CSCI 334: Principles of Programming Languages

Lecture 7: PL Fundamentals III

Instructor: Dan Barowy

Williams

Announcements

Announcements

Resubmission procedure

Announcements

Resubmission procedure

Wednesday Office Hours now
3pm-5pm
(originally: 10am-noon)

Announcements

Resubmission procedure

Wednesday Office Hours now
3pm-5pm
(originally: 10am-noon)

(If these hours still don't work
for you, make an appointment)

Mental Technique #3

Mental Technique #3

Mental Technique #3

Confusion is not necessarily a bad thing.

Mental Technique #3

Mental Technique #3

Sometimes Confusion is a Good Thing
Tania Lombrozo
NPR, December 14, 2015

https://www.npr.org/sections/13.7/2015/12/14/459651340/sometimes-confusion-is-a-good-thing

Mental Technique #3

Sometimes Confusion is a Good Thing
Tania Lombrozo
NPR, December 14, 2015

"Students who were confused … as reflected in inconsistent responses on subsequent questions … ultimately did better on a final test assessing whether they learned the key points from the lessons."

https://www.npr.org/sections/13.7/2015/12/14/459651340/sometimes-confusion-is-a-good-thing

Mental Technique #3

Sometimes Confusion is a Good Thing
Tania Lombrozo
NPR, December 14, 2015

"One possibility is that confusion is … a marker that an important cognitive process has taken place: The learner has appreciated some inconsistency or deficit in her prior beliefs. … [A]nother possibility is that confusion is itself a step toward learning — an experience that motivates the learner to reconcile an inconsistency or remedy some deficit. In this view, confusion isn't just a side effect of beneficial cognitive processes, but a beneficial process itself. Supporting this stronger view, there's evidence that experiencing difficulties in learning can sometimes be desirable, leading to deeper processing and better long-term memory."

https://www.npr.org/sections/13.7/2015/12/14/459651340/sometimes-confusion-is-a-good-thing

Mental Technique #3

The importance of stupidity in scientific research
Martin A. Schwartz
Journal of Cell Science 2008 121: 1771 doi: 10.1242/jcs.033340

"Focusing on important questions puts us in the awkward position of being ignorant. One of the beautiful things about science is that it allows us to bumble along, getting it wrong time after time, and feel perfectly fine as long as we learn something each time. No doubt, this can be difficult for students who are accustomed to getting the answers right."

Mental Technique #3

Confusion is not necessarily a bad thing.

Mental Technique #3

Confusion is not necessarily a bad thing.

It is a signal that you are not confident in your knowledge.

Mental Technique #3

Confusion is not necessarily a bad thing.

It is a signal that you are not confident in your knowledge.

Use this signal to guide your study.

Parse Trees

Parse Trees

There are at least two forms of trees
that we might refer to “parse trees”

Derivation Tree

Derivation Tree

Describes exactly how input was parsed

Derivation Tree

Describes exactly how input was parsed

```
e ::= n | e+e | e-e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Derivation Tree

Describes exactly how input was parsed

```
e ::= n | e+e | e-e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

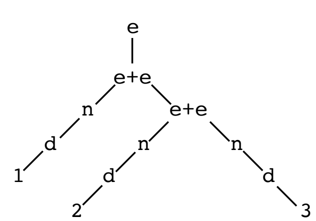
1+2+3

Derivation Tree

Describes exactly how input was parsed

```
e ::= n | e+e | e-e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

1+2+3



Abstract Syntax Tree

```
e ::= n | e+e | e-e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Abstract Syntax Tree

Abstracts over representation details

```
e ::= n | e+e | e-e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Abstract Syntax Tree

Abstracts over representation details

```
e ::= n | e+e | e-e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

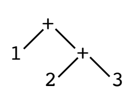
1+2+3

Abstract Syntax Tree

Abstracts over representation details

```
e ::= n | e+e | e-e
n ::= d | nd
d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

1+2+3



Solution to reduction in reading prompt

```
(λa.λb.(- a b)) 2 1
```

Solution to reduction in reading prompt

`(λa.λb.(- a b)) 2 1`

Anyone want to give this a try on the board?

Activity

Activity

`(λf.λx.f(f x))(λz.(+ x z))2`

Activity

Normal order reduction:

`(λf.λx.f(f x))(λz.(+ x z))2`

Activity

Activity

(λf.λx.f(f x))(λz.(+ x z))2

Activity

Applicative order reduction:

(λf.λx.f(f x))(λz.(+ x z))2

Computability

Computability

Computability

i.e., what can and cannot
be done with a computer

Computability

i.e., what can and cannot
be done with a computer

def: a function *f* is **computable** if there
is a program *P* that computes *f*.

Computability

i.e., what can and cannot
be done with a computer

def: a function *f* is **computable** if there
is a program *P* that computes *f*.

In other words, for **any** (valid) input *x*, the
computation *P(x)* **halts** with output *f(x)*.

Computability

Computability

example

Computability

example

valid inputs are **integers**

Computability

example

valid inputs are **integers**

P(x) is:

$f(x) = x + 5$

Computability

<u>example</u>

valid inputs are **integers**

P(x) is:
f(x) = x + 5

computable?

Computability

<u>example</u>

valid inputs are **integers**

P(x) is:
f(x) = x + 5

computable?
yes.

Computability

Computability

<u>example</u>

Computability

example

valid inputs are **integers**

Computability

example

valid inputs are **integers**

P(x) is:
f(x) = 5/x

Computability

example

valid inputs are **integers**

P(x) is:
f(x) = 5/x

computable?

Computability

example

valid inputs are **integers**

P(x) is:
f(x) = 5/x

computable?
yes, *partially*.

Total Function

Total Function

$f\colon A \rightarrow B$ is a subset $f \subseteq A \times B$ subject to

Total Function

$f\colon A \rightarrow B$ is a subset $f \subseteq A \times B$ subject to

1. for every $a \in A$, there is a $b \in B$ with $\langle a, b \rangle \in f$

Total Function

$f\colon A \rightarrow B$ is a subset $f \subseteq A \times B$ subject to

1. for every $a \in A$, there is a $b \in B$ with $\langle a, b \rangle \in f$ **totality**

Total Function

f: A→B is a subset f ⊆ A×B subject to

1. for every a∈A, there is a b∈B with ⟨a,b⟩∈f
2. if ⟨a,b⟩∈f and ⟨a,c⟩∈f then b=c

Total Function

f: A→B is a subset f ⊆ A×B subject to

1. for every a∈A, there is a b∈B with ⟨a,b⟩∈f **totality**
2. if ⟨a,b⟩∈f and ⟨a,c⟩∈f then b=c

Total Function

f: A→B is a subset f ⊆ A×B subject to

1. for every a∈A, there is a b∈B with ⟨a,b⟩∈f **totality**
2. if ⟨a,b⟩∈f and ⟨a,c⟩∈f then b=c **single valued**

Total Function

f: A→B is a subset f ⊆ A×B subject to

1. for every a∈A, there is a b∈B with ⟨a,b⟩∈f **totality**
2. if ⟨a,b⟩∈f and ⟨a,c⟩∈f then b=c **single valued**

e.g,
f(x) = x + 5

Partial Function

f: A→B is a subset f ⊆ A×B subject to

1. for every a∈A, there is a b∈B with ⟨a,b⟩∈f **totality**
2. if ⟨a,b⟩∈f and ⟨a,c⟩∈f then b=c **single valued**

Partial Function

f: A→B is a subset f ⊆ A×B subject to

1. ~~for every a∈A, there is a b∈B with ⟨a,b⟩∈f~~ **totality**

2. if ⟨a,b⟩∈f and ⟨a,c⟩∈f then b=c **single valued**

Partial Function

f: A→B is a subset f ⊆ A×B subject to

1. ~~for every a∈A, there is a b∈B with ⟨a,b⟩∈f~~ **totality**

2. if ⟨a,b⟩∈f and ⟨a,c⟩∈f then b=c **single valued**

e.g,

f(x) = 5/x

Activity

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,

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

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

Clarifications:

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

Clarifications:

`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,

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

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,

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

How might this work?

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,

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

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`

The Halting Problem

The Halting Problem

Notes on the proof:

The Halting Problem

Notes on the proof:

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

The Halting Problem

Notes on the proof:

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

Literally: “reduction to absurdity”.

The Halting Problem

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.

The Halting Problem

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.

The Halting Problem

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.

The Halting Problem

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.

Therefore, the presupposition must be false.

The Halting Problem

The Halting Problem

Notes on the proof:

The Halting Problem

Notes on the proof:

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

The Halting Problem

Notes on the proof:

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

Remember: we are looking to produce a contradiction.

The Halting Problem

Notes on the proof:

The proof relies on the kind of substitution principle 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

The Halting Problem: Proof

Suppose:

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

The Halting Problem: Proof

Suppose:

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

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 }

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

The Halting Problem: Proof

Observations so far:

The Halting Problem: Proof

Observations so far:

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

The Halting Problem: Proof

Observations so far:

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

The Halting Problem: Proof

Observations so far:

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

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

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

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

The Halting Problem

Isn't `DNH` itself a program?

The Halting Problem

Isn't `DNH` itself a program?

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

The Halting Problem

Isn't `DNH` itself a program?

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

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

The Halting Problem

Isn't `DNH` itself a program?

What happens if we call `DNH(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)`?

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

This literally makes no sense.

The Halting Problem

Isn't `DNH` itself a program?

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

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

This literally makes no sense.

`Foo` is true if `Foo` is false.

The Halting Problem

Isn't `DNH` itself a program?

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

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

This literally makes no sense.

`Foo` is true if `Foo` is false.
`Foo` is false if `Foo` is true.

The Halting Problem

Isn't `DNH` itself a program?

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

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

This literally makes no sense.

`Foo` is true if `Foo` is false.
`Foo` is false if `Foo` is true.

Therefore, the `Halt` function cannot exist.

Next class:

How we can use the Halting Problem to show that other problems cannot be solved (in general) by "reduction" to the Halting Problem.

Next class:

How we can use the Halting Problem to show that other problems cannot be solved (in general) by “reduction” to the Halting Problem.

We cannot tell, in general…

Next class:

How we can use the Halting Problem to show that other problems cannot be solved (in general) by “reduction” to the Halting Problem.

We cannot tell, in general…

… if a program will run forever.

Next class:

How we can use the Halting Problem to show that other problems cannot be solved (in general) by “reduction” to the Halting Problem.

We cannot tell, in general…

… if a program will run forever.

… if a program eventually produces an error.

Next class:

How we can use the Halting Problem to show that other problems cannot be solved (in general) by “reduction” to the Halting Problem.

We cannot tell, in general…

… if a program will run forever.

… if a program eventually produces an error.

… if a program will re-read an item in memory.