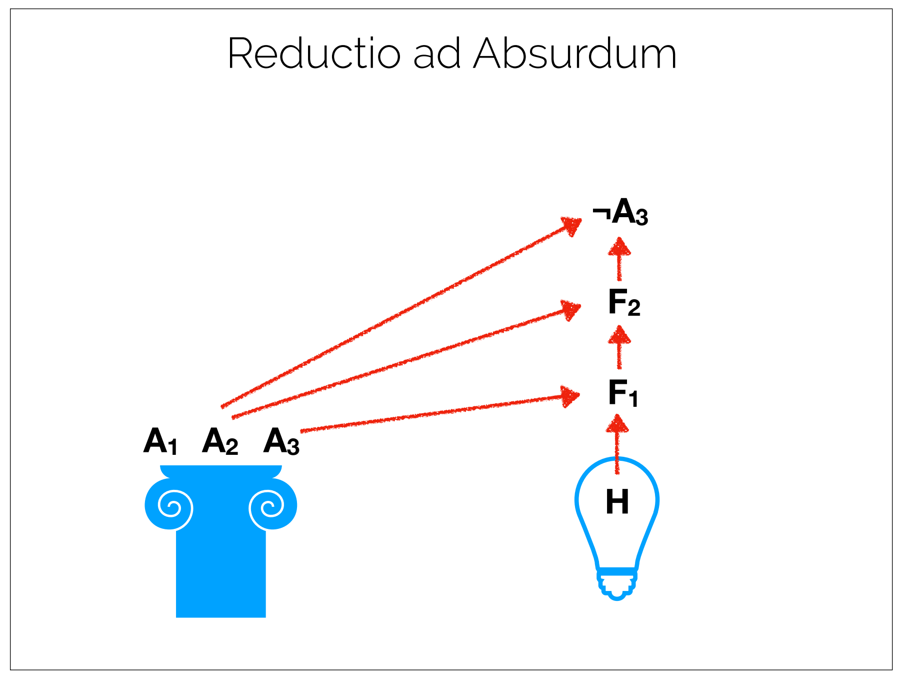
Reductio ad Absurdum



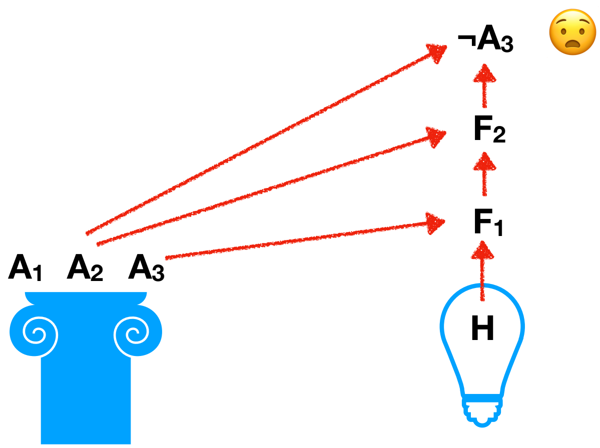

Reductio ad Absurdum
A_1 A_2 A_3
H


Reductio ad Absurdum
F_1
A_1 A_2 A_3
H

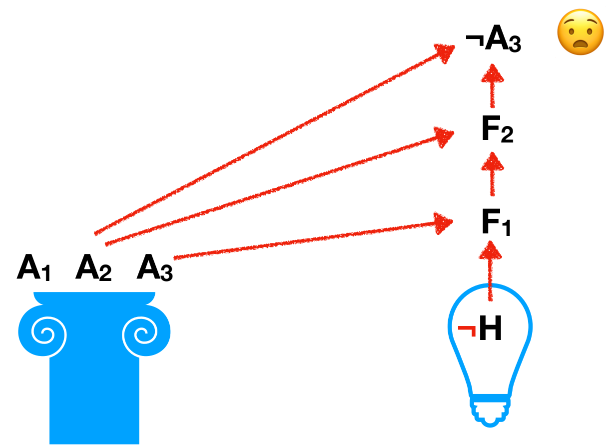

Reductio ad Absurdum
F_2
F_1
A_1 A_2 A_3
H


Reductio ad Absurdum
$\neg A_3$
F_2
F_1
A_1 A_2 A_3
H

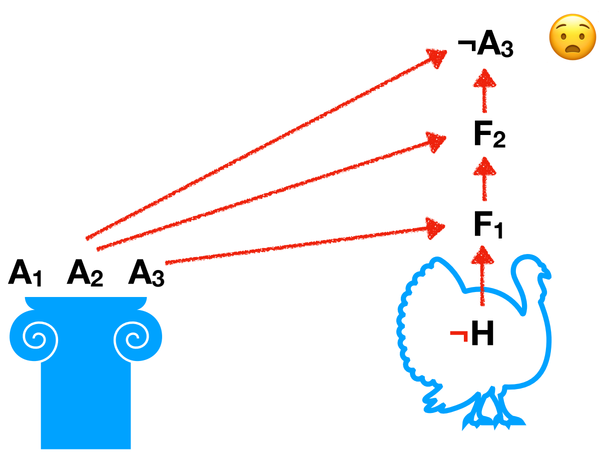
Reductio ad Absurdum



Reductio ad Absurdum



Reductio ad Absurdum



The Halting Problem: Proof

The Halting Problem: Proof

Suppose:

$$\texttt{Halt(P,x)} = \begin{cases} \text{returns } \texttt{true} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

The Halting Problem: Proof

Suppose:

$$\texttt{Halt(P,x)} = \begin{cases} \text{returns } \texttt{true} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases} \Bigg\} \begin{array}{c}\texttt{Halt} \\ \text{always} \\ \text{halts!}\end{array}$$

The Halting Problem: Proof

Suppose:

$$\texttt{Halt(P,x)} = \begin{cases} \text{returns } \texttt{true} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases} \Bigg\} \begin{array}{c}\texttt{Halt} \\ \text{always} \\ \text{halts!}\end{array}$$

Construct:

$$\texttt{DNH(P)} = \begin{cases} \text{if } \texttt{Halt(P,P)} \text{ is } \texttt{true}, \texttt{ while(1)\{\}} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

The Halting Problem: Proof

Suppose:

$$\texttt{Halt(P,x)} = \begin{cases} \text{returns } \texttt{true} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases} \Bigg\} \begin{array}{c}\texttt{Halt} \\ \text{always} \\ \text{halts!}\end{array}$$

Construct:

$$\begin{array}{c}\texttt{DNH} \\ \text{does not} \\ \text{always halt!}\end{array} \Bigg\{ \texttt{DNH(P)} = \begin{cases} \text{if } \texttt{Halt(P,P)} \text{ is } \texttt{true}, \texttt{ while(1)\{\}} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

The Halting Problem: Proof

The Halting Problem: Proof

Observations so far:

The Halting Problem: Proof

Observations so far:

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

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`.

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:

$$\texttt{DNH(P)} = \begin{cases} \text{if } \texttt{Halt(P,P)} \text{ is } \texttt{true}\text{, } \texttt{while(1)\{\}} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

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:

$$\texttt{DNH(P)} = \begin{cases} \text{if } \texttt{P(P)} \text{ halts, run forever} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

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:

$$\texttt{DNH(P)} = \begin{cases} \text{if } \texttt{P(P)} \text{ halts, run forever} \\ \text{halt} \end{cases}$$

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:

$$\texttt{DNH(P)} = \begin{cases} \text{if } \texttt{P(P)} \text{ halts, run forever} \\ \text{halt} \end{cases}$$

The Halting Problem

`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?

`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)`?

`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` (`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.

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!

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.

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.

Generality

Generality

```
def myprog(x):
  return 0
```

Generality

```
def myprog(x):
  return 0
```

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

Reductions

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

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…

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.

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.

Reductions

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

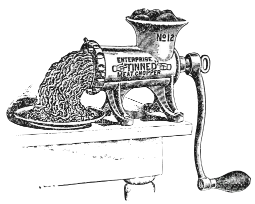
We cannot tell, in general…

… if a program will run forever.

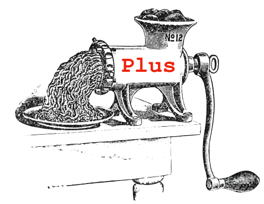
… if a program will eventually produce an error.

… if a program is done using a variable.

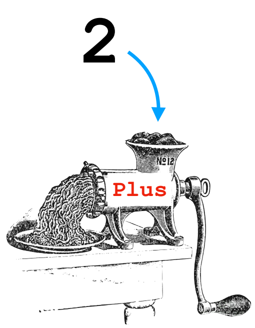
Reductions



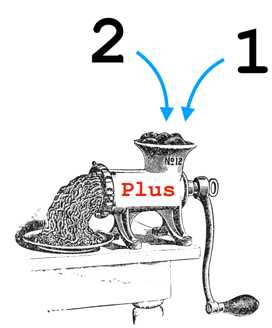
Reductions



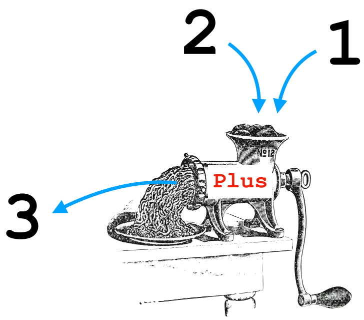
Reductions



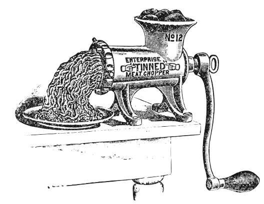
Reductions



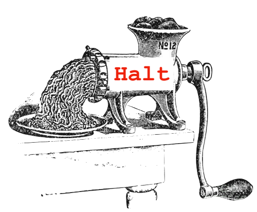
Reductions



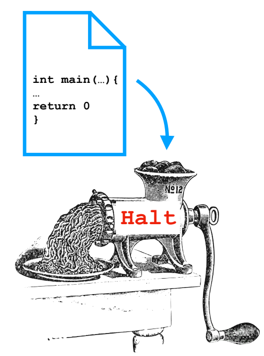
Reductions



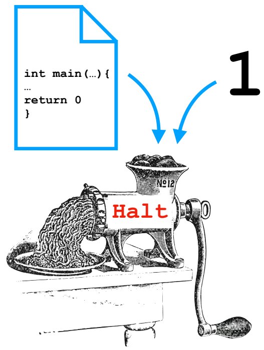
Reductions



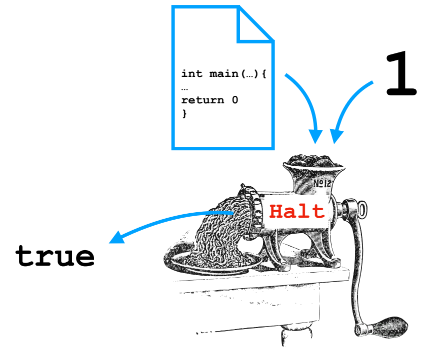
Reductions



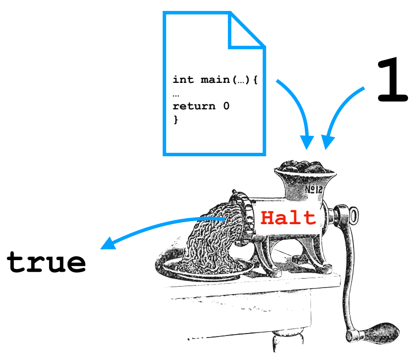
Reductions



Reductions

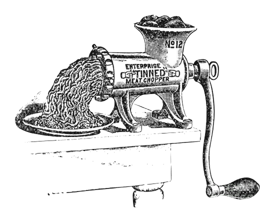


Reductions

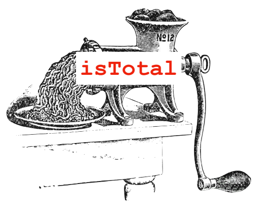


We know that `Halt` is impossible to compute.

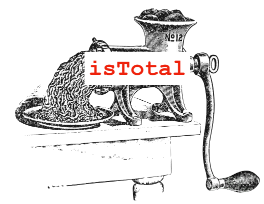
Reductions



Reductions

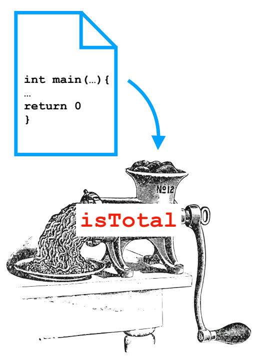


Reductions



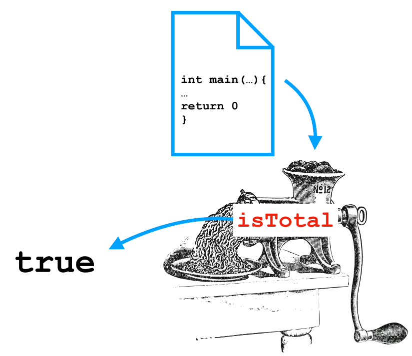
A function `f(i)` is total if and only if `f` is defined for every input `i`.

Reductions



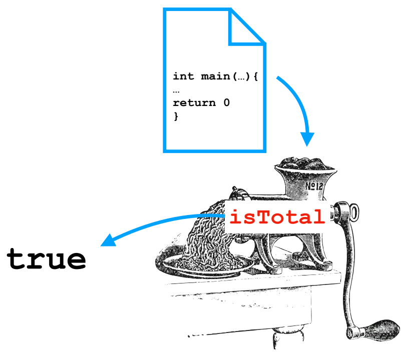
A function `f(i)` is total if and only if `f` is defined for every input `i`.

Reductions



A function `f(i)` is total if and only if `f` is defined for every input `i`.

Reductions



A function `f(i)` is total if and only if `f` is defined for every input `i`.

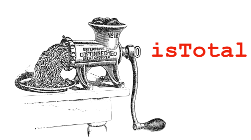
Is `isTotal` computable?

Reductions

Reductions

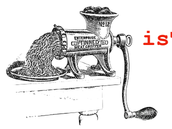
Assume that `isTotal` is computable.

Reductions



Assume that `isTotal` is computable.

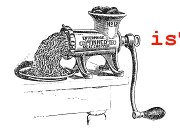
Reductions



Assume that `isTotal` is computable.

(e.g., it's in your standard library)

Reductions

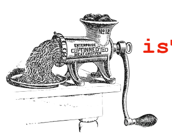


Assume that `isTotal` is computable.

(e.g., it's in your standard library)

Construct `Halt` using `isTotal`.

Reductions

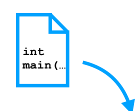


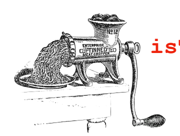
Assume that `isTotal` is computable.

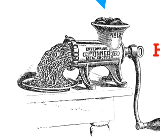
(e.g., it's in your standard library)

Construct `Halt` using `isTotal`.

Reductions







Assume that `isTotal` is computable.

(e.g., it's in your standard library)

Construct `Halt` using `isTotal`.

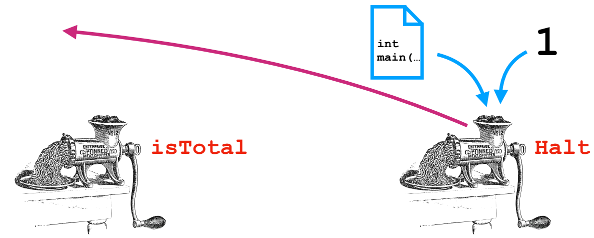
Reductions

Assume that `isTotal` is computable.

(e.g., it's in your standard library)

Construct `Halt` using `isTotal`.

Reductions

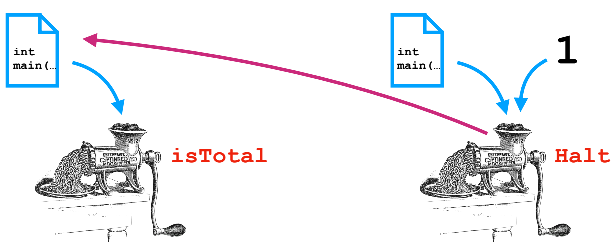


Assume that `isTotal` is computable.

(e.g., it's in your standard library)

Construct `Halt` using `isTotal`.

Reductions

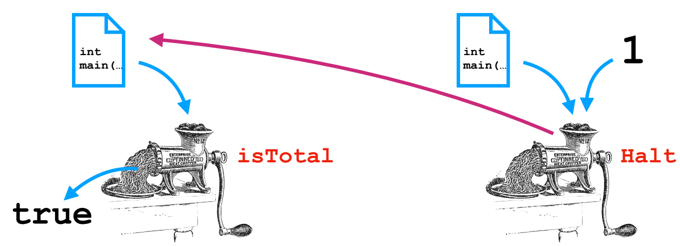


Assume that `isTotal` is computable.

(e.g., it's in your standard library)

Construct `Halt` using `isTotal`.

Reductions

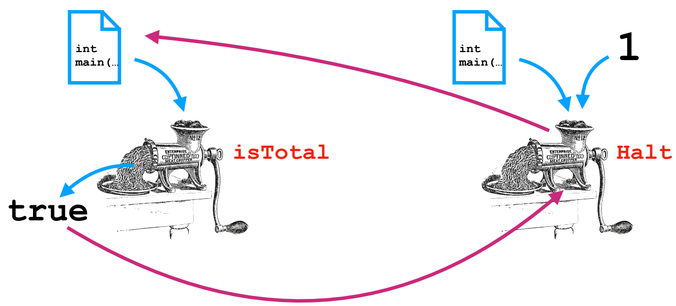


Assume that `isTotal` is computable.

(e.g., it's in your standard library)

Construct `Halt` using `isTotal`.

Reductions

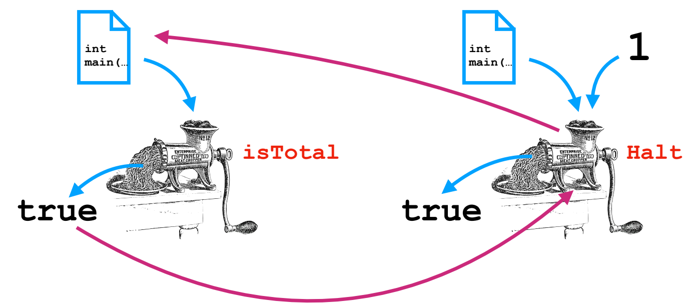


Assume that `isTotal` is computable.

(e.g., it's in your standard library)

Construct `Halt` using `isTotal`.

Reductions

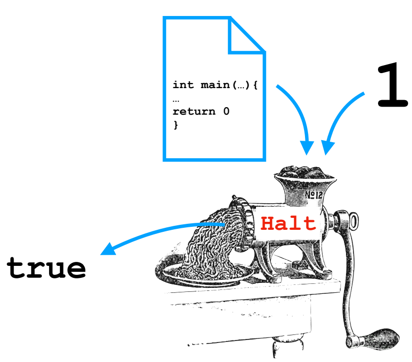


Assume that `isTotal` is computable.

(e.g., it's in your standard library)

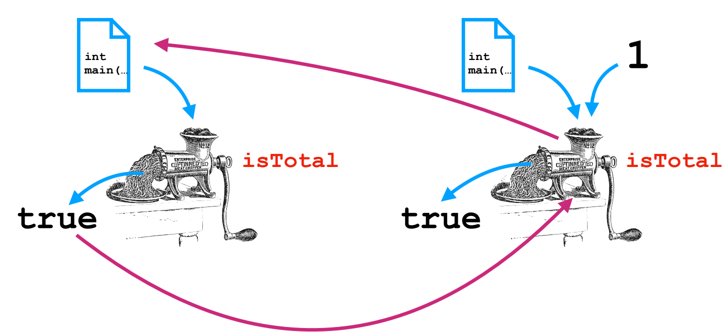
Construct `Halt` using `isTotal`.

Reductions



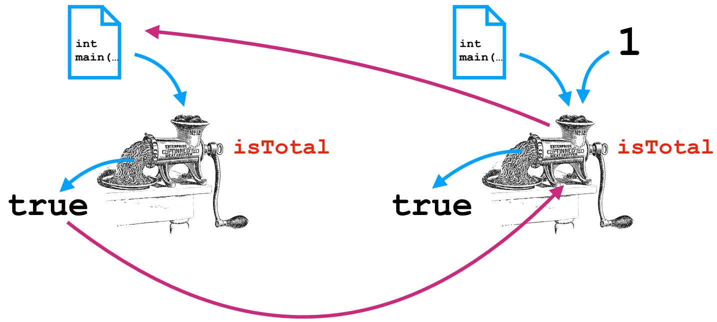
We know that `Halt` is impossible to compute.

Reductions



If we can do it, what does this mean for `isTotal`?

Reductions



If we can do it, what does this mean for `isTotal`?

`isTotal` is not computable.

Reductions

Reductions

```
def Halt(p,i):

    return Total(                    )
```

Reductions

```
def Halt(p,i):

    return Total(        p          )
```

Reductions

```
def Halt(p,i):

    return Total(      p       )
```

Reductions

```
def Halt(p,i):

    return Total(      p       )
```

Why doesn't this version do what we want?

Reductions

```
def Halt(p,i):

    return Total(      p       )
```

Why doesn't this version do what we want?

Because `Total` tells us whether `p` halts on all inputs.

Reductions

```
def Halt(p,i):

    return Total(      p       )
```

Why doesn't this version do what we want?

Because `Total` tells us whether `p` halts on all inputs.

`Halt` asks specifically about input `i`.

Reductions

```
def Halt(p,i):


    return Total(                  )
```

Reductions

```
def Halt(p,i):
    def Runs(j):
        p(i)
        return "done!"
    return Total(                  )
```

Reductions

```
def Halt(p,i):
    def Runs(j):
        p(i)
        return "done!"
    return Total( getsource(Runs) )
```

Reductions

```
def Halt(p,i):
    def Runs(j):
        p(i)
        return "done!"
    return Total( getsource(Runs) )
```

This is a valid program.

Reductions

```
def Halt(p,i):
    def Runs(j):
        p(i)
        return "done!"
    return Total( getsource(Runs) )
```

This is a valid program.

So we can solve the Halting Problem!

Reductions

```
def Halt(p,i):
    def Runs(j):
        p(i)
        return "done!"
    return Total( getsource(Runs) )
```

This is a valid program.

So we can solve the Halting Problem!

Therefore, `Total` is not computable. 😞

Mini-Midterm Review

What have we learned?

Mini-Midterm Review

What have we learned?

The C programming language

Mini-Midterm Review

What have we learned?

The C programming language

- Basic syntax

Mini-Midterm Review

What have we learned?

The C programming language

- Basic syntax
- Program evaluation using a call stack

Mini-Midterm Review

What have we learned?

The C programming language

- Basic syntax
- Program evaluation using a call stack
- Memory as an explicitly-managed resource

Mini-Midterm Review

What have we learned?

The C programming language

- Basic syntax
- Program evaluation using a call stack
- Memory as an explicitly-managed resource

Grammars and parsing

Mini-Midterm Review

What have we learned?

The C programming language
- Basic syntax
- Program evaluation using a call stack
- Memory as an explicitly-managed resource

Grammars and parsing
- Backus-Naur form

Mini-Midterm Review

What have we learned?

The C programming language
- Basic syntax
- Program evaluation using a call stack
- Memory as an explicitly-managed resource

Grammars and parsing
- Backus-Naur form
- Prec. and assoc. rules for ambiguous grammars

Mini-Midterm Review

What have we learned?

The C programming language
- Basic syntax
- Program evaluation using a call stack
- Memory as an explicitly-managed resource

Grammars and parsing
- Backus-Naur form
- Prec. and assoc. rules for ambiguous grammars
- Derivation and abstract syntax trees

Mini-Midterm Review

What have we learned?

The C programming language
- Basic syntax
- Program evaluation using a call stack
- Memory as an explicitly-managed resource

Grammars and parsing
- Backus-Naur form
- Prec. and assoc. rules for ambiguous grammars
- Derivation and abstract syntax trees

The lambda calculus

Mini-Midterm Review

What have we learned?

The C programming language
- Basic syntax
- Program evaluation using a call stack
- Memory as an explicitly-managed resource

Grammars and parsing
- Backus-Naur form
- Prec. and assoc. rules for ambiguous grammars
- Derivation and abstract syntax trees

The lambda calculus
- Syntax

Mini-Midterm Review

What have we learned?

The C programming language
- Basic syntax
- Program evaluation using a call stack
- Memory as an explicitly-managed resource

Grammars and parsing
- Backus-Naur form
- Prec. and assoc. rules for ambiguous grammars
- Derivation and abstract syntax trees

The lambda calculus
- Syntax
- Program evaluating using reduction rules

Mini-Midterm Review

Not on the exam:

Mini-Midterm Review

Not on the exam:

- Halting problem
- Reductions