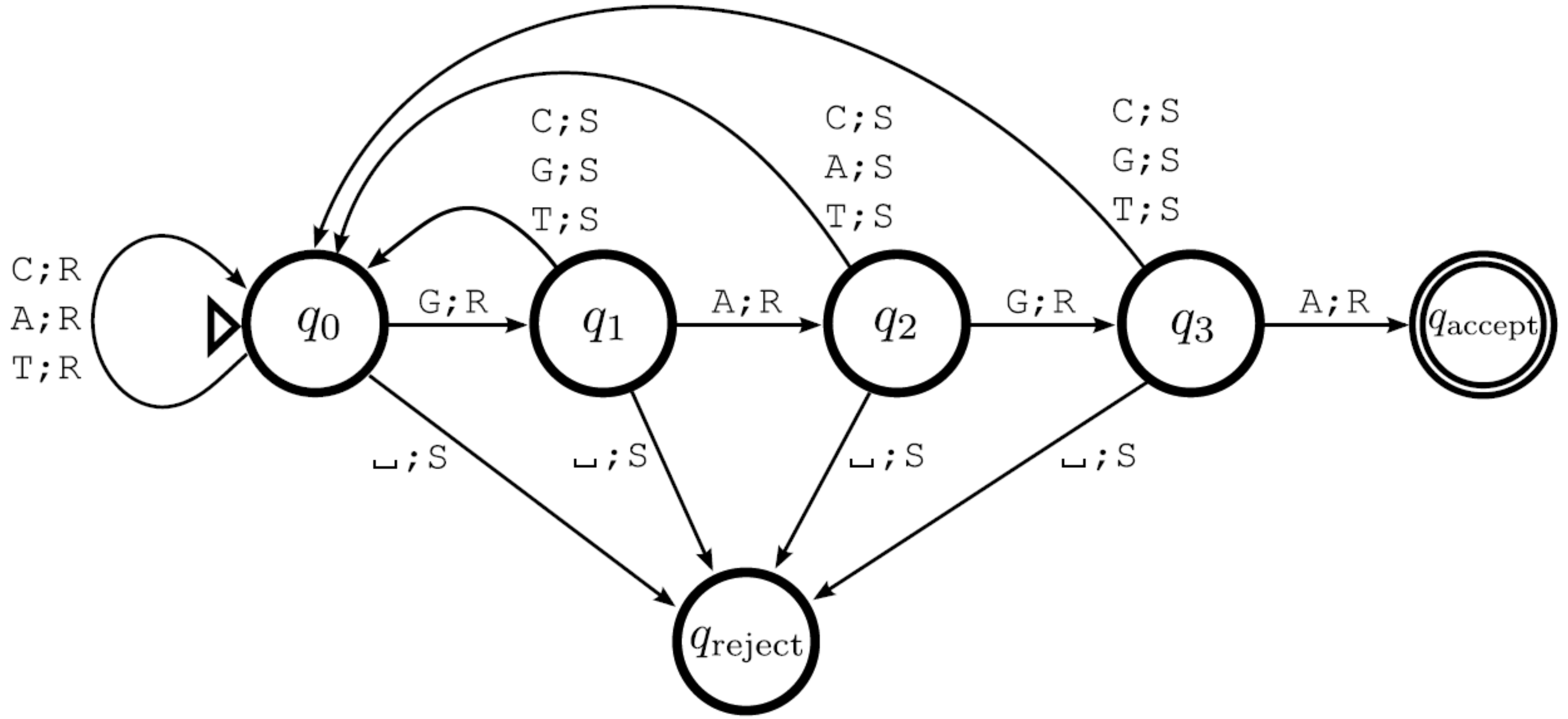
Undecidability and more, using real computer programs

John MacCormick

Dickinson College and University of East Anglia

Computer programs vs Turing machines

```
def containsGAGA(inString):
    if 'GAGA' in inString:
        return 'yes'
    else:
        return 'no'
```



rf ≡ readFile

```
>>> rf('wasteland.txt')

>>> rf('geneticString.txt')

>>> rf('containsGAGA.py')
```

Programs can analyze other programs, and they can analyze themselves

- A program analyzing another program:

```
>>> countLines(rf('containsGAGA.py'))
```

- A program analyzing itself:

```
>>> countLines(rf('countLines.py'))
```

- [demo: Word reading Word]

Some example decision programs we will need:

```
def yes(inString):
    return 'yes'
```

```
def longerThan1K(inString):
    if len(inString) > 1000:
        return 'yes'
    else:
        return 'no'
```

Suggestion: fill in this table interactively

Shell command (with `rf = utils.readFile`)	Output
`containsGAGA('CTGAGAT')`	
`containsGAGA(rf('geneticString.txt'))`	
`containsGAGA(rf('longerThan1K.py'))`	
`containsGAGA(rf('containsGAGA.py'))`	
`yes('CTGAGAT')`	
`yes(rf('geneticString.txt'))`	
`yes(rf('containsGAGA.py'))`	
`yes(rf('yes.py'))`	
`longerThan1K('CTGAGAT')`	
`longerThan1K(rf('geneticString.txt'))`	
`longerThan1K(rf('containsGAGA.py'))`	
`longerThan1K(rf('longerThan1K.py'))`	

Suggestion: fill in this table interactively

Shell command (with `rf = utils.readFile`)	Output
`containsGAGA('CTGAGAT')`	yes
`containsGAGA(rf('geneticString.txt'))`	yes
`containsGAGA(rf('longerThan1K.py'))`	no
`containsGAGA(rf('containsGAGA.py'))`	yes
`yes('CTGAGAT')`	yes
`yes(rf('geneticString.txt'))`	yes
`yes(rf('containsGAGA.py'))`	yes
`yes(rf('yes.py'))`	yes
`longerThan1K('CTGAGAT')`	no
`longerThan1K(rf('geneticString.txt'))`	yes
`longerThan1K(rf('containsGAGA.py'))`	no
`longerThan1K(rf('longerThan1K.py'))`	no

Definition of yesOnString.py

$$\texttt{yesOnString.py}(P, I) = \begin{cases} \text{“}\texttt{yes}\text{”} & \text{if } P \text{ is a Python program, } P(I) \\ & \text{is defined, and } P(I) = \text{“}\texttt{yes}\text{”}, \\ \text{“}\texttt{no}\text{”} & \text{otherwise.} \end{cases}$$

Suggestion: fill in this table interactively

shell command (with `rf = utils.readFile`)	output
`yesOnString('not a program','CAGT')`	
`yesOnString(rf('containsGAGA.py'),'CAGT')`	
`yesOnString(rf('containsGAGA.py'),rf('containsGAGA.py'))`	
`yesOnString(rf('yes.py'),'CAGT')`	
`yesOnString(rf('yes.py'),rf('yes.py'))`	
`yesOnString(rf('longerThan1K.py'),rf('geneticString.txt'))`	
`yesOnString(rf('longerThan1K.py'),rf('longerThan1K.py'))`	

$$\texttt{yesOnString.py}(P, I) = \begin{cases} \text{“}\texttt{yes}\text{”} & \text{if } P \text{ is a Python program, } P(I) \\ & \text{is defined, and } P(I) = \text{“}\texttt{yes}\text{”}, \\ \text{“}\texttt{no}\text{”} & \text{otherwise.} \end{cases}$$

Solutions for yesOnString.py:

shell command (with `rf = utils.readFile`)	output
`yesOnString('not a program','CAGT')`	`no`
`yesOnString(rf('containsGAGA.py'),'CAGT')`	`no`
`yesOnString(rf('containsGAGA.py'),rf('containsGAGA.py'))`	`yes`
`yesOnString(rf('yes.py'),'CAGT')`	`yes`
`yesOnString(rf('yes.py'),rf('yes.py'))`	`yes`
`yesOnString(rf('longerThan1K.py'),rf('geneticString.txt'))`	`yes`
`yesOnString(rf('longerThan1K.py'),rf('longerThan1K.py'))`	`no`

$$\texttt{yesOnString.py}(P, I) = \begin{cases} \text{“}\texttt{yes}\text{”} & \text{if } P \text{ is a Python program, } P(I) \text{ is defined, and } P(I) = \text{“}\texttt{yes}\text{”}, \\ \text{“}\texttt{no}\text{”} & \text{otherwise.} \end{cases}$$

Definition of yesOnSelf.py

$$\texttt{yesOnSelf.py}(P) = \begin{cases} \text{``}\texttt{yes}\text{''} & \text{if } P \text{ is a Python program, } P(P) \\ & \text{is defined, and } P(P) = \text{``}\texttt{yes}\text{''}, \\ \text{``}\texttt{no}\text{''} & \text{otherwise.} \end{cases}$$

Suggestion: fill in this table interactively

shell command (with `rf = utils.readFile`)	output
`yesOnSelf('not a program')`	
`yesOnSelf(rf('containsGAGA.py'))`	
`yesOnSelf(rf('yes.py'))`	
`yesOnSelf(rf('longerThan1K.py'))`	
`yesOnSelf(rf('yesOnSelf.py'))`	

$$\texttt{yesOnSelf.py}(P) = \begin{cases} \text{“}\texttt{yes}\text{”} & \text{if } P \text{ is a Python program, } P(P) \\ & \text{is defined, and } P(P) = \text{“}\texttt{yes}\text{”}, \\ \text{“}\texttt{no}\text{”} & \text{otherwise.} \end{cases}$$

Solutions for yesOnSelf.py:

shell command (with `rf = utils.readFile`)	output
`yesOnSelf('not a program')`	no
`yesOnSelf(rf('containsGAGA.py'))`	yes
`yesOnSelf(rf('yes.py'))`	yes
`yesOnSelf(rf('longerThan1K.py'))`	no
`yesOnSelf(rf('yesOnSelf.py'))`	yes or no (!?)

$$\texttt{yesOnSelf.py}(P) = \begin{cases} \text{“}\texttt{yes}\text{”} & \text{if } P \text{ is a Python program, } P(P) \\ & \text{is defined, and } P(P) = \text{“}\texttt{yes}\text{”}, \\ \text{“}\texttt{no}\text{”} & \text{otherwise.} \end{cases}$$

notYesOnSelf.py reverses yesOnSelf.py

$$\texttt{yesOnSelf.py}(P) = \begin{cases} \text{“}\texttt{yes}\text{”} & \text{if } P \text{ is a Python program, } P(P) \\ & \text{is defined, and } P(P) = \text{“}\texttt{yes}\text{”}, \\ \text{“}\texttt{no}\text{”} & \text{otherwise.} \end{cases}$$

$$\texttt{notYesOnSelf.py}(P) = \begin{cases} \text{“}\texttt{no}\text{”} & \text{if } P \text{ is a Python program, } P(P) \\ & \text{is defined, and } P(P) = \text{“}\texttt{yes}\text{”}, \\ \text{“}\texttt{yes}\text{”} & \text{otherwise.} \end{cases}$$

Suggestion: use the earlier results to fill in the bottom table interactively

shell command (with `rf = utils.readFile`)	output
`yesOnSelf('not a program')`	no
`yesOnSelf(rf('containsGAGA.py'))`	yes
`yesOnSelf(rf('yes.py'))`	yes
`yesOnSelf(rf('longerThan1K.py'))`	no
`yesOnSelf(rf('yesOnSelf.py'))`	yes or no (!?)

shell command (with `rf = utils.readFile`)	output
`notYesOnSelf('not a program')`	
`notYesOnSelf(rf('containsGAGA.py'))`	
`notYesOnSelf(rf('yes.py'))`	
`notYesOnSelf(rf('longerThan1K.py'))`	
`notYesOnSelf(rf('notYesOnSelf.py'))`	

Solutions for yesOnSelf.py and notYesOnSelf.py :

shell command (with `rf = utils.readFile`)	output
`yesOnSelf('not a program')`	no
`yesOnSelf(rf('containsGAGA.py'))`	yes
`yesOnSelf(rf('yes.py'))`	yes
`yesOnSelf(rf('longerThan1K.py'))`	no
`yesOnSelf(rf('yesOnSelf.py'))`	yes or no (!?)

shell command (with `rf = utils.readFile`)	output
`notYesOnSelf('not a program')`	yes
`notYesOnSelf(rf('containsGAGA.py'))`	no
`notYesOnSelf(rf('yes.py'))`	no
`notYesOnSelf(rf('longerThan1K.py'))`	yes
`notYesOnSelf(rf('notYesOnSelf.py'))`	**!?!?!?**

No output is correct here…

… therefore, notYesOnSelf.py cannot exist!

If yesOnString.py existed, we could create notYesOnSelf.py

```
from yesOnString import yesOnString
def yesOnSelf(progString):
    return yesOnString(progString, progString)
```

```
def notYesOnSelf(progString):
    val = yesOnSelf(progString)
    if val == 'yes':
        return 'no'
    else:
        return 'yes'
```

Therefore, yesOnString.py can't exist either

Proof:

1. Assume yesOnString.py exists
2. Create notYesOnSelf.py as on previous slide (and summarized here)
3. This contradicts the impossibility of notYesOnSelf.py

Program name	**Program behavior**
$\texttt{yesOnString}(P, I)$	– return "yes", if $P(I) =$ "yes" – return "no", otherwise
$\downarrow$	
$\texttt{yesOnSelf(P)}$	– return "yes", if $P(P) =$ "yes" – return "no", otherwise
$\downarrow$	
$\texttt{notYesOnSelf}(P)$	– return "no", if $P(P) =$ "yes" – return "yes", otherwise

By combining many tricks into one program, a much briefer proof is possible

```python
from yesOnString import yesOnString
def weirdYesOnString(progString):
    if yesOnString(progString, progString)=='yes':
        return 'no'
    else:
        return 'yes'
```

Proof that yesOnString.py doesn’t exist:

1. Assume yesOnString.py exists
2. Create weirdYesOnString.py as above
3. Observe that weirdYesOnString.py produces a contradiction when given itself as input (it outputs “yes” if and only if it outputs “no”)

Similar reasoning shows that no program can correctly predict, for all possible inputs, whether other programs will crash

Proof that crashOnString.py doesn't exist:

1. Assume crashOnString.py exists
2. Create weirdCrashOnSelf.py as shown
3. Observe that weirdCrashOnSelf.py produces a contradiction when given itself as input (crashes if and only if it doesn't crash)

Program name	**Program behavior**
`crashOnString`(P, I)	– return "yes", if P crashes on input I – return "no", otherwise
↓	
`crashOnSelf`(P)	– return "yes", if P crashes on input P – return "no", otherwise
↓	
`weirdCrashOnSelf`(P)	– return without crashing, if P crashes on input P – crash, otherwise

Be careful to interpret the “impossibility of bug-finding programs” correctly

- It is true that no program *P* can correctly predict, for all programs *Q*, whether *Q* will crash
- However, *P* might work correctly on many inputs
- Software companies and academic researchers invest great effort in doing exactly this: developing programs *P* that work efficiently and correctly on useful classes of software

Many other ideas from theoretical computer science can be taught using real computer programs

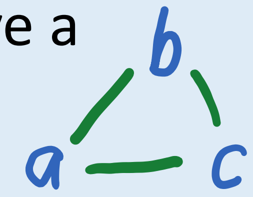
Examples:

- Universal computation
- Non-decision programs
- Complexity theory
- Gödel’s incompleteness theorem

Universal Python program

```
def universal(progString, inString):
    # Execute the definition of the function in progString. This defines
    # the function, but doesn't invoke it.
    exec(progString)
    # Now that the function is defined, we can extract a reference to it.
    progFunction = utils.extractMainFunction(progString,
                                             locals())
    # Invoke the desired function with the desired input string.
    return progFunction(inString)
```

Use *non-decision* problems for better learning outcomes

	Traditional (decision)	**Practical (nondecision)**
HamCycle	Does this graph have a Hamilton cycle?  e.g. “`a,b b,c c,a`” $\mapsto$ “`yes`”	Please give me a Hamilton cycle of this graph. e.g. “`a,b b,c c,a`” $\mapsto$ “`a,b,c`”
Factor	Does this integer have a nontrivial factor? e.g. “`51295697`” $\mapsto$ “`yes`”	Please give me a nontrivial factor of this integer. e.g. “`51295697`” $\mapsto$ “`8779`”

Prove results in complexity theory

Example: this program provides a proof that we can’t determine in sub-exponential time whether or not a program requires super-exponential time

```
# Import H, which is assumed to solve HaltsInExpTime in polynomial
# time. (In reality, we will prove that H cannot exist.)
from H import H
def weirdH(progString):
    if H(progString, progString) == 'yes':
        # deliberately enter infinite loop
        print('Entering infinite loop...')
        utils.loop()
    else:
        # The return value is irrelevant, but the fact that we return
        # right away is important.
        return 'finished already!'
```

We can even prove Gödel's first incompleteness theorem!

```
def godel(inString):
    godelProg = rf('godel.py')
    haltInPeano = convertHaltToPeano(godelProg)
    notHaltInPeano = 'NOT ' + haltInPeano
    if provableInPeano(notHaltInPeano) == 'yes':
        return 'halted' # any value would do
    else: # This line will never be executed! But anyway...
        utils.loop() # deliberate infinite loop
```

An ASCII string representing a statement in number theory that is true but unprovable!

That sounds interesting. How can I learn CS theory using real computer programs?

Answer: There is a new text book from Princeton University Press that takes this approach:

> *What Can Be Computed?: A Practical Guide to the Theory of Computation*

Also, there's a SIGCSE paper: "Strategies for basing the CS theory course on non-decision problems." In *Proc. ACM SIGCSE*, pp521-526, 2018.