Exam 2: Out at end of class today, due Tuesday at 2:01pm

Menu

- Turing-equivalent Grammar
- Universal Programming Languages
- “Return-Oriented Programming”
- Problems Computers Can and Cannot Solve

What does undecidability mean for problems real people (not just CS theorists and Busy Beavers) care about?

Models of Computation

<table>
<thead>
<tr>
<th>Machine</th>
<th>Replacement Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite Automata (Class 2-5)</td>
<td>Regular Grammar ($A \rightarrow aB$)</td>
</tr>
<tr>
<td>Pushdown Automata (add a stack) (Classes 6-7)</td>
<td>Context-free Grammar ($A \rightarrow BC$)</td>
</tr>
<tr>
<td>Turing machine (add an infinite tape) (Classes 14-20)</td>
<td>?</td>
</tr>
</tbody>
</table>

Unrestricted Grammar

Right and left sides of a grammar rule can be any sequence of terminals and nonterminals.

$$aXbY \rightarrow cZd$$

How can we prove unrestricted grammars are equivalent to a Turing machine?

Simulation Proof

**$L(UG) \subseteq L(TM)$**

Show that we can simulate every Unrestricted Grammar with some TM. Fairly easy, but tedious (not shown): design a TM that does the grammar replacements by writing on the tape.

**$L(TM) \subseteq L(UG)$**

Show that we can simulate every TM with some Unrestricted Grammar.

Simulating TM with UG

$$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$
Simulating TM with UG

Grammar variables = \( Q \cup \{ R \} \)
Grammar terminals = \( \Gamma \)
String = \(<\text{left tape}>Q_{\text{current}}<\text{right tape}>\)
for all \( z \in \Gamma, q_i, q_j \in Q - \{ q_{\text{accept}}, q_{\text{reject}} \}, a, b \in \Gamma \)
\[ \delta(q_i, a) = (q_j, b, L) \quad \Rightarrow \quad zQ_i a \rightarrow Q_j z b \]
\[ \delta(q_i, a) = (q_j, b, R) \quad \Rightarrow \quad Q_i a \rightarrow b Q_j \]

Initia Configuration

\[ S \rightarrow Q_0 w \]

Models of Computation

<table>
<thead>
<tr>
<th>Machine</th>
<th>Replacement Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite Automata (Class 2-5)</td>
<td>Regular Grammar (A \rightarrow aB)</td>
</tr>
<tr>
<td>Pushdown Automata (add a stack) (Classes 6-7)</td>
<td>Context-free Grammar (Classes 7-9) (A \rightarrow BC)</td>
</tr>
<tr>
<td>Turing machine (add an infinite tape) (Classes 14-20)</td>
<td>Unrestricted Grammar (Class 21) (\alpha \rightarrow \beta)</td>
</tr>
</tbody>
</table>

Universal Programming Language

- Definition: a programming language that can describe every algorithm.
- Equivalently: a programming language that can simulate every Turing Machine.
- Equivalently: a programming language in which you can implement a Universal Turing Machine.

Which of these are Universal Programming Languages?

- BASIC
- C++
- COBOL
- C#
- PostScript
- Python
- x86
- Java
- HTML
- PDF
- JavaScript
- Fortran
- Scheme
- TeX
- Ruby
- TeX
- Ruby
Proofs

- BASIC, C, C++, C#, Fortran, Java, JavaScript, PDF, PostScript, Python, Ruby, Scheme, TeX, etc. are universal
  - Proof: implement a TM simulator in the PL
- HTML (before HTML5) is not universal:
  - Proof: show some algorithm that cannot be implemented in HTML
    - An infinite loop
  - HTML5 might be a universal programming language! (Proof is worth challenge bonus.)

Why is it impossible for a programming language to be both universal and resource-constrained?

Resource-constrained means it is possible to determine an upper bound on the resources any program in the language can consume.

All universal programming language are equivalent in power: they can all simulate a TM, which can carry out any mechanical algorithm.

Proliferation of Universal PLs

- "Aesthetics"
  - Some people like \( \Rightarrow \), others prefer \( = \).
  - Some people think whitespace shouldn't matter (e.g., Java), others think programs should be formatted like they mean (e.g., Python)
  - Some people like goto, others like throw.
- Expressiveness vs. Simplicity
  - Hard to write programs in SUBLEG
- Expressiveness vs. "Truthiness"
  - How much you can say with a little code vs. how likely it is your code means what you think it does

<table>
<thead>
<tr>
<th>Expressiveness</th>
<th>&quot;Truthiness&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>high</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Spec#</th>
<th>Ada</th>
<th>Java</th>
<th>x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>more mistake prone</td>
<td>strict typing, static</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Do most x86 programs contain Universal Turing Machines?


---

x86 programs are just sequences of bytes (1 byte = 8 bits = 2 hex characters)

Instructions are encoded using variable-length (1-15 bytes)

First byte is opcode that identifies the type of instruction

```
bb ef be ad de  MOV $0xDEADBEEF, %eax
```

5-byte instruction that writes the constant 0xDEADBEEF into register %eax

```
c3 RET
```

1-byte instruction that returns (jumps to the return address that is stored in a location on the stack)

```
eb fe JMP -2
```

2-byte instruction that moves the instruction pointer back 2

---

“Return-Oriented Programming”

---

Injecting Malicious Code
Buffer Overflows

```c
int main (void) {
    int x = 9;
    char s[4];
    gets(s);
    printf ("s is: %s\n", s);
    printf ("x is: %d\n", x);
}
```

```bash
> gcc -o bounds bounds.c
> bounds

```

```
abcdefghijklmnopqrstuvwxyz
s is: abcdefghijkl
x is: 9
```

```
> bounds

```

```
abcdefghijklmnopqrstuvwxyz
s is: abcdefghijkl
x is: 9
```

```
> bounds

```

```
> bounds

```

```
> bounds

```

What does this kind of mistake look like in a popular server?

Defenses

- Use a type-safe programming language (e.g., bounds checking)
- Write-xor-Execute pages
  - When the OS loads a page into memory, it is marked as either executable or writable: can't be both
  - Hence: attacker can inject all the code it wants on the stack, but can't jump to it and execute it

“Return-Oriented Programming”

```
41 AC 16 FA A0 44 79 8C 2D 43 B3 47 6C 62
69 80 B8 C9 5F B8 34 59 89 4E 75 25 A5 64
5F 5A 5B 6B 6F 68 84 8E 36 7B 7D 91 C7 D3 E1 49 DB A5
FE A4 61 5C 5D E4 8C 8D 6C 33 C3 EC 7E 27
F7 88 85 37 5E F9 80 B0 88 89 42 11 43 4F 6B
57 83 5E 67 79 8B 48 9A 77 FC 1D 0D 0D 0D 0D
9B 38 OB 57 8B 58 8B 8D A7 CE F5 96 50 5B
36 82 0F DA 39 16 35 55 6B 17 C0 7F 89 52 89
FA C7 4C 1F EF B7 56 00 17 82 0F DA 39 16 35 55 6B
D8 76 B1 DD F2 4E DF 3F C6 FF A7 BF 4B 89
D4 11 2F 3F 4F F7 93 B5 CB 6A AA D1 E1 E6
```

Does it really work?

- Likelihood of finding enough gadgets in “random” bytes to make Turing-complete
  libc (C library included in nearly all Unix programs) contains more than enough (18MB ~ expect to have ~ 74000 RET (c3) instructions)
- Demonstration of attack on voting machine:

  [http://www.youtube.com/watch?v=IsfG3KPrD1](http://www.youtube.com/watch?v=IsfG3KPrD1)

Vulnerability Detection

Input: an x86 program P

Output: True if there is some input w, such that running P on w allows the attacker to overwrite return addresses on stack; False otherwise.
Example: Morris Internet Worm (1988)

\[ P = \text{fingerd} \]

- Program used to query user status (running on most Unix servers)

\[ \text{isVulnerable}(P)? \]

Yes, for \( w = \text{"nop\(^{400}\) pushl $68732f pushl $6e69622f movl sp,r10 pushl $0 pushl $0 pushl r10 pushl $3 movl sp,ap chmk $3b"} \)

- Worm infected several thousand computers (~10% of Internet in 1988)

Vulnerability Detection

**Input:** an x86 program \( P \)

**Output:** \( \text{True} \) if there is some input \( w \), such that running \( P \) on \( w \) allows the attacker to overwrite return addresses on stack; \( \text{False} \) otherwise.

\[ L_{\text{HACKABLE}} = \{ \langle P \rangle \mid \text{where } P \text{ is an x86 program and there exists some input } w \in \{0,1\}^* \text{ that causes a return address on the stack to be overwritten } \} \]

Vulnerability Detection is Undecidable

Assume \( L_{\text{HACKABLE}} \) is decidable.

Then, some TM \( M_{\text{HACKABLE}} \) decides it.

We can use \( M_{\text{HACKABLE}} \) to construct a machine that decides \( \text{HALT}_{TM} \):

\[ M_{H}(\langle M, w \rangle) : \]

1. Simulate \( M \) on \( w \) and hack stack
2. Simulate \( M_{\text{HACKABLE}}(\langle M, w \rangle) \) and output result.

Vulnerability Detection is Undecidable

Assume \( L_{\text{HACKABLE}} \) is decidable.

Then, some TM \( M_{\text{HACKABLE}} \) decides it.

We can use \( M_{\text{HACKABLE}} \) to construct a machine that decides \( \text{HALT}_{TM} \):

\[ M_{\text{HALTS}}(\langle M, w \rangle) = \text{Construct } P, \text{ an x86 program that:} \]

1. Simulates \( M \) on input \( w \).
2. Overwrites a return address on the stack.

Simulate \( M_{\text{HACKABLE}}(P) \) and output the result.

“Solving” Undecidable Problems

- Undecidable means there is no program that
  1. \textit{Always} gives the correct answer, and
  2. \textit{Always} terminates
- Must give up one of these:
  - Giving up #2 is not acceptable in most cases
  - Must give up #1: cannot be correct on all inputs
- \textit{Or} change the problem
  - e.g., modify \( P \) to make it invulnerable, etc.

“Impossibility” of Vulnerability Detection
Actual Vulnerability Detectors

- Sometimes give the wrong answer:
  - “False positive”: say P is a vulnerable when it isn’t
  - “False negative”: say P is safe when it is
- Heuristics to find common errors
- Heuristics to rank-order possible problems

Can Microsoft squash 63,000 bugs in Windows 2000?
... Overall, there are more than 65,000 “potential issues” that could emerge as problems, as discovered by Microsoft’s Prefix tool. Microsoft is estimating that 28,000 of these are likely to be “real” problems.

Computability in Theory and Practice
(Inellectual Computability Discussion on TV)

http://video.google.com/videoplay?docid=1623254076490030585#

Ali G Problem

**Input:** a list of numbers (mostly 9s)
**Output:** the product of the numbers

$L_{ALIG} = \{ < k_1, k_2, \ldots, k_n, p> | \text{each } k_i \text{ represents a number and } p \text{ represents a number that is the product of all the } k_i \text{s. numbers } \}$

Is $L_{ALIG}$ decidable? **Yes.** It is easy to see a simple algorithm (e.g., elementary school multiplication) that decides it.

Can real computers solve it?

Ali G was Right!

- Theory assumes ideal computers:
  - Unlimited, perfect memory
  - Unlimited (finite) time
- Real computers have:
  - Limited memory, time, power outages, flaky programming languages, etc.
  - There are many decidable problems we cannot solve with real computer: the actual inputs **do** matter (in practice, but not in theory!)
Charge

• Exam 2 out now
• Due at beginning of class, Tuesday

• It has some pretty tough questions (and no really easy questions): don’t get stressed out if you can’t answer everything