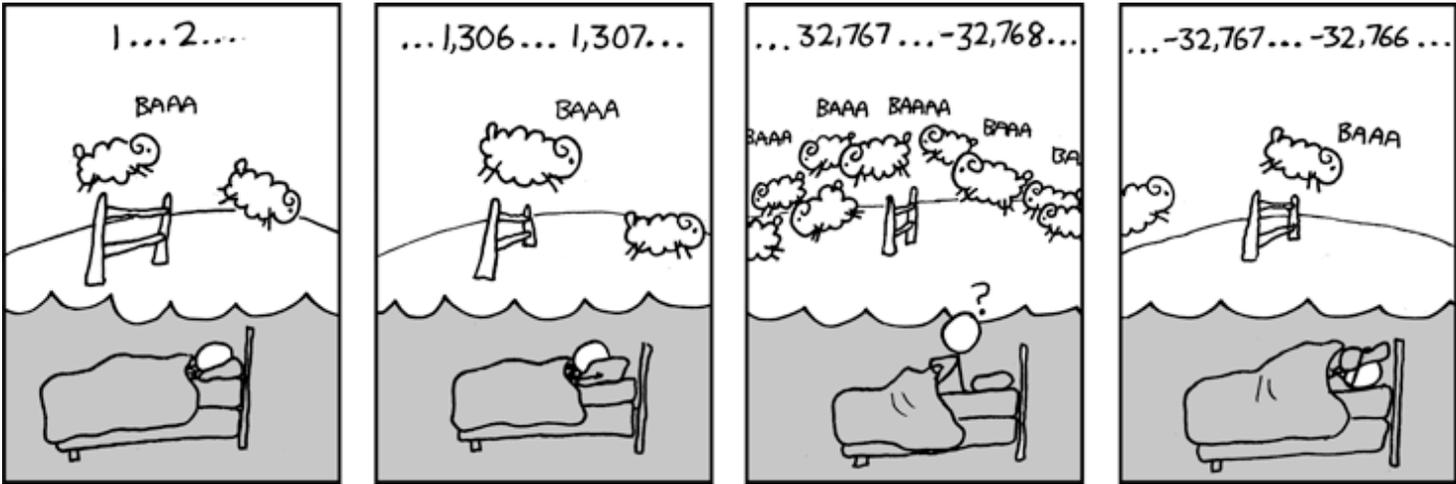


CSE 311: Foundations of Computing

Topic 7: Applications



Applications of Computability

Recall: Undecidability of the Halting Problem

CODE(P) means “the code of the program **P**”

The Halting Problem

Given: - CODE(P) for any program **P**
- input **x**

Output: **true** if **P** halts on input **x**
false if **P** does not halt on input **x**

Theorem [Turing]: There is no program that solves the Halting Problem

Recall: Proving Problems Undecidable

- Can use the fact that the Halting Problem is undecidable to show that other problems are undecidable

General method (a “reduction” from **A** to **B**):

Prove that, if there were a program deciding **B**, then there would be a program deciding the Halting Problem.

“**B** decidable \rightarrow **A** decidable”

Show how to use a solution to **B** to build a solution to **A**

Proves that **B** is *at least* as hard as **A**

Recall: Proving Problems Undecidable

- Can use the fact that the Halting Problem is undecidable to show that other problems are undecidable

General method (a “reduction” from Halting Problem):

Prove that, if there were a program deciding **B**, then there would be a program deciding the Halting Problem.

“**B** decidable \rightarrow **Halting Problem** decidable”

Contrapositive:

“**Halting Problem** undecidable \rightarrow **B** undecidable”

Therefore, **B** is undecidable

Some Problems About Code Are Decidable

Not every problem on programs is undecidable!

Which of these is decidable?

- Input $\text{CODE}(Q)$ and x
Output: **true** if Q prints “ERROR” on input x
after less than 100 steps
false otherwise
- Input $\text{CODE}(Q)$ and x
Output: **true** if Q prints “ERROR” on input x
after more than 100 steps
false otherwise

Another undecidable problem

```
public class Q {
    private static String x = "...";

    public static void main(String[] args) {
        PrintStream out = System.out;
        System.setOut(new PrintStream(
            new WriterOutputStream(new StringWriter())));
        System.setIn(new ReaderInputStream(new StringReader(x)));

        for (int i = 0; i < 100; i++); // waste 100 steps

        P.main(args);

        out.println("ERROR");
    }
}

class P {
    public static void main(String[] args) { ... }
    ...
}
```

Rice's Theorem

Not every problem on programs is undecidable...
but almost all of them are!

Rice's Theorem:

Any “non-trivial” property of the input-output behavior of Java programs is undecidable.

Rice's Theorem

Not every problem on programs is undecidable...
but almost all of them are!

Rice's Theorem (a.k.a. "Compilers

ARE DIFFICULT

Any "non-trivial" property of the input-output behavior of
Java programs is undecidable.

Proof of Rice's Theorem

Rice's Theorem:

Any “**non-trivial**” property of the **input-output behavior** of Java programs is undecidable.

- “**Non-trivial**” means there is at least one program with the property and at least one program without it.
 - if all programs have (or don't have) the property, it's easy
 - suppose that program **A** has the property and **B** does not
- Consider the infinite loop program **L**:

```
public static void main(String[] args) {  
    while (true);  
}
```

Proof of Rice's Theorem

- Suppose that program **A** has the property and **B** does not
- Suppose that **L** doesn't have the property
 - so **A** as the property and **L** doesn't
 - (other case is similar)
- Suppose, for a contradiction, that **T** checks for the property
- Given a program **P**, construct this program **Q**...

Proof of Rice's Theorem

```
public class Q {
    private static String x = "...";

    public static void main(String[] args) {
        InputStream in = System.in; // save System.in and System.out
        PrintStream out = System.out;
        System.setOut(new PrintStream(
            new WriterOutputStream(new StringWriter())));
        System.setIn(new ReaderInputStream(new StringReader(x)));

        P.main(args); // If P doesn't halt, then it acts like L

        System.setOut(out); // restore System.in and System.out
        System.setIn(in);

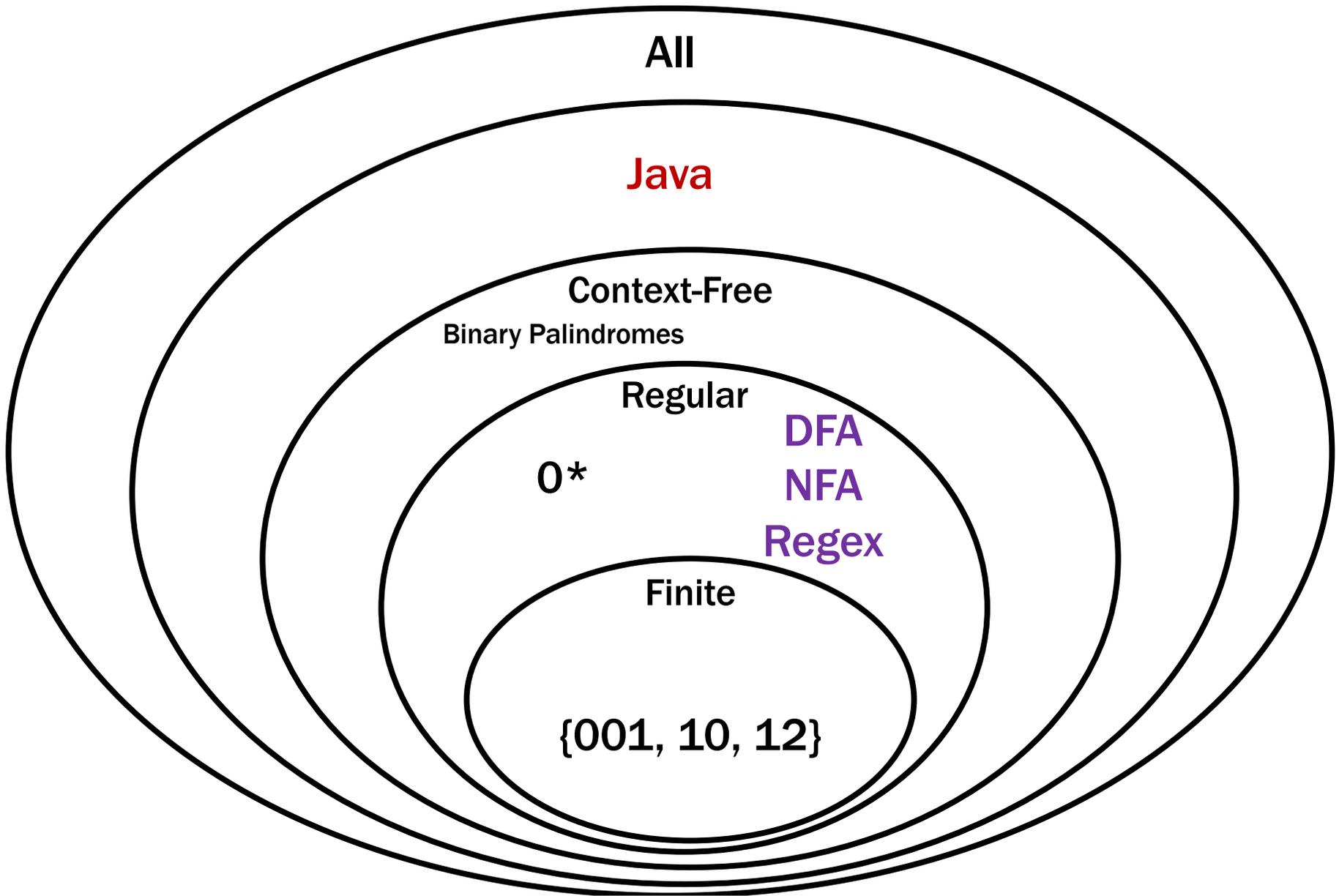
        A.main(args); // If P does halt, then it acts like A
    }
}

class P { ... }
```

Proof of Rice's Theorem

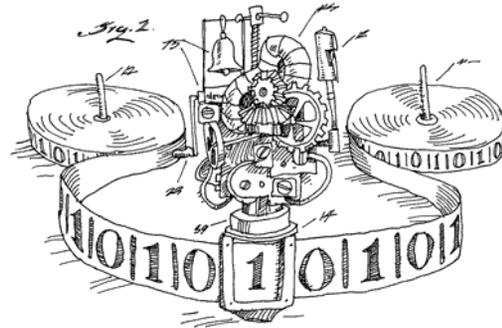
- Suppose that program **A** has the property and **B** does not
- Suppose that **L** has the property (other case is similar).
 - so **L** has the property and **B** does not
- Suppose, for a contradiction, that **T** checks for the property
- Construct this program **Q** that
 - acts like **A** (has the property) if **P** halts on x
 - acts like **L** (doesn't have property) if **P** doesn't halt on x
- Ask **T** if **Q** has the property
 - if it does, then **P** halts on x
 - if it doesn't, then **P** doesn't halt on x

Recall our language picture



Done

- **We proved that there is no computer program that can solve the Halting Problem.**
 - **There was nothing special about Java***
[Church-Turing thesis]



**This is essentially a law of physics
(not mathematics)**

Before Java

1930's:

How can we formalize what algorithms are possible?

- **Turing machines** (Turing, Post)
 - basis of modern computer hardware
- **Lambda Calculus** (Church, Kleene)
 - basis for functional programming languages

Turing machines

Church-Turing Thesis:

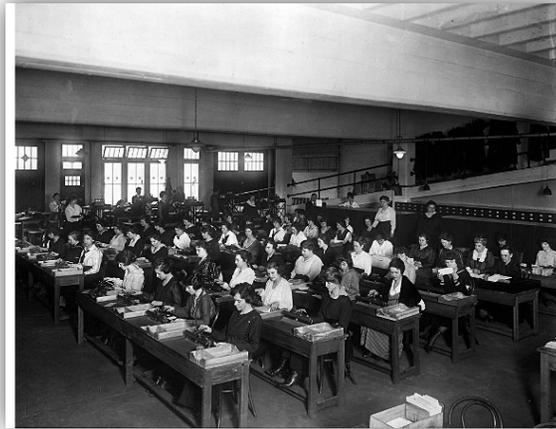
Any reasonable model of computation that includes all possible algorithms is equivalent in power to a Turing machine

Evidence

- Huge numbers of models based on radically different ideas turned out to be equivalent to TMs
- TM can simulate the physics of any machine that we could build (even quantum computers)
- In physics, this is considered a law of nature

Computers and algorithms

- Does Java (or any programming language) cover all possible computation? Every possible algorithm?
- There was a time when computers were people who did calculations on sheets paper to solve computational problems



- Computers as we know them arose from trying to understand everything these people could do.

Turing machines

- **Finite Control**
 - Brain/CPU that has only a finite # of possible “states of mind”
- **Recording medium**
 - An unlimited supply of blank “scratch paper” on which to write & read symbols, each chosen from a finite set of possibilities
 - Input also supplied on the scratch paper
- **Focus of attention**
 - Finite control can only focus on a small portion of the recording medium at once
 - Focus of attention can only shift a small amount at a time

Turing machines

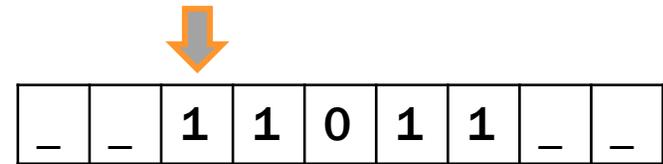
- **Recording medium**
 - An **infinite** read/write “**tape**” marked off into **cells**
 - Each **cell** can store **one symbol** or be “**blank**”
 - **Tape** is **initially** all **blank** except a few **cells** of the tape containing the input string
 - **Read/write head** can scan **one cell** of the tape - **starts** on input

Turing machines

- **In each step**, a Turing machine
 1. Reads the currently scanned cell
 2. Based on current state and scanned symbol
 - i. Overwrites symbol in scanned cell
 - ii. Moves read/write head left or right one cell
 - iii. Changes to a new state
- Each Turing Machine is specified by its **finite set of rules**

Turing machines

	-	0	1
s_1	(1, L, s_3)	(1, L, s_4)	(0, R, s_2)
s_2	(0, R, s_1)	(1, R, s_1)	(0, R, s_1)
s_3			
s_4			



UW CSE's Steam-Powered Turing Machine



Original in Sieg Hall stairwell

Turing machines

Ideal Java/C programs:

- Just like the Java/C you're used to programming with, except you never run out of memory
 - no `OutOfMemoryError`

Equivalent to Turing machines but easier to program:

- Turing machine definition is useful for breaking computation down into simplest steps
 - easier to analyze TMs than Java programs
- We only care about high level so we use programs

Turing-Complete Programming Languages

- Programming languages equal in power to Java / C / etc. are called "**Turing-complete**"
- **Some surprising languages are Turing-complete**
 - any general-purpose programming language you've heard of
proof: show that you write a TM simulator in your language
 - Excel macros
 - TypeScript's type system
 - Conway's "Game of Life"
 - Magic: The Gathering



Turing-Complete Programming Languages

- Turing's contemporary, Alonzo Church, proposed a different language called "**Lambda calculus**"
 - first "high level" programming language
 - a simple "functional" programming language
 - widely used in programming languages even today
- Turing's proof of equivalence
 - TM implements LC: world's first **interpreter**
 - LC implements TM: world's first **hardware simulator**

Turing's big idea part 1: Machines as data

Original Turing machine definition:

- A different “machine” **M** for each task
- Each machine **M** is defined by a finite set of possible operations on finite set of symbols
- So... **M** has a finite description as a sequence of symbols, its “code”, which we denote **<M>**

You already are used to this idea with the notion of the program code, but this was a new idea in Turing's time.

Turing's big idea part 2: A Universal TM

- A Turing machine interpreter **U**
 - On input $\langle M \rangle$ and its input x ,
U outputs the same thing as **M** does on input x
 - At each step it decodes which operation **M** would have performed and simulates it.
- One Turing machine is enough
 - Basis for modern stored-program computer (& "software")
Von Neumann studied Turing's UTM design



Recall: Context-Free Grammars

- A Context-Free Grammar (CFG) is given by a finite set of substitution rules involving
 - A finite set \mathbf{V} of *variables* that can be replaced
 - Alphabet Σ of *terminal symbols* that can't be replaced
 - One variable, usually \mathbf{S} , is called the *start symbol*
- The substitution rules involving a variable \mathbf{A} , written as

$$\mathbf{A} \rightarrow w_1 \mid w_2 \mid \cdots \mid w_k$$

where each w_i is a string of variables and terminals

- that is $w_i \in (\mathbf{V} \cup \Sigma)^*$

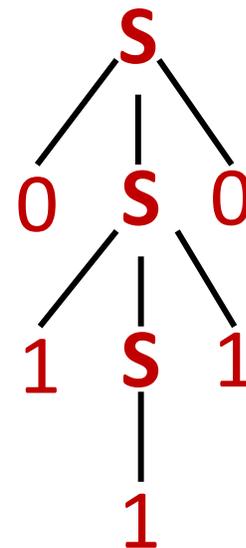
Recall: Parse Trees

Suppose that grammar G generates a string x

- A *parse tree* of x for G has
 - Root labeled S (start symbol of G)
 - The children of any node labeled A are labeled by symbols of w left-to-right for some rule $A \rightarrow w$
 - The symbols of x label the leaves ordered left-to-right

$$S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$$

Parse tree of 01110

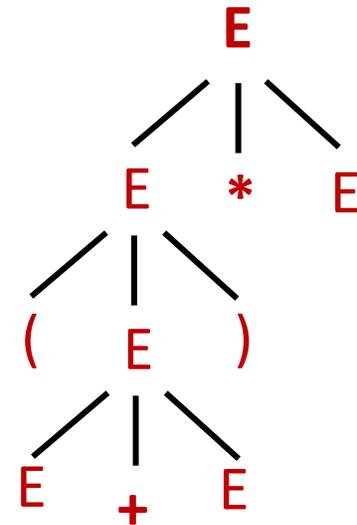


Simple Arithmetic Expressions

$E \rightarrow E + E \mid E * E \mid (E) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4$
 $\mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Generate $(2 + x) * y$

$E \rightarrow E * E$
 $\rightarrow (E) * E$
 $\rightarrow (E + E) * E$

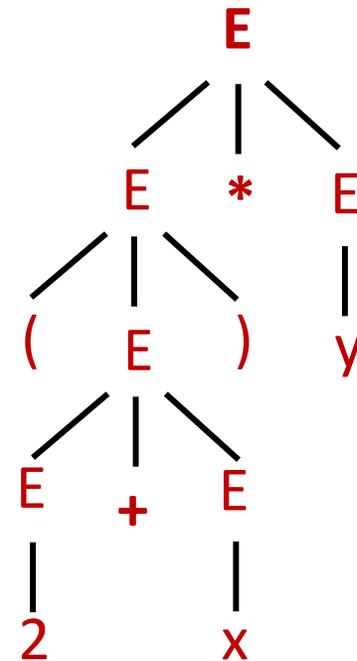


Simple Arithmetic Expressions

$E \rightarrow E + E \mid E * E \mid (E) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4$
 $\mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Generate $(2 + x) * y$

$E \rightarrow E * E$
 $\rightarrow (E) * E$
 $\rightarrow (E + E) * E$
 $\rightarrow (2 + E) * E$
 $\rightarrow (2 + x) * E$
 $\rightarrow (2 + x) * y$

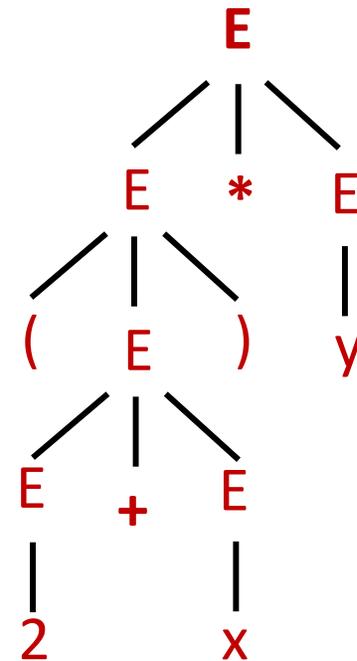


Simple Arithmetic Expressions

$E \rightarrow E + E \mid E * E \mid (E) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4$
 $\mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Generate $(2 + x) * y$

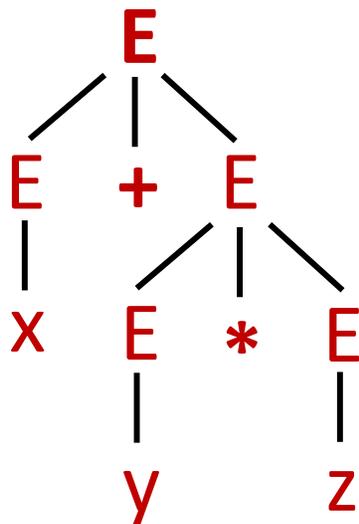
$E \rightarrow E * E$
 $\rightarrow (E) * E$
 $\rightarrow (E + E) * E$ or...
 $\rightarrow (2 + E) * E$ $\rightarrow (E + E) * y$
 $\rightarrow (2 + x) * E$ $\rightarrow (E + x) * y$
 $\rightarrow (2 + x) * y$ $\rightarrow (2 + x) * y$



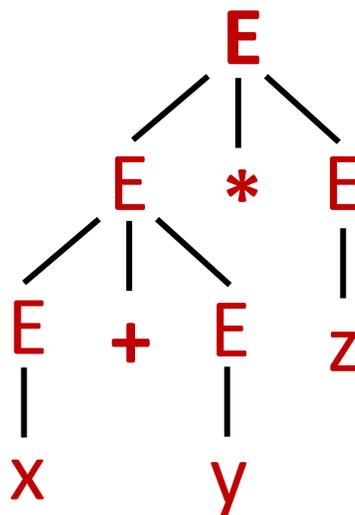
Simple Arithmetic Expressions

$E \rightarrow E + E \mid E * E \mid (E) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4$
 $\mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Generate $x+y*z$ in ways that give two *different* parse trees



$E \Rightarrow E + E \Rightarrow x + E \Rightarrow x + E * E \Rightarrow x + y * E \Rightarrow x + y * z$
(multiply y with z and then add to x)



$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow x + E * E$
 $\Rightarrow x + y * E \Rightarrow x + y * z$
(add x to y , then multiply by z)

building precedence in simple arithmetic expressions

- **E** – expression (start symbol)
- **T** – term **F** – factor **I** – identifier **N** - number

E → **T** | **E+T**

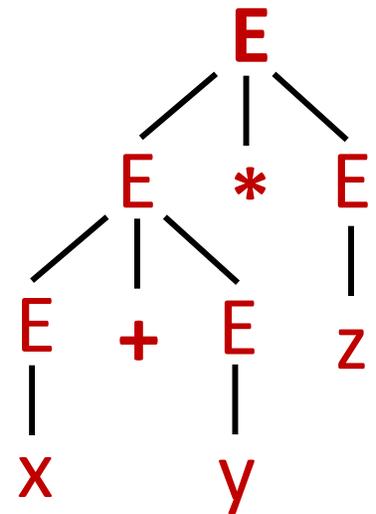
T → **F** | **F*T**

F → (**E**) | **I** | **N**

I → **x** | **y** | **z**

N → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

No longer
allows:



building precedence in simple arithmetic expressions

- **E** – expression (start symbol)
- **T** – term **F** – factor **I** – identifier **N** - number

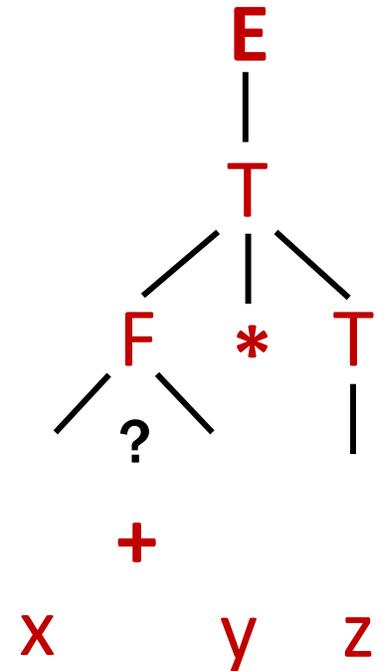
E → **T** | **E+T**

T → **F** | **F*T**

F → (**E**) | **I** | **N**

I → **x** | **y** | **z**

N → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**



building precedence in simple arithmetic expressions

- **E** – expression (start symbol)
- **T** – term **F** – factor **I** – identifier **N** - number

E → **T** | **E+T**

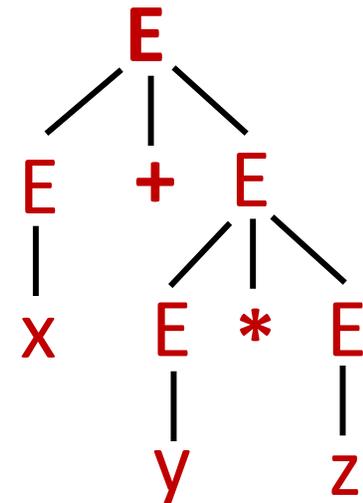
T → **F** | **F*T**

F → (**E**) | **I** | **N**

I → **x** | **y** | **z**

N → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Still
allows:



building precedence in simple arithmetic expressions

- **E** – expression (start symbol)
- **T** – term **F** – factor **I** – identifier **N** - number

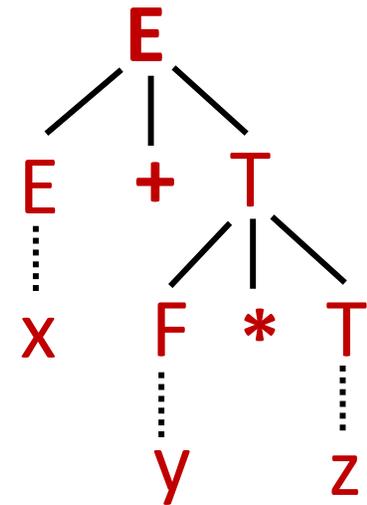
E → **T** | **E+T**

T → **F** | **F*T**

F → (**E**) | **I** | **N**

I → **x** | **y** | **z**

N → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**



Backus-Naur Form (The same thing...)

BNF (Backus-Naur Form) grammars

- Originally used to define programming languages
- Variables denoted by long names in angle brackets, e.g.
 - <identifier>, <if-then-else-statement>,
<assignment-statement>, <condition>
 - ::= used instead of \rightarrow

BNF for C

```
statement:
  ((identifier | "case" constant-expression | "default") ":")*
  (expression? ";" |
  block |
  "if" "(" expression ")" statement |
  "if" "(" expression ")" statement "else" statement |
  "switch" "(" expression ")" statement |
  "while" "(" expression ")" statement |
  "do" statement "while" "(" expression ")" ";" |
  "for" "(" expression? ";" expression? ";" expression? ")" statement |
  "goto" identifier ";" |
  "continue" ";" |
  "break" ";" |
  "return" expression? ";"
  )

block: "{" declaration* statement* "}"

expression:
  assignment-expression%

assignment-expression: (
  unary-expression (
    "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" | ">>=" | "&=" |
    "^=" | "|="
  )
  )
  )* conditional-expression

conditional-expression:
  logical-OR-expression ( "?" expression ":" conditional-expression )?
```

BNF for (Simple) English

Back to middle school:

<sentence> ::= <noun phrase> <verb phrase>

<noun phrase> ::= <article> <adjective> <noun>

<verb phrase> ::= <verb> <adverb> | <verb> <object>

<object> ::= <noun phrase>

Parse:

The yellow duck squeaked loudly

The red truck hit a parked car

Applications of Set Theory

Recall: Sets

Sets are collections of objects called **elements**.

Write $a \in B$ to say that a is an element of set B ,
and $a \notin B$ to say that it is not.

Some simple examples

$$A = \{1\}$$

$$B = \{1, 3, 2\}$$

$$C = \{\square, 1\}$$

$$D = \{\{17\}, 17\}$$

$$E = \{1, 2, 7, \text{cat}, \text{dog}, \emptyset, \alpha\}$$

Representing Sets Using Bits

- Suppose universe U is $\{1, 2, \dots, n\}$
- Can represent set $B \subseteq U$ as a vector of bits:

$$b_n \dots b_2 b_1 \text{ where } \begin{array}{l} b_i = 1 \text{ when } i \in B \\ b_i = 0 \text{ when } i \notin B \end{array}$$

- Called the *characteristic vector* of set B
- With $n = 5$:
 - the characteristic vector for $\{1, 2, 5\}$ is **10011**
 - what is characteristic vector for $\{2, 3, 5\}$?
 - characteristic vectors are and **10110**

Representing Sets Using Bits

- Suppose universe U is $\{1, 2, \dots, n\}$
- Can represent set $B \subseteq U$ as a vector of bits:
 $b_1 b_2 \dots b_n$ where $b_i = 1$ when $i \in B$
 $b_i = 0$ when $i \notin B$
 - Called the *characteristic vector* of set B
- With $n = 5$, $A = \{1, 2, 5\}$, and $B = \{2, 3, 5\}$,
 - characteristic vectors are **10011** and **10110**
 - what is characteristic vector for $A \cup B$? $A \cap B$?
 - characteristic vectors are **10111** and **10010**

Bitwise Operations

$$\begin{array}{r} 01101101 \\ \vee \quad 00110111 \\ \hline 01111111 \end{array}$$

Java: $z = x | y$

$$\begin{array}{r} 00101010 \\ \wedge \quad 00001111 \\ \hline 00001010 \end{array}$$

Java: $z = x \& y$

$$\begin{array}{r} 01101101 \\ \oplus \quad 00110111 \\ \hline 01011010 \end{array}$$

Java: $z = x \wedge y$

A Useful Identity

- If x and y are bits: $(x \oplus y) \oplus y = ?$

x	y	$x \oplus y$	$(x \oplus y) \oplus y$
0	0		
0	1		
1	0		
1	1		

A Useful Identity

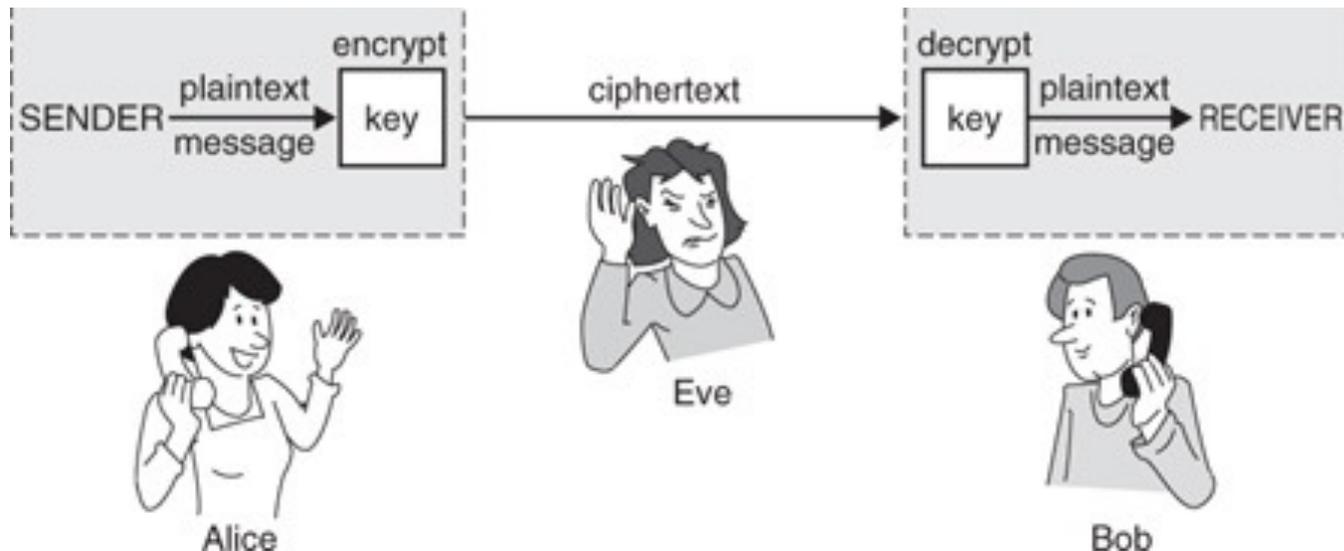
- If x and y are bits: $(x \oplus y) \oplus y = ?$

x	y	$x \oplus y$	$(x \oplus y) \oplus y$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

- What if x and y are **bit-vectors**?

Private Key Cryptography

- **Alice** wants to communicate message secretly to **Bob** so that eavesdropper **Eve** who hears their conversation cannot tell what **Alice's** message is.
- **Alice** and **Bob** can get together and privately share a secret key **K** ahead of time.



Recall: Relations

Let A and B be sets,

A **binary relation from A to B** is a subset of $A \times B$

n-ary Relations

Let A_1, A_2, \dots, A_n be sets. An ***n*-ary** relation on these sets is a subset of $A_1 \times A_2 \times \dots \times A_n$.

Relational Databases

STUDENT

Student_Name	ID_Number	Office	GPA
Knuth	328012098	022	4.00
Von Neuman	481080220	555	3.78
Russell	238082388	022	3.85
Einstein	238001920	022	2.11
Newton	1727017	333	3.61
Karp	348882811	022	3.98
Bernoulli	2921938	022	3.21

Database Operations: Projection

Find all offices: $\Pi_{\text{Office}}(\text{STUDENT})$

Office
022
555
333

Find offices and GPAs: $\Pi_{\text{Office,GPA}}(\text{STUDENT})$

Office	GPA
022	4.00
555	3.78
022	3.85
022	2.11
333	3.61
022	3.98
022	3.21

Database Operations: Selection

Find students with GPA > 3.9 : $\sigma_{\text{GPA}>3.9}(\text{STUDENT})$

Student_Name	ID_Number	Office	GPA
Knuth	328012098	022	4.00
Karp	348882811	022	3.98

Retrieve the name and GPA for students with GPA > 3.9:

$\Pi_{\text{Student_Name,GPA}}(\sigma_{\text{GPA}>3.9}(\text{STUDENT}))$

Student_Name	GPA
Knuth	4.00
Karp	3.98

Relational Databases

STUDENT

Student_Name	ID_Number	Office	GPA	Course
Knuth	328012098	022	4.00	CSE311
Knuth	328012098	022	4.00	CSE351
Von Neuman	481080220	555	3.78	CSE311
Russell	238082388	022	3.85	CSE312
Russell	238082388	022	3.85	CSE344
Russell	238082388	022	3.85	CSE351
Newton	1727017	333	3.61	CSE312
Karp	348882811	022	3.98	CSE311
Karp	348882811	022	3.98	CSE312
Karp	348882811	022	3.98	CSE344
Karp	348882811	022	3.98	CSE351
Bernoulli	2921938	022	3.21	CSE351

What's not so nice?

Relational Databases

STUDENT

Student_Name	ID_Number	Office	GPA
Knuth	328012098	022	4.00
Von Neuman	481080220	555	3.78
Russell	238082388	022	3.85
Einstein	238001920	022	2.11
Newton	1727017	333	3.61
Karp	348882811	022	3.98
Bernoulli	2921938	022	3.21

TAKES

ID_Number	Course
328012098	CSE311
328012098	CSE351
481080220	CSE311
238082388	CSE312
238082388	CSE344
238082388	CSE351
1727017	CSE312
348882811	CSE311
348882811	CSE312
348882811	CSE344
348882811	CSE351
2921938	CSE351

Better

Database Operations: Natural Join

Student ⋈ Takes

Student_Name	ID_Number	Office	GPA	Course
Knuth	328012098	022	4.00	CSE311
Knuth	328012098	022	4.00	CSE351
Von Neuman	481080220	555	3.78	CSE311
Russell	238082388	022	3.85	CSE312
Russell	238082388	022	3.85	CSE344
Russell	238082388	022	3.85	CSE351
Newton	1727017	333	3.61	CSE312
Karp	348882811	022	3.98	CSE311
Karp	348882811	022	3.98	CSE312
Karp	348882811	022	3.98	CSE344
Karp	348882811	022	3.98	CSE351
Bernoulli	2921938	022	3.21	CSE351

Relational Databases

- **Store data as a set of relations**
- **Implements relational operators**
 - written in a higher-level language (e.g., SQL)
 - compiled into relational algebra expressions
- **Not hard to implement across multiple servers:**
 - add a "shuffle" operation that repartitions data
- **Not hard to implement more complex data types**

Applications of Number Theory

Recall: Fibonacci Numbers

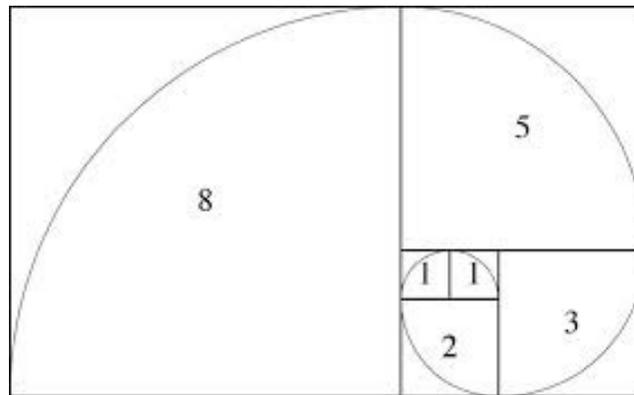
$$f_0 := 0$$

$$f_1 := 1$$

$$f_{n+2} := f_{n+1} + f_n$$

$$f(0) := 0 \text{ and } f(1) := 1$$

$$f(n+2) := f(n+1) + f(n)$$



Fibonacci Numbers

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2} \text{ for all } n \geq 2$$



Tamás Görbe
@TamasGorbe

A Mathematician's Way* of Converting Miles to
Kilometers

$$3 \text{ mi} \approx 5 \text{ km}$$

$$5 \text{ mi} \approx 8 \text{ km}$$

$$8 \text{ mi} \approx 13 \text{ km}$$

$$f_n \text{ mi} \approx f_{n+1} \text{ km}$$

Recall: Bounding Fibonacci I: $f_n < 2^n$ for all $n \geq 0$

1. Let $P(n)$ be " $f_n < 2^n$ ". We prove that $P(n)$ is true for all integers $n \geq 0$ by strong induction.
2. Base Cases: $f_0 = 0 < 1 = 2^0$ so $P(0)$ is true and $f_1 = 1 < 2 = 2^1$ so $P(1)$ is true.
3. Inductive Hypothesis: Assume that for some arbitrary integer $k \geq 1$, we have $f_j < 2^j$ for every integer j from 0 to k .
4. Inductive Step: We can calculate that

$$\begin{aligned} f_{k+1} &= f_k + f_{k-1} && \text{def of } f \text{ (since } k+1 \geq 2\text{)} \\ &< 2^k + 2^{k-1} && \text{by IH (since } k-1 \geq 0\text{)} \\ &< 2^k + 2^k \\ &= 2^{k+1} \end{aligned}$$

so $P(k+1)$ is true.

5. Therefore, by strong induction, $f_n < 2^n$ for all integers $n \geq 0$.

Bounding Fibonacci II: $f_n \geq 2^{n/2} - 1$ for all $n \geq 2$

1. Let $P(n)$ be “ $f_n \geq 2^{n/2} - 1$ ”. We prove that $P(n)$ is true for all integers $n \geq 2$ by strong induction.

2. Base Cases: $f_2 = f_1 + f_0 = 1 \geq 1 = 2^0 = 2^{2/2} - 1$ so $P(2)$ holds
 $f_3 = f_2 + f_1 = 2 \geq 2^{1/2} = 2^{3/2} - 1$ so $P(3)$ holds

3. Inductive Hypothesis: Assume that for some arbitrary integer $k \geq 3$, $P(j)$ is true for every integer j from 2 to k .

4. Inductive Step: We can calculate

$$\begin{aligned} f_{k+1} &= f_k + f_{k-1} && \text{def of } f \text{ (since } k+1 \geq 4) \\ &\geq 2^{k/2-1} + f_{k-1} && \text{by the IH} \\ &\geq 2^{k/2-1} + 2^{(k-1)/2-1} && \text{by the IH (since } k-1 \geq 2) \\ &\geq 2 \cdot 2^{(k-1)/2-1} = 2^{(k+1)/2-1} \end{aligned}$$

so $P(k+1)$ is true.

5. Therefore by strong induction, $f_n \geq 2^{n/2} - 1$ for all integers $n \geq 2$.

Running time of Euclid's algorithm

Theorem: Suppose that Euclid's Algorithm takes n steps for $\gcd(a, b)$ with $a \geq b > 0$. Then, $a \geq f_{n+1}$.

Why does this help us bound the running time of Euclid's Algorithm?

Running time of Euclid's algorithm

Theorem: Suppose that Euclid's Algorithm takes n steps for $\gcd(a, b)$ with $a \geq b > 0$. Then, $a \geq f_{n+1}$.

Why does this help us bound the running time of Euclid's Algorithm?

We already proved that $f_n \geq 2^{n/2} - 1$ so $f_{n+1} \geq 2^{(n+1)/2}$

Therefore, if Euclid's Algorithm takes n steps for $\gcd(a, b)$ with $a \geq b > 0$ then $a \geq 2^{(n+1)/2}$

so $\log_2 a \geq (n + 1)/2$ or $n \leq 2 \log_2 a - 1$
i.e., # of steps $<$ twice the # of bits in a .

Recall: Euclid's Algorithm

$$\text{gcd}(a, b) = \text{gcd}(b, a \bmod b)$$

$$\text{gcd}(a, 0) = a$$

```
int gcd(int a, int b) { /* Assumes: a >= b >= 0 */
    if (b == 0) {
        return a;
    } else {
        return gcd(b, a % b);
    }
}
```

Note: $\text{gcd}(b, a) = \text{gcd}(a, b)$

Recall: Euclid's Algorithm

Repeatedly use $\gcd(a, b) = \gcd(b, a \bmod b)$ to reduce numbers until you get $\gcd(g, 0) = g$.

$$\begin{aligned} & \gcd(660, 126) \\ = & \gcd(126, 660 \bmod 126) & = \gcd(126, 30) \\ = & \gcd(30, 126 \bmod 30) & = \gcd(30, 6) \\ = & \gcd(6, 30 \bmod 6) & = \gcd(6, 0) \\ & & = 6 \end{aligned}$$

Recall: Euclid's Algorithm

Repeatedly use $\gcd(a, b) = \gcd(b, a \bmod b)$ to reduce numbers until you get $\gcd(g, 0) = g$.

At each step, we calculate "mod" with the division theorem:

$$\begin{aligned} & \gcd(660, 126) & 660 &= 5 \cdot 126 + 30 \\ &= \gcd(126, 30) & 126 &= 4 \cdot 30 + 6 \\ &= \gcd(30, 6) & 30 &= 5 \cdot 6 \\ &= \gcd(6, 0) = 6 \end{aligned}$$

Recall: Euclid's Algorithm

Repeatedly use $\gcd(a, b) = \gcd(b, a \bmod b)$ to reduce numbers until you get $\gcd(g, 0) = g$.

At each step, we calculate "mod" with the division theorem:

$$660 = 5 \cdot 126 + 30$$

$$126 = 4 \cdot 30 + 6$$

$$30 = 5 \cdot 6$$

$$r_4 = 660$$

$$r_3 = 126$$

$$r_2 = 30$$

$$r_1 = 6$$

$$r_4 = 5 \cdot r_3 + r_2$$

$$r_3 = 4 \cdot r_2 + r_1$$

$$r_2 = 5 \cdot r_1$$

Running time of Euclid's algorithm

Theorem: Suppose that Euclid's Algorithm takes n steps for $\gcd(a, b)$ with $a \geq b > 0$. Then, $a \geq f_{n+1}$.

An informal way to get the idea: Consider an n step gcd calculation starting with $r_{n+1} = a$ and $r_n = b$:

$$r_{n+1} = q_n r_n + r_{n-1}$$

$$r_n = q_{n-1} r_{n-1} + r_{n-2}$$

...

$$r_3 = q_2 r_2 + r_1$$

$$r_2 = q_1 r_1$$

For all $k \geq 2$, $r_{k-1} = r_{k+1} \bmod r_k$

Now $r_1 \geq 1$ and each q_k must be ≥ 1 . If we replace all the q_k 's by 1 and replace r_1 by 1, we can only reduce the r_k 's. After that reduction, $r_k = f_k$ for every k .

Running time of Euclid's algorithm

Theorem: Suppose that Euclid's Algorithm takes n steps for $\gcd(a, b)$ with $a \geq b > 0$. Then, $a \geq f_{n+1}$.

We go by strong induction on n .

Let $P(n)$ be “ $\gcd(a, b)$ with $a \geq b > 0$ takes n steps $\rightarrow a \geq f_{n+1}$ ” for all $n \geq 1$.

Base Case: $n=1$ Suppose Euclid's Algorithm with $a \geq b > 0$ takes 1 step.

Running time of Euclid's algorithm

Theorem: Suppose that Euclid's Algorithm takes n steps for $\gcd(a, b)$ with $a \geq b > 0$. Then, $a \geq f_{n+1}$.

We go by strong induction on n .

Let $P(n)$ be “ $\gcd(a, b)$ with $a \geq b > 0$ takes n steps $\rightarrow a \geq f_{n+1}$ ” for all $n \geq 1$.

Base Case: $n=1$ Suppose Euclid's Algorithm with $a \geq b > 0$ takes 1 step.
By assumption, $a \geq b \geq 1 = f_2$ so $P(1)$ holds.

Base Case: $n=2$ Suppose Euclid's Algorithm with $a \geq b > 0$ takes 2 steps.

Running time of Euclid's algorithm

Theorem: Suppose that Euclid's Algorithm takes n steps for $\gcd(a, b)$ with $a \geq b > 0$. Then, $a \geq f_{n+1}$.

We go by strong induction on n .

Let $P(n)$ be “ $\gcd(a, b)$ with $a \geq b > 0$ takes n steps $\rightarrow a \geq f_{n+1}$ ” for all $n \geq 1$.

Base Case: $n=1$ Suppose Euclid's Algorithm with $a \geq b > 0$ takes 1 step.
By assumption, $a \geq b \geq 1 = f_2$ so $P(1)$ holds.

Base Case: $n=2$ Suppose Euclid's Algorithm with $a \geq b > 0$ takes 2 steps.
By assumption, $a \geq b \geq 2 = f_3$ so $P(2)$ holds.

Induction Hypothesis: Suppose that for some integer $k \geq 2$, $P(j)$ is true for all integers j s.t. $1 \leq j \leq k$

Running time of Euclid's algorithm

Theorem: Suppose that Euclid's Algorithm takes n steps for $\gcd(a, b)$ with $a \geq b > 0$. Then, $a \geq f_{n+1}$.

We go by strong induction on n .

Let $P(n)$ be “ $\gcd(a, b)$ with $a \geq b > 0$ takes n steps $\rightarrow a \geq f_{n+1}$ ” for all $n \geq 1$.

Base Case: $n=1$ Suppose Euclid's Algorithm with $a \geq b > 0$ takes 1 step. By assumption, $a \geq b \geq 1 = f_2$ so $P(1)$ holds.

Base Case: $n=2$ Suppose Euclid's Algorithm with $a \geq b > 0$ takes 2 steps. By assumption, $a \geq b \geq 2 = f_3$ so $P(1)$ holds.

Induction Hypothesis: Suppose that for some integer $k \geq 2$, $P(j)$ is true for all integers j s.t. $1 \leq j \leq k$

Inductive Step: We want to show: if $\gcd(a, b)$ with $a \geq b > 0$ takes $k+1$ steps, then $a \geq f_{k+2}$.

Running time of Euclid's algorithm

Induction Hypothesis: Suppose that for some integer $k \geq 1$, $P(j)$ is true for all integers j s.t. $1 \leq j \leq k$

Inductive Step: **Goal: if $\gcd(a,b)$ with $a \geq b > 0$ takes $k+1$ steps, then $a \geq f_{k+2}$.**

Next suppose that $k+1 \geq 3$ so for the first 3 steps of Euclid's algorithm on a and b we have

$$a = q_{k+1} b + r_k$$

$$b = q_k r_k + r_{k-1}$$

$$r_k = q_{k-1} r_{k-1} + r_{k-2}$$

and there are $k-2$ more steps after this.

Running time of Euclid's algorithm

Induction Hypothesis: Suppose that for some integer $k \geq 1$, $P(j)$ is true for all integers j s.t. $1 \leq j \leq k$

Inductive Step: Goal: if $\gcd(a,b)$ with $a \geq b > 0$ takes $k+1$ steps, then $a \geq f_{k+2}$.

Next suppose that $k+1 \geq 3$ so for the first 3 steps of Euclid's algorithm on a and b we have

$$a = q_{k+1} b + r_k$$

$$b = q_k r_k + r_{k-1}$$

$$r_k = q_{k-1} r_{k-1} + r_{k-2}$$

and there are $k-2$ more steps after this. Note that this means that the $\gcd(b, r_k)$ takes k steps and $\gcd(r_k, r_{k-1})$ takes $k-1$ steps.

So since $k \geq 1$ and $k-1 \geq 1$, by the IH, we have $b \geq f_{k+1}$ and $r_k \geq f_k$.

Running time of Euclid's algorithm

Induction Hypothesis: Suppose that for some integer $k \geq 1$, $P(j)$ is true for all integers j s.t. $1 \leq j \leq k$

Inductive Step: Goal: if $\gcd(a,b)$ with $a \geq b > 0$ takes $k+1$ steps, then $a \geq f_{k+2}$.

Next suppose that $k+1 \geq 3$ so for the first 3 steps of Euclid's algorithm on a and b we have

$$a = q_{k+1} b + r_k$$

$$b = q_k r_k + r_{k-1}$$

$$r_k = q_{k-1} r_{k-1} + r_{k-2}$$

and there are $k-2$ more steps after this. Note that this means that the $\gcd(b, r_k)$ takes k steps and $\gcd(r_k, r_{k-1})$ takes $k-1$ steps.

So since $k \geq 1$ and $k-1 \geq 1$, by the IH, we have $b \geq f_{k+1}$ and $r_k \geq f_k$.

Also, since $a \geq b$, we must have $q_{k+1} \geq 1$.

So $a = q_{k+1} b + r_k \geq b + r_k \geq f_{k+1} + f_k = f_{k+2}$ as required.

Algorithmic Problems

- **Multiplication**

- Given primes p_1, p_2, \dots, p_k , calculate their product $p_1 p_2 \dots p_k$

- **Factoring**

- Given an integer n , determine the prime factorization of n

Factoring

Factor the following 232 digit number [RSA768]:

123018668453011775513049495838496272077
285356959533479219732245215172640050726
365751874520219978646938995647494277406
384592519255732630345373154826850791702
612214291346167042921431160222124047927
4737794080665351419597459856902143413

12301866845301177551304949583849627207728535695953347
92197322452151726400507263657518745202199786469389956
47494277406384592519255732630345373154826850791702612
21429134616704292143116022212404792747377940806653514
19597459856902143413

=

334780716989568987860441698482126908177047949837
137685689124313889828837938780022876147116525317
43087737814467999489

×

367460436667995904282446337996279526322791581643
430876426760322838157396665112792333734171433968
10270092798736308917

Famous Algorithmic Problems

- **Factoring**
 - Given an integer n , determine the prime factorization of n
- **Primality Testing**
 - Given an integer n , determine if n is prime
- **Factoring is hard**
 - (on a classical computer)
- **Primality Testing is easy**

GCD and Factoring

$$a = 2^3 \cdot 3 \cdot 5^2 \cdot 7 \cdot 11 = 46,200$$

$$b = 2 \cdot 3^2 \cdot 5^3 \cdot 7 \cdot 13 = 204,750$$

$$\text{GCD}(a, b) = 2^{\min(3,1)} \cdot 3^{\min(1,2)} \cdot 5^{\min(2,3)} \cdot 7^{\min(1,1)} \cdot 11^{\min(1,0)} \cdot 13^{\min(0,1)}$$

Factoring is hard

Yet, we can compute **GCD(a,b)** without factoring!

Using Fast Modular Exponentiation

- E-commerce web transactions use SSL, which is based on RSA encryption
- RSA:
 - Vendor chooses random 512-bit or 1024-bit primes p, q and 512/1024-bit exponent e . Computes $m = p \cdot q$
 - Vendor broadcasts (m, e)
 - To send a to vendor, you compute $C = a^e \bmod m$ and send C to the vendor.
 - Using secret p, q the vendor computes d that is the *multiplicative inverse* of $e \bmod (p - 1)(q - 1)$.
 - Vendor computes $C^d \bmod m = (a^e)^d \bmod m = a^{ed} \bmod m$
 - HW3 EC implies $a^{ed} \bmod m = a \bmod m$
 - Assumes we can calculate exponentials quickly!

Exponentiation

- **Compute** 78365^{81453}
- **Compute** $78365^{81453} \bmod 104729$
- **Output is small**
 - need to keep intermediate results small

Small Multiplications

Since $\mathbf{b} = qm + (\mathbf{b} \bmod m)$, we have $\mathbf{b} \bmod m \equiv_m \mathbf{b}$.

And since $\mathbf{c} = tm + (\mathbf{c} \bmod m)$, we have $\mathbf{c} \bmod m \equiv_m \mathbf{c}$.

Multiplying these gives $(\mathbf{b} \bmod m)(\mathbf{c} \bmod m) \equiv_m \mathbf{bc}$.

By a Lemma from Topic 3, this tells us that

$$\mathbf{bc} \bmod m = (\mathbf{b} \bmod m)(\mathbf{c} \bmod m) \bmod m.$$

Okay to mod \mathbf{b} and \mathbf{c} by m before multiplying if we are planning to mod the result by m

Repeated Squaring – small and fast

Since $b \bmod m \equiv_m b$ and $c \bmod m \equiv_m c$

we have $bc \bmod m = (b \bmod m)(c \bmod m) \bmod m$

So $a^2 \bmod m = (a \bmod m)^2 \bmod m$

and $a^4 \bmod m = (a^2 \bmod m)^2 \bmod m$

and $a^8 \bmod m = (a^4 \bmod m)^2 \bmod m$

and $a^{16} \bmod m = (a^8 \bmod m)^2 \bmod m$

and $a^{32} \bmod m = (a^{16} \bmod m)^2 \bmod m$

Can compute $a^k \bmod m$ for $k = 2^i$ in only i steps

What if k is not a power of 2?

Fast Exponentiation Algorithm

81453 in binary is 10011111000101101

$$81453 = 2^{16} + 2^{13} + 2^{12} + 2^{11} + 2^{10} + 2^9 + 2^5 + 2^3 + 2^2 + 2^0$$

$$a^{81453} = a^{2^{16}} \cdot a^{2^{13}} \cdot a^{2^{12}} \cdot a^{2^{11}} \cdot a^{2^{10}} \cdot a^{2^9} \cdot a^{2^5} \cdot a^{2^3} \cdot a^{2^2} \cdot a^{2^0}$$

$$a^{81453} \bmod m =$$

$$\begin{aligned} & (\dots(((((a^{2^{16}} \bmod m \cdot \\ & \quad a^{2^{13}} \bmod m) \bmod m \cdot \\ & \quad a^{2^{12}} \bmod m) \bmod m \cdot \\ & \quad a^{2^{11}} \bmod m) \bmod m \cdot \\ & \quad a^{2^{10}} \bmod m) \bmod m \cdot \\ & \quad a^{2^9} \bmod m) \bmod m \cdot \\ & \quad a^{2^5} \bmod m) \bmod m \cdot \\ & \quad a^{2^3} \bmod m) \bmod m \cdot \\ & \quad a^{2^2} \bmod m) \bmod m \cdot \\ & \quad a^{2^0} \bmod m) \bmod m \end{aligned}$$

Uses only $16 + 9 = 25$
multiplications

The fast exponentiation algorithm computes

$a^k \bmod m$ using $\leq 2 \log k$ multiplications $\bmod m$

Fast Exponentiation

```
public static int FastModExp(int a, int k, int modulus) {  
  
    if (k == 0) {  
        return 1;  
  
    } else if ((k % 2) == 0) {  
        long temp = FastModExp(a,k/2,modulus);  
        return (temp * temp) % modulus;  
  
    } else {  
        long temp = FastModExp(a,k-1,modulus);  
        return (a * temp) % modulus;  
    }  
}
```

$$a^{2j} \bmod m = (a^j \bmod m)^2 \bmod m$$

$$a^{2j+1} \bmod m = ((a \bmod m) \cdot (a^{2j} \bmod m)) \bmod m$$

Basic Applications of mod

- Hashing
- Pseudo random number generation
- Two's Complement

Hashing

Scenario:

Map a small number of data values from a large domain $\{0, 1, \dots, M - 1\}$...

...into a small set of locations $\{0, 1, \dots, n - 1\}$ so one can quickly check if some value is present

- $\text{hash}(x) = x \bmod p$ for p a prime close to n
 - or $\text{hash}(x) = (ax + c) \bmod p$
- Latter depends on all the bits of the data
 - $\text{hash}(x)$ and $\text{hash}(x + 1)$ can be very far apart

Hashing

- $\text{hash}(x) = (ax + c) \bmod p$ for prime p
 - deterministic function with random-ish behavior

- Suppose that $\text{hash}(x) = \text{hash}(y) \dots$

$$ax + c \equiv_p ay + c$$

$$ax \equiv_p ay$$

$$x \equiv_p y$$

add $-c$ to both sides

multiply both sides by s

where $as \equiv_p 1$

- Output as evenly spread as $\text{hash}(x) = x \bmod p$

Hashing

- $\text{hash}(x) = (ax + c) \bmod p$ for prime p
 - deterministic function with random-ish behavior
- Applications
 - map integer to location in array (hash tables)
 - map user ID or IP address to machine
 - requests from the same user / IP address go to the same machine
 - requests from different users / IP addresses spread randomly

Pseudo-Random Number Generation

Linear Congruential method

$$x_{n+1} = (a x_n + c) \bmod m$$

Choose random x_0, a, c, m and produce a long sequence of x_n 's

Recall: I'm ALIVE!

```
public class Test {
    final static int SEC_IN_YEAR = 365*24*60*60;
    public static void main(String args[]) {
        System.out.println(
            "I will be alive for at least " +
            SEC_IN_YEAR * 101 + " seconds."
        );
    }
}
```

```
----jGRASP exec: java Test
I will be alive for at least -186619904 seconds.
----jGRASP: operation complete.
```

n-bit Unsigned Integer Representation

- Represent integer x as sum of powers of 2:

$$99 = 64 + 32 + 2 + 1 = 2^6 + 2^5 + 2^1 + 2^0$$

$$18 = 16 + 2 = 2^4 + 2^1$$

- Binary representation shows which powers are used:

99: 0110 0011

18: 0001 0010

n-bit Unsigned Integer Representation

- Suppose we write numbers with 4 bits:

$$14 = 8 + 4 + 2 = 2^3 + 2^2 + 2^1 = 1110$$

$$11 = 8 + 2 + 1 = 2^3 + 2^1 + 2^0 = 1011$$

- Largest number we can write in 4 bits is:

$$15 = 8 + 4 + 2 + 1 = 2^3 + 2^2 + 2^1 + 2^0 = 1111$$

- Note that $15 = 16 - 1 = 2^4 - 1$
 - we proved this before!

n-bit Unsigned Integer Representation

- Suppose we write numbers with 4 bits (0 .. 15):

$$14 = 8 + 4 + 2 = 2^3 + 2^2 + 2^1 = 1110$$

$$11 = 8 + 2 + 1 = 2^3 + 2^1 + 2^0 = 1011$$

- Adding these numbers gives us 25 with 5 bits:

$$25 = 16 + 8 + 1 = 2^4 + 2^3 + 2^0 = 11001$$

- If we drop the highest bit, we have

$$9 = 8 + 1 = 2^3 + 2^0 = 1001$$

n-bit Unsigned Integer Representation

$$\begin{array}{rclcl} 25 & = & 16 + 8 + 1 & = & 2^4 + 2^3 + 2^0 & = & 11001 \\ 9 & = & 8 + 1 & = & 2^3 + 2^0 & = & 1001 \end{array}$$

- Note that $9 \equiv_{16} 25$ since $25 - 9 = 16$
 - dropping 2^4 bit subtracts 16
 - dropping 2^5 bit subtracts $32 = 2 \cdot 16$
 - dropping 2^6 bit subtracts $64 = 4 \cdot 16$
- Throwing away all but 4 bits is arithmetic mod 16
 - easier to implement normal arithmetic!

Sign-Magnitude Integer Representation

n-bit signed integers

Suppose that $-2^{n-1} < x < 2^{n-1}$

First bit as the sign, $n - 1$ bits for the value

$$99 = 64 + 32 + 2 + 1$$

$$18 = 16 + 2$$

For $n = 8$:

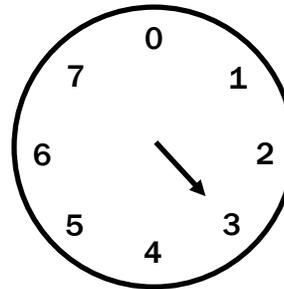
99: 0110 0011

-18: 1001 0010

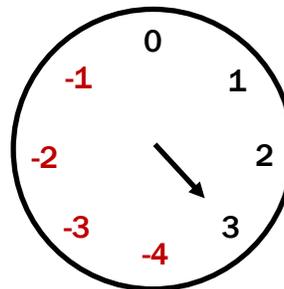
Problem: this has both +0 and -0 (annoying)

Arithmetic on a Clock

3 bits, unsigned



3 bits, signed



Since $-1 \equiv_8 7$, arithmetic is unchanged

Only differences are printing and comparison

Two's Complement Representation

Suppose that $0 \leq x < 2^{n-1}$

x is represented by the binary representation of x

Suppose that $-2^{n-1} \leq x < 0$

x is represented by the binary representation of $x + 2^n$

result is in the range $2^{n-1} \leq x < 2^n$

$$99 = 64 + 32 + 2 + 1$$

$$18 = 16 + 2$$

For $n = 8$:

$$99: \quad 0110\ 0011$$

$$-18: \quad 1110\ 1110$$

$$(-18 + 256 = 238)$$

Two's Complement Representation

Suppose that $0 \leq x < 2^{n-1}$

x is represented by the binary representation of x

Suppose that $-2^{n-1} \leq x < 0$

x is represented by the binary representation of $x + 2^n$

result is in the range $2^{n-1} \leq x < 2^n$

With 4 bits:

0	1	2	3	4	5	6	7	-8	-7	-6	-5	-4	-3	-2	-1
0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Key property: First bit is still the sign bit!

Key property: Twos complement representation of any number y is equivalent to $y \bmod 2^n$ so arithmetic works **mod** 2^n

$$y + 2^n \equiv_{2^n} y$$

Two's Complement Representation

- For $0 < x \leq 2^{n-1}$, $-x$ is represented by the binary representation of $-x + 2^n$
 - How do we calculate $-x$ from x ?
 - E.g., what happens for “return $-x$;” in Java?

$$-x + 2^n = (2^n - 1) - x + 1$$

- To compute this, flip the bits of x then add 1!
Flip the bits of x means replace x by $2^n - 1 - x$
Then add 1 to get $-x + 2^n$

Applications of Formal Logic

Satisfiability (SAT)

Problem: Given a Propositional Logic expression, is there a way to set the values of the variables to make the expression evaluate to T?

- if yes, the expression is "satisfiable"
 - if not, the expression is "unsatisfiable"
-
- Many problems can be stated as SAT problems
 - e.g., many "puzzle" type problems
see HW1 for an example
 - lots of important & useful problems in this category
e.g., verifying correctness of hardware

P and NP

- **P** = solvable by TMs in polynomial time
 - anything outside of this is "too slow" for real use
- **NP** = ... by nondeterministic TMs in poly time
 - any number of options for next state
 - e.g., (write a 0, move left, change to state s_1) or (write a 1, move right, change to state s_2)
- **P** =? **NP** is the most famous open problem in CS
 - DFAs need exponential space to simulate NFAs, do TMs need exponential time to simulate NTMs?
 - $P \neq NP$ is taken as a *law of nature* in physics

NP-Completeness

- **NP** = ... by nondeterministic TMs in poly time
 - any number of options for next state
 - e.g., (write a 0, move left, change to state s_1) or
(write a 1, move right, change to state s_2)
- Problems "as hard as any in **NP**" are "**NP-complete**"
 - prove this by reducing all NP problems to your problem
 - Cook reduce all NP problems to **SAT** in the early 1970s
 - Karp then reduced **SAT** to many other problems
 - enormous list of important problems known to be NP-complete

SAT Solvers

Problem: Given a Propositional Logic expression, is there a way to set the values of the variables to make the expression evaluate to T?

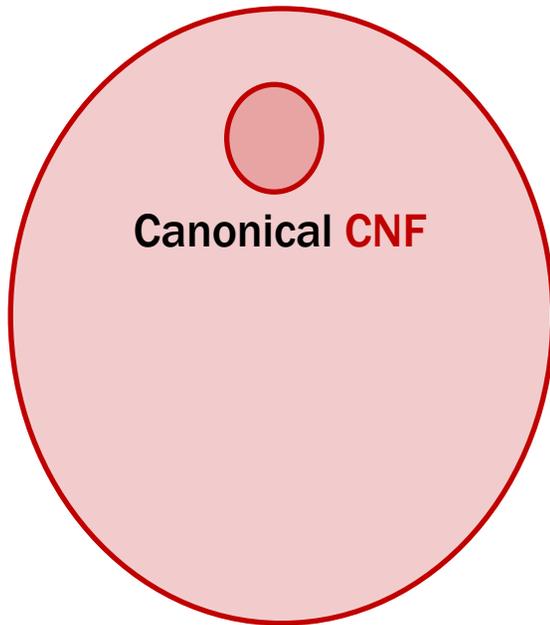
- if yes, the expression is "satisfiable"
- if not, the expression is "unsatisfiable"
- Brute force is doesn't get far with 264 variables...
 - $2^{264} \approx \#$ atoms in the observable universe
- Modern **SAT** solvers handle *millions* of variables
 - would be nice to have access to these!

SAT Solvers

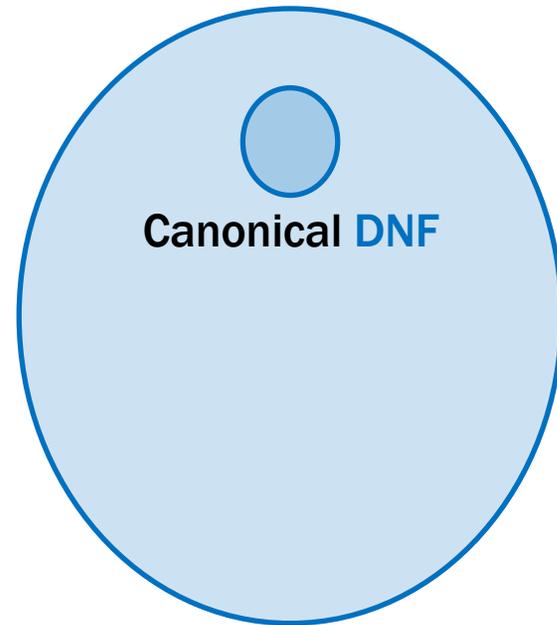
- Usually, do not accept *arbitrary* Logic expressions
 - require the expression to come in a simpler form
- Typically, require the expression in "CNF"
 - CNF not DNF due to how SAT is defined
 - "tautology" problem would want the input in DNF
- We learned about CNF before...

Recall: CNF & DNF

CNF

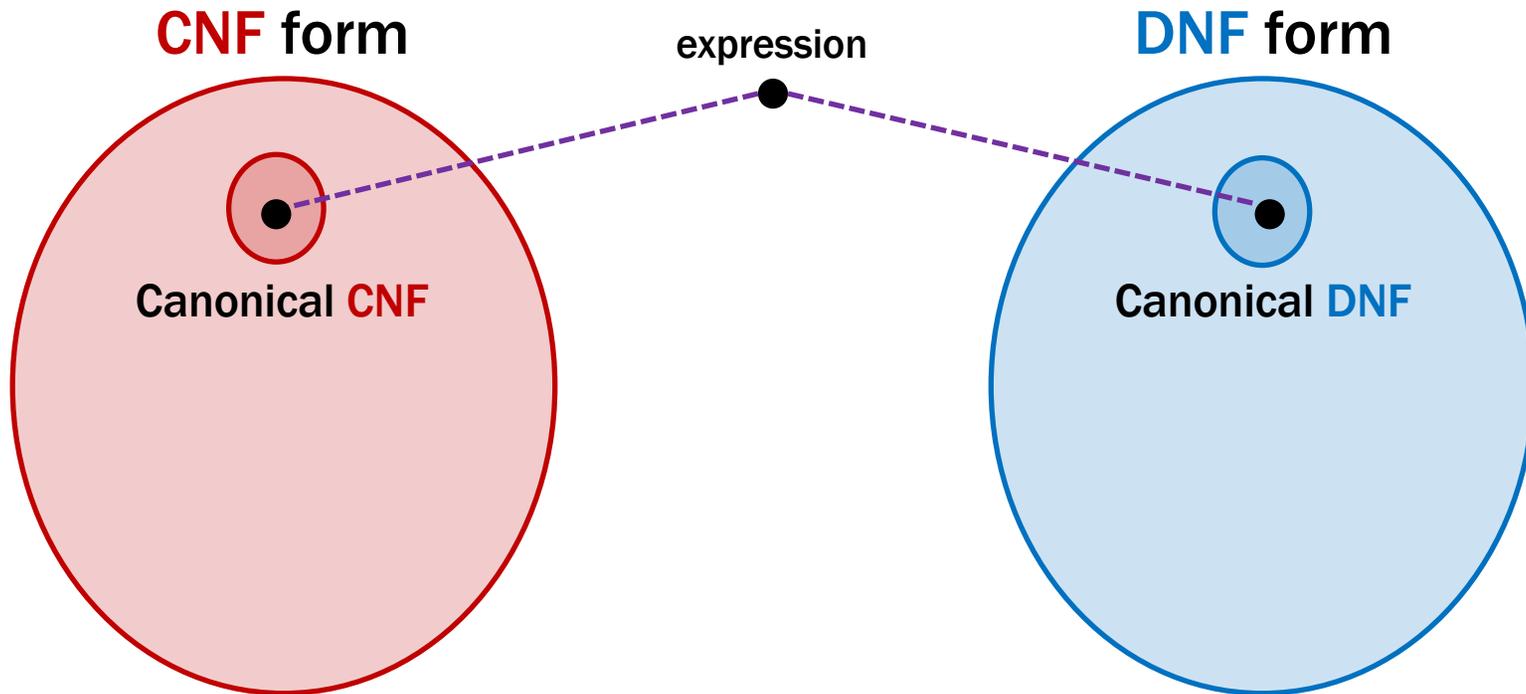


DNF



All Logic Expressions

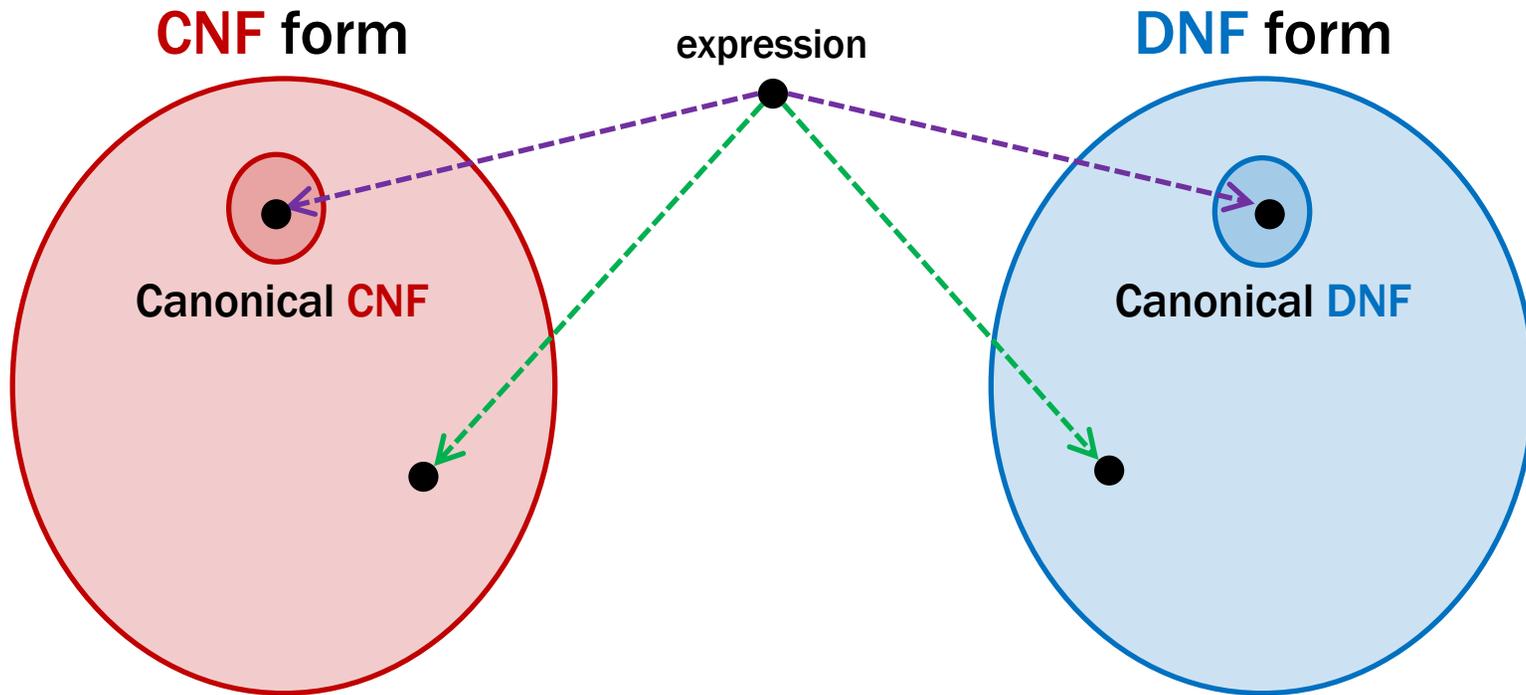
CNF & DNF



equivalent to exactly one in canonical CNF (up to reordering)

if our expressions are in canonical CNF,
then they are **equivalent** iff they are the **same**

CNF & DNF



- - - equivalent but slow conversion
- - - fast conversion but only "equi-satisfiable"
(details are outside the scope of 311)