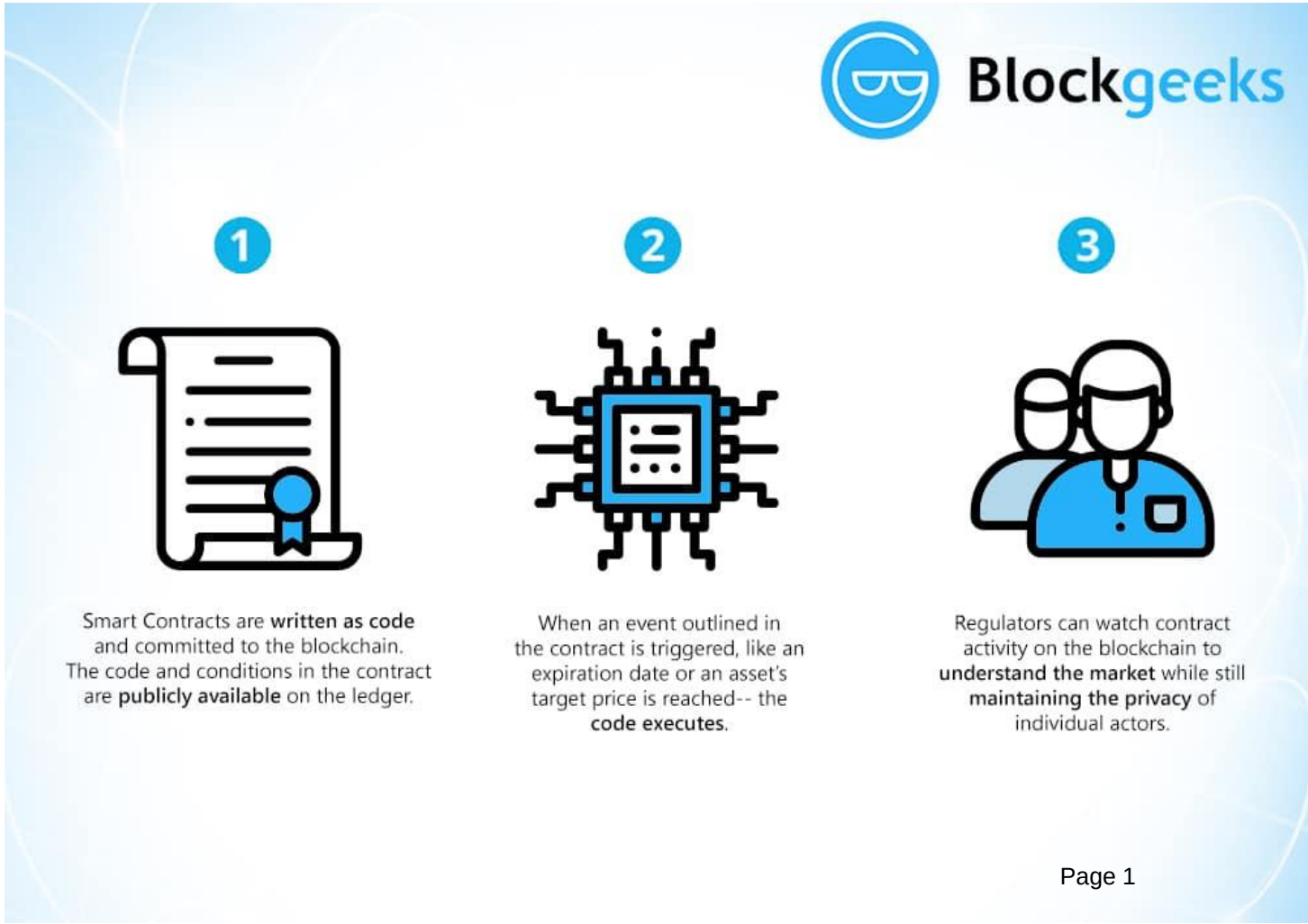
Smart Contracts

CAS Blockchain

Abraham Bernstein

Daniele Dell'Aglio



Agenda

- Rationale: What is a Smart Contract? Why Smart Contracts?
- A Detour into the Real World: Distributed Systems Principles Revisited
- A Detour into the Real World: Etherum, Solidity, and Remix
  - Writing Smart Contracts
  - Compiling and deploying the smart contracts
- Use Cases: Writing a Smart Contract in Solidity with Remix
  - Use Case 1: Auction
  - Use Case 2: Lottery in Solidity
- Smart Contracts Differently
  - Imperative Programming and Contracting
  - Logic and Rules – A Primer
  - Rule-based systems in blockchain
- Use Case: Lottery with rules
- Where to go from here...

Rationale:

- What is a Smart Contract?
- Why Smart Contracts?

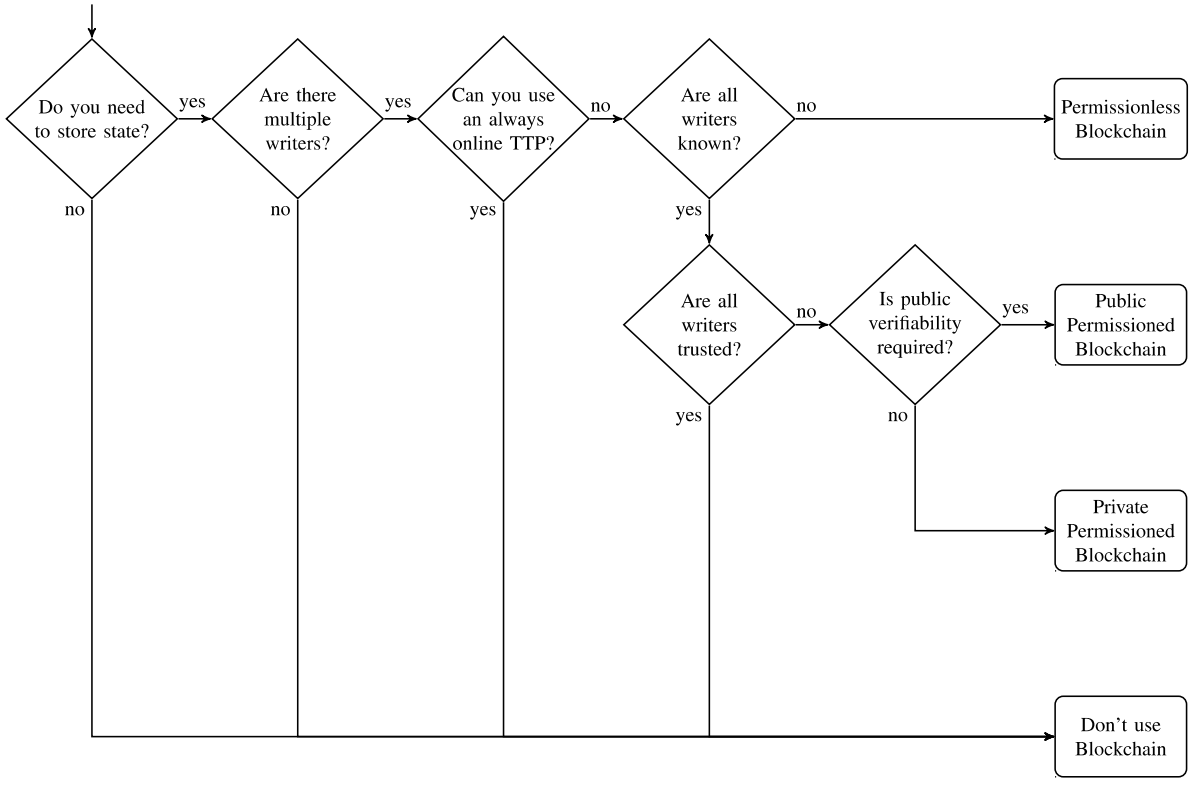
Course material inspired by:

- Antonopoulos & Wood, 2018

Blockchain in a slide…

A blockchain solution is most suitable for applications with these characteristics:

- Decentralized problems, meaning participant hold the assets and are not co-located.
- peer-to-peer transaction without intermediaries.
- Operate beyond the boundaries of trust among unknown peers.
- Require validation, verification, and recording on a universally timestamped, immutable ledger.
- Autonomous operations guided by rules and policies.



http://doyouneedablockchain.com

Smart Contracts …
… are a misnomer

– Smart Contracts are neither smart nor contracts

– They are:

Immutable, **deterministic computer programs** that run in an **isolated** fashion on the **blockchain**

– **computer program** → a collection of instructions that performs a specific task when executed by a computer
– **Immutable** → Once deployed it cannot change
– **Deterministic** → The outcome of a contract is supposed to be the same for anybody who runs **it**
– **Isolated** → The program should not affect the execution of other programs
– **Blockchain** → Runs in a distributed, autonomous, trustworthy fashion

Smart Contracts …

… are Computer Programs

– **Immutable**, **deterministic computer programs** that run in an **isolated** fashion on the **blockchain**

– **computer program** →
a collection of instructions that performs a specific task when executed by a computer

– What does that mean?

Smart Contracts …

… are Computer Programs (part II)

- **Immutable**, **deterministic computer programs** that run in an **isolated** fashion on the **blockchain**

- **computer program** →
  a collection of instructions that performs a specific task when executed by a computer

- We are not done yet: **It is desirable that the program will eventually stop**

- How can one achieve this?
  - Turing incompleteness
  - Step and Fee Meter
  - Timer
  - …

Smart Contracts …
… are Immutable

– **Immutable**, **deterministic computer programs** that run in an **isolated** fashion on the **blockchain**

– **Immutable** → Once deployed it cannot change

– What does that mean? Why is it desirable? What problems arise from it?

Smart Contracts …
… are Deterministic

- **Immutable**, **deterministic computer programs** that run in an **isolated** fashion on the **blockchain**

- **Deterministic** → The outcome of a contract is supposed to be the same for anybody who runs **it**

- What does that mean? Why is it desirable? What problems arise from it?
- Are there cases, where determinism is undesirable?

Smart Contracts …
… are Isolated

- **Immutable**, **deterministic computer programs** that run in an **isolated** fashion on the **blockchain**

- **Isolated** → The program should not affect the execution of other programs

- What does that mean? Why is it desirable? What problems arise from it? How do you achieve this?

Smart Contracts …

… run in a distributed, autonomous, trustworthy fashion

– **Immutable**, **deterministic computer programs** that run in an **isolated** fashion on the **blockchain**

– **Blockchain** → Runs in a distributed, autonomous, trustworthy fashion

– What does that mean? Why is it desirable? What problems arise from it? How do you achieve this?

Smart Contract (in summary)

A smart contract …

- … is a function that is deployed in the blockchain
  - Hence, it is an immutable piece of code (it cannot be changed once deployed)
- … operates on the blockchain, hence it leverages the immutable recording and trust model of the blockchain
- … is triggered by transactions on the blockchain, which provide its inputs, or by messages sent from other smart contracts.
- … works in the (application-specific) semantics and constraints of a transaction and verifies, validates and executes them

Smart Contracts – Brief History

- The idea of smart contracts have been around for a while
  - Nick Szabo detailed it in the 90's before bitcoin and "coined" the term
  - 1997 publication
    - http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html
  - Essentially an electronic coordination mechanism embedded in the blockchain

- Powerful feature that makes certain assumptions about the world and contracts
  - cf. open vs. closed world assumption, control over and relation real-world assets
- Buggy design can lead to significant failures of the distributed coordination
  - DAO hack
  - Parity wallet lockup.

Why would you want a Smart Contract?

- You might want to maintain Constraints on Transaction:
  - Deliver something on a certain date
  - Ensure a particular feature (certain quality or color): How do a ascertain that feature?
    - Review credentials?

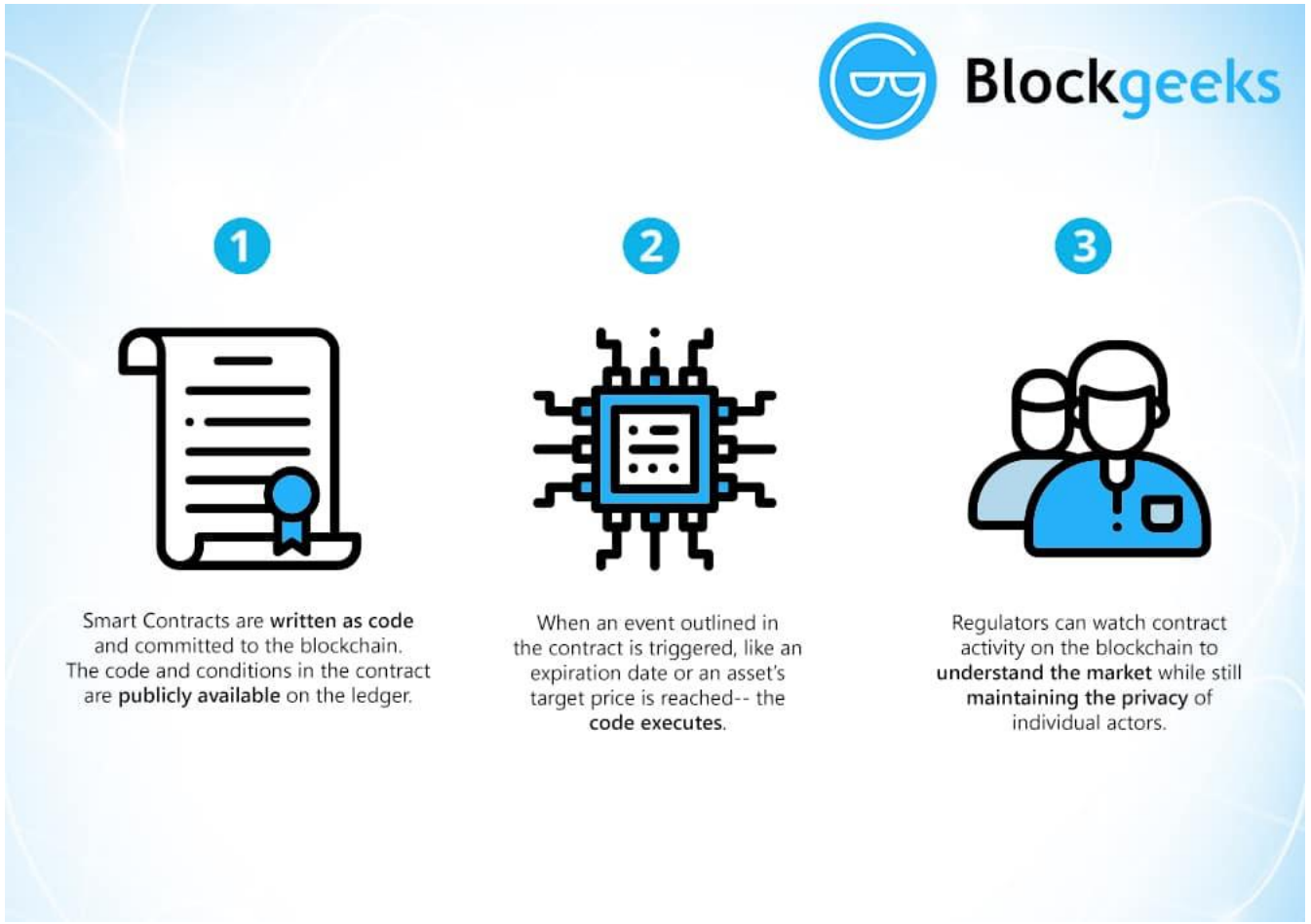
- **Introduce conditions, rules, policies into simple transactions**
  - Smart contracts provide a mechanism to achieve application-specific validation for blockchain applications.

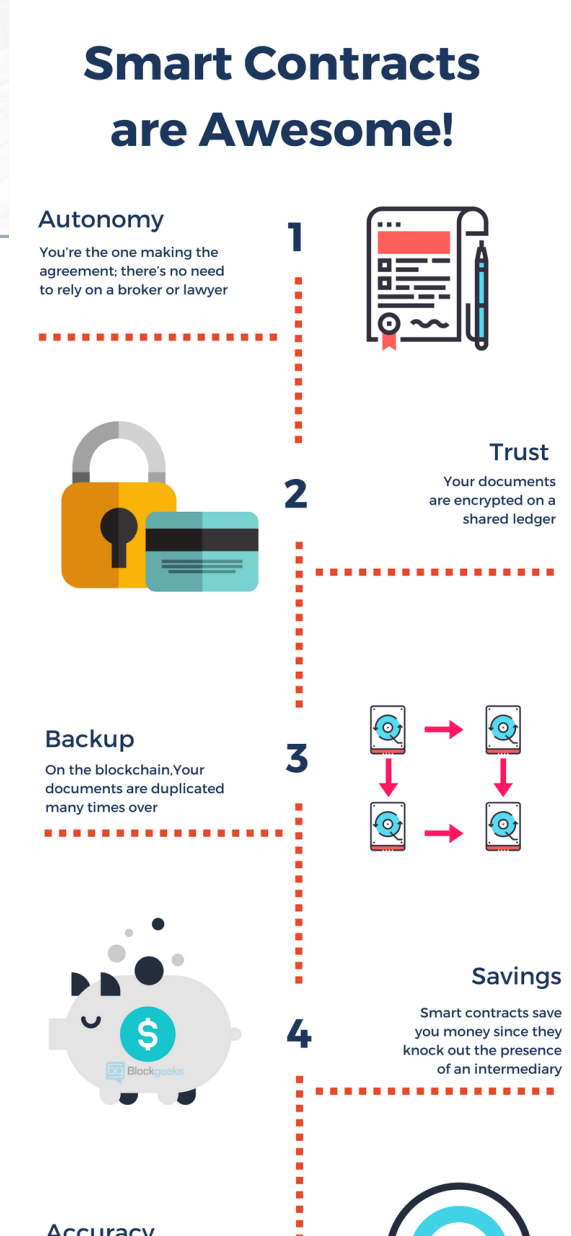
When would you want a Smart Contract

- Let's say we agree on a set of rules – I will give you a good grade, when you answer my questions correctly
- We encode these rules in a program,
- If I run the server, then I can "just" change the rules
- With blockchains, no one is in control of the rules.
- Smart contracts proponents argue we are solving a social problem

- Visible to all
  → What about secrecy?
- Executed out of control of a single party (distributed)
  → What about bugs?
- Based on collective agreement on rules, regulation, policies, or governance enforced, and the decision and the provenance for it must be recorded.
  → What if disagreement arises?
  → Are rule-bases typically complete?
- Does not replace client / server or stateless distributed solutions.

What about the Buzz?





A Detour into the Real World: Distributed Computing Principles Revisited

"A collection of independent computers that appears to its users as a single coherent system."

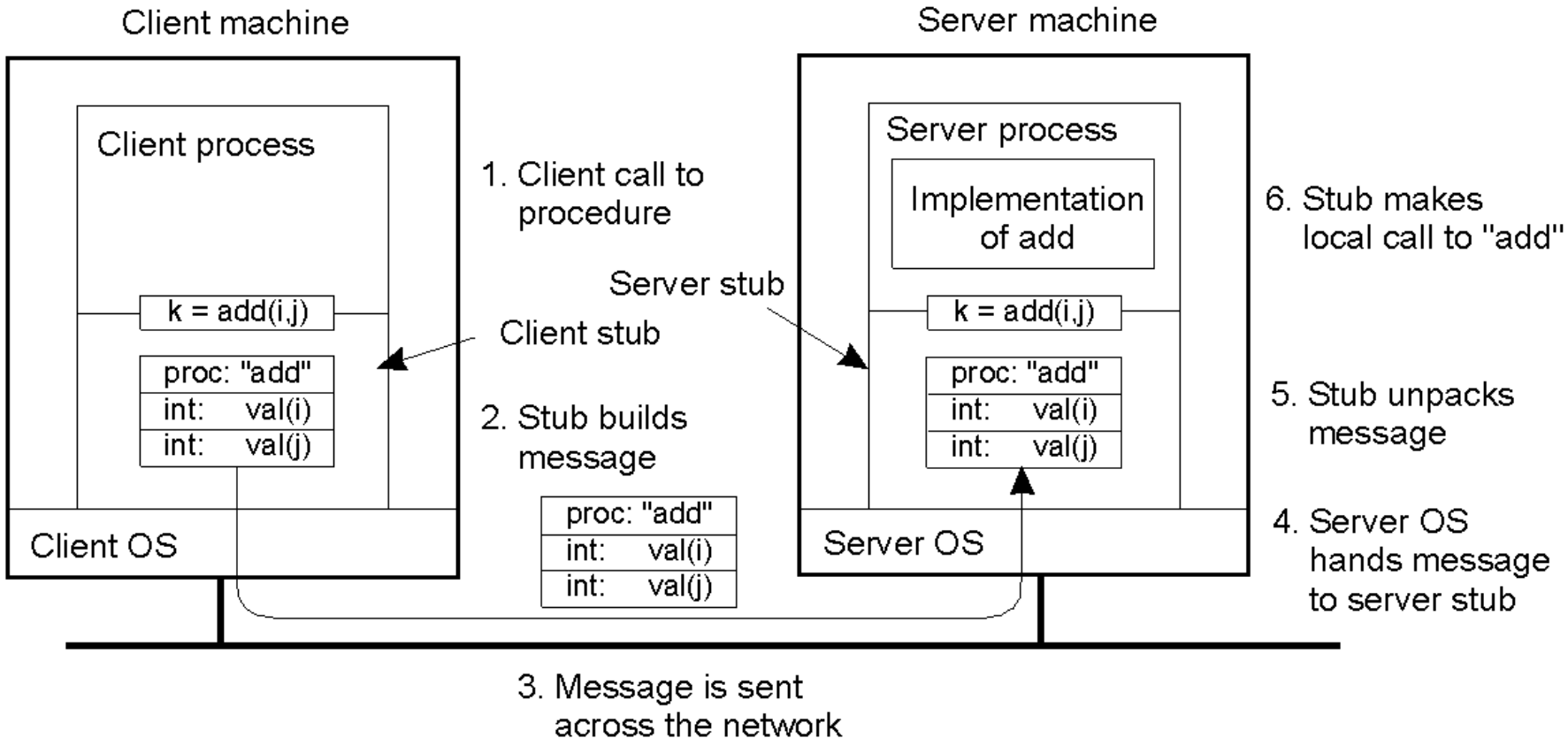
"You know you have one when the crash of a computer you've never heard of stops you from getting any work done."
Leslie Lamport

Course material based on:
- Tanenbaum & van Steen, 2007
- Coulouris et al, 2012

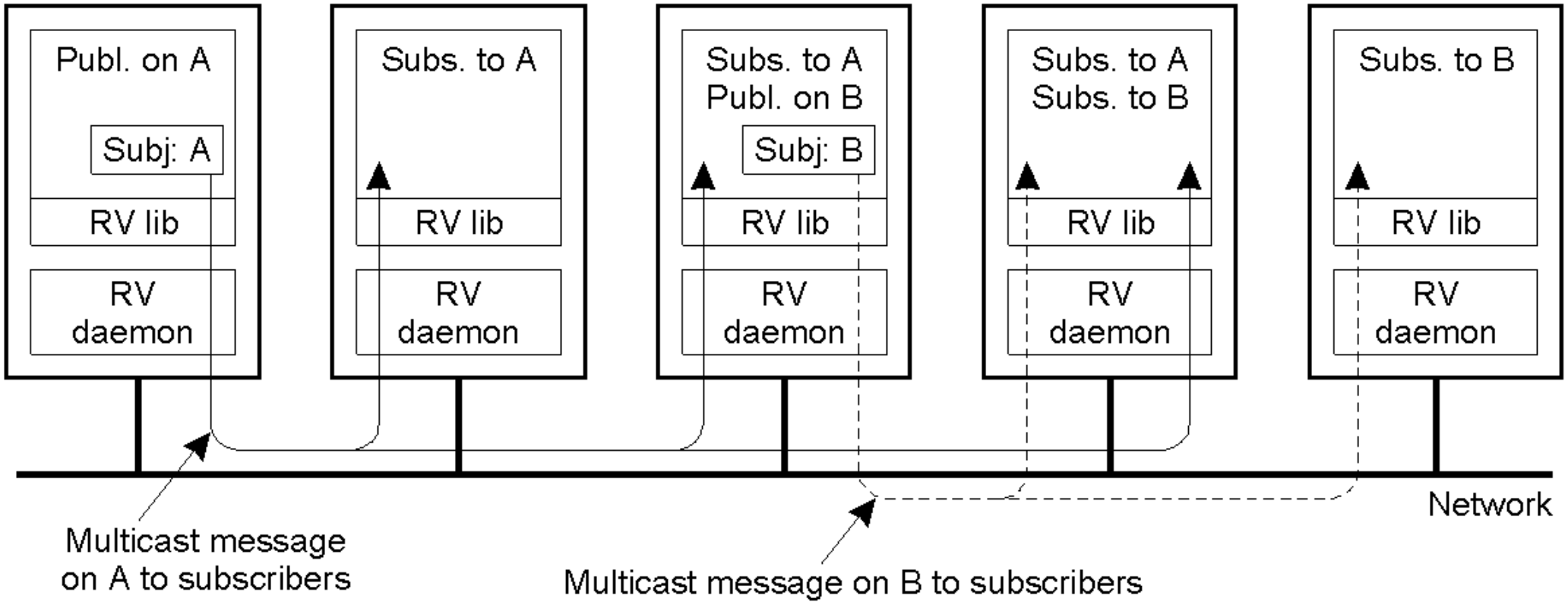
Challenges in Distributed Systems

- Heterogeneity
  - Networks
  - Computers (HW)
  - Operating systems
  - Programming languages
  - Developers
- Failure Handling
  - Detecting
  - Masking
  - Tolerating
  - Recovery
  - Redundancy
- Transparency
  - Single view of the system
  - Hide numerous details
- Scalability
  - Controlling the cost of resources
  - Controlling the performance
  - Preventing resources from running out
  - Avoiding performance bottlenecks
- Openness
  - Extensibility
  - Publication of interfaces
- Security
  - Secrecy
  - Authentication
  - Authorization
  - Non-repudiation
  - Mobile code
  - Denial of service

Steps in a Remote procedure Call



Publish / Subscribe Models (in TIB/Randezvous)



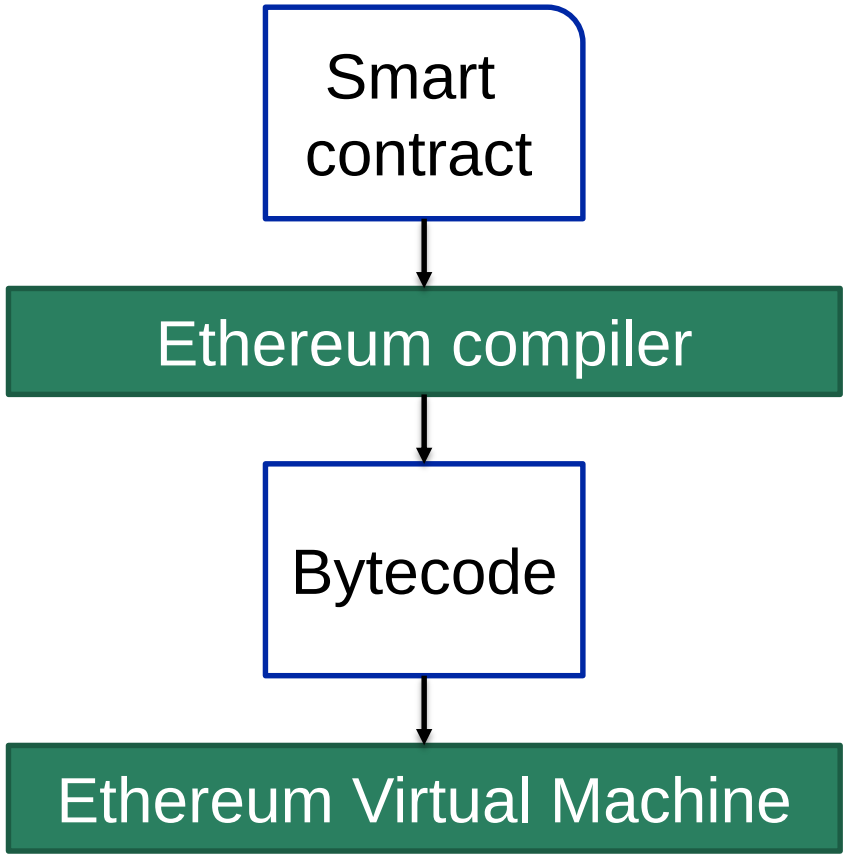
A Detour into the Real World: Etherum and Solidity

Course material based on:
- Solidity, 2019

Ethereum

- Ethereum is a mainstream blockchain
  - Compared to BitCoin, in Ethereum smart contracts are first-class citizens
- The smart contracts are executed in the Ethereum Virtual Machine (EVM)
  - It allows to execute programs in a platform-independent fashion (like Java or .NET)
  - It is designed to be distributed: the program is executed on different nodes of the network
  - It supports a Turing-complete instruction set
- The tokens are named Ether (ETH)
  - 1 Ether is 10^{18} Wei and 10^{9} Gwei
  - 1 Ether is ca. $ 385 (Sept. 2020, https://ethereumprice.org)
  - They can be used to invoke a smart contract, to cover the mining expenses, etc.
- It uses Gas to quantify the cost of executing a smart contract and set a cap to it

Creation of a smart contract



- The first step is writing a smart contract
  - Requires a programming language
- When the smart contract is ready, it is compiled by an Ethereum compiler, producing Bytecode
  - The Ethereum compiler usually optimizes the code to reduce the computational load
- The bytecode is a set of low-level instructions for the EVM
- The Bytecode is submitted to the EVM
  - It is stored in a block of the blockchain
  - It becomes available for invocation.

Install Metamask and get your first (test) ether!

Main steps:

- Crypto wallet for Firefox or Chrome
- Create a wallet
- As a network (top right), choose Ropsten
- Deposit ether
- Test faucet
- Request one ether from faucet, and connect it Metamask
- Check the wallet, and you will find your first ether!

Writing smart contracts

Course material based on:
- Akhtar et al. 2019
- Ramamurthy, 2019
- Solidity, 2019

Writing smart contracts

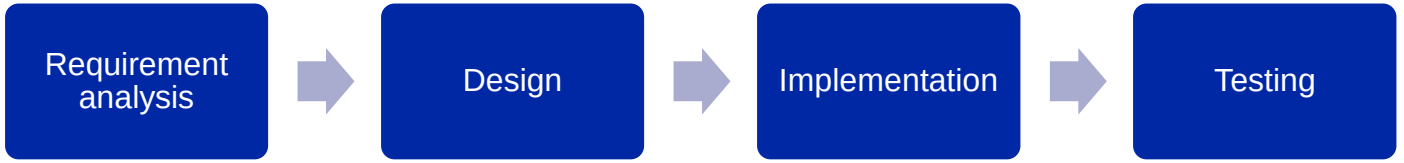
- To write a smart contract, we need a programming language
  - A set of operators to specify the business logic of the application
- There are several languages to write smart contract for Ethereum. We will focus on Solidity and Lity
  - Solidity is the most popular language for Ethereum
  - Inspired by JavaScript, Java, and C++.
  - Solidity is specifically designed to write smart contracts for the Ethereum
  - Lity is an ongoing project which is extending Solidity and the EVM to support rule execution
- Examples of other languages:
  - Vyper: Python-inspired language
  - Bamboo: alternative to Solidity focusing on the execution flow

Main phases of writing a smart contract

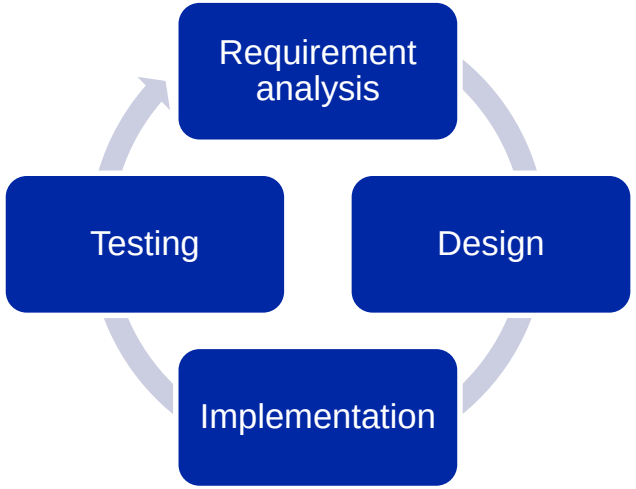
- At high level, writing smart contracts does not differ from developing other applications
- We can distinguish four main phases:
  - Requirement analysis
  - Design
  - Implementation
  - Testing

Development methodologies

- Phases can be combined in different ways:
  - Waterfall



  - Spiral



  - Agile, etc.

Requirements analysis

- The requirement analysis captures the goals and the objectives of the stakeholders, representing them as requirements
- A requirement is usually expressed in plain English, and it uses one of the following words:
  - MUST – the solution must satisfy this requirement
  - MUST NOT
  - SHOULD – a recommendation
  - SHOULD NOT
  - MAY – optional
- Having a well-defined list of requirements allow to drive the design of the software functionalities

Design

- This step takes the requirements as inputs, and define the functionalities of the software, as well as its structure and components
- UML is a standard model language for designing software solutions. It includes different diagrams to represent the components, the data model, and so on.
- While designing a smart contract, it is key to identify:
  - Which data is needed for the processing – ideally, we want to keep in the blockchain a “minimal” amount of information, to reduce costs and to avoid to make all the information public. We distinguish between in-chain and off-chain data.
  - What users interacting with our contract should send and what they should receive back (more in the next slides). This is the *interface* of the smart contract, and a simple and clear interface simplify the stakeholder experience
  - Which calls should be charged and which should be free

Implementation

- It is time to start to write the smart contract in Solidity!
- A smart contract is a file with extension sol
- It contains:
  - Data (or state variables)
  - User-defined types (struct and enum types)
  - Functions
  - Function modifiers
  - Events

Declaring the smart contract

```
pragma solidity >=0.5.0;

contract MyFirstContract {
        …
}
```

- pragma indicates which version of the compiler is required to compile the contract
- Solidity uses camel-case style for names, i.e. variables, functions and contracts with complex names use capital letter and do not use space
  - my_first_contract would be an example of snake-case style

State variables

- State variables permanently store data in the contract
- Defining a variable requires three elements:
  - A name
  - A data type
  - A visibility level (default: `internal`)
- The name should be meaningful, ideally self-explicative

Data types – base types

- Solidity supports many of the basic types of a high-level programming language:
  - Booleans (`bool`)
  - Signed and unsigned integers (`int`/`uint`)
  - String literals (`string`)

```
pragma solidity >=0.5.0;

contract MyFirstContract {
        uint age;
        string name;
        …
}
```

Data types – addresses

- A key type is `address`:
  - It identifies Ethereum addresses
    - E.g. 0x1ED13B2108B2c43e7e6Dbf7f089205A3A3b2A69D
  - Each user and smart contract in the blockchain has assigned an address
    - It is a unique identifier
    - Contains the balance in wei
  - It can be used to transfer Ether between accounts or to invoke a smart contract

Data types – arrays and mappings

- Arrays are list of variables of the same type
  - We can have arrays of arrays (e.g. matrices and tensors)
- A `mapping` is a key-value store (an hash map)
  - It allows to access the value given the key
    - An address book can be stored in a key-value store: given the name (key) it is possible to retrieve the address (the value)
    - It is not possible to retrieve the key given the value

```
uint[] grades;
mapping(address => uint) ages;
…
grades[10]; //retrieves the 11th element
ages[0x1ED13B2108B2c43e7e6Dbf7f089205A3A3b2A69D]; //retrieves the age of 0x1ED13…
```

Data types – structs and enums

- It often happens we want to describe complex entities, e.g. a person with a name, an age and an address. The `struct` type allows to declare user-defined types assigning it a unique name.
  - Individual elements can be accessed through the dot operator
- An enumeration (`enum`) is a set of possible values a variable can assume, e.g. North, South, West and East
  - Enumerations are useful to define states of smart contracts.

```
enum Priority {Low, Medium, High};
strcut Alert {
        Priority priority;
        string message;
}
Alert a = new Alert(Priority.High, “The sink is leaking!”)
a.priority; // retrieves the priority of a
```

State variable visibility

- The visibility indicates who can access the state variable
  - `public`: they can be accessed inside and outside the contract
  - `private`: they are visible only in the contract where they are declared
  - `internal`: they can be accessed from the contract and extending contracts
- If no visibility is declared, the visibility is set to `internal`.
- When the state variable is `public`, Solidity generates a getter function
  - A special function which returns the content of the variable

```
uint age; // the visibility level is internal
string private name;
```

Functions

- State variables define a state and store data, but they cannot process and modify it
- Functions embeds operations as set of instructions to be executed
  - Functions may access and modify the state of the contract
- To define a function we need:
  - A name
  - Zero or more input arguments (defined as data type + name)
  - Zero or more output parameter data types
  - A level of visibility and (optionally) modifiers

```
function checkAge(uint age) returns(bool) {
        return age >= 18;
}
checkAge(15) // returns false
```

Function visibility

- As state variables, also functions have visibility
- They are:
  - `public`: the function can be invoked inside and outside the contract
  - `private`: the function can be invoked in the contract that defined it
  - `internal`: the function can be invoked inside the function that defined it and in the extending contracts
  - `external`: the function can be invoked only from outside the contact defining it
- The default visibility level is `internal`.

Function modifiers – payable, pure, view

- Function declarations can be enriched with four modifiers:
  - `payable`: allows a function to receive Ether when invoked
    - The payment is collected by the contract (a contract can receive and send Ether, as a user)
  - `pure`: the execution of the function does depend only on the input arguments, and there are no operations on state variables (read/write)
  - `view`: the function can access the state variable but cannot modify them

```
function checkAge(uint age) public pure returns(bool) {
        return age >= 18;
}
checkAge(15) // returns false
```

Constructor

- The constructor is a special function which initialize the state
  - It takes zero or more input arguments and returns zero
  - Every contract can have at most one constructor
  - It is optional (it may miss)

```
contract OwnedContract {
        address owner;
        constructor() { owner = msg.address; } // msg is a special object with information
                                                about the user invoking the function
        function whoIsTheOwner() public view returns(address) { return owner; }
}
```

Function modifiers – require

- Require sets conditions which need to be satisfied to execute the function
- When the condition is not satisfied, the function execution stops and returns a message

```
contract OwnedContract2 {
        address owner;
        constructor() { owner = msg.address; }
        function sayMeHi() public view returns(string) {
                require(msg.sender == owner, “Only the owner can invoke me”);
                returns “Hello!”
        }
}
```

Function modifiers – modifier

- modifier can group the require statements and assign them a name
- Functions can use then use the modifiers as the base ones (e.g. view)

```
contract OwnedContract3 {
        address owner;
        modifier onlyOwner {
                require(msg.sender == owner, “Only the owner can invoke me”);
        }
        constructor() { owner = msg.address; }
        function sayMeHi() public onlyOwner returns(string) { returns “Hello!” }
        function sum(uint a, uint b) public onlyOwner returns(uint){ returns a+b; }
}
```

Inheritance

- Contracts can inherit state variables, functions, etc. from other contracts
- The inheriting contract can access the `public` and `internal` state variables and functions

```
contract OwnedContractBase {
        address owner;
        modifier onlyOwner { require(msg.sender == owner, “You are not the owner”); }
        constructor() { owner = msg.address; }
}
contract TimeContract is OwnerContractBase {
        function tellMeTime() public onlyOwner returns(uint) { return now; }
}
```

- `now` is a special keyword that returns the timestamp of the current block (in UNIX time)

Tools for writing smart contracts

- To write a smart contract, any text editor can do the job
- It is, however, better to use an IDE supporting Solidity as programming language
  - Text highlighting
  - Code completion
  - Etc.
- Microsoft Visual Studio Code is an open source editor which supports Solidity
- Remix is an online IDE specifically designed for Solidity and Ethereum
  - It includes a compiler and runtime test environments to try the smart contracts
  - Online at: https://remix.ethereum.org/#optimize=false

Testing

- After we write the smart contract, we want to know if it works
- Deploying it to the public Ethereum blockchain is not the best solution
  - It's expensive
  - We may want to make several tests – redeploy several times the contract
- The best way to go is to have a local testing node
  - The node is disconnected from the rest of the blockchain network
  - They allow to create different users with high budget
  - The mining of the transactions is almost instantaneous
- Some IDE offers an internal testing nodes
- Alternatively, we can install it as standalone applications

Writing smart contracts – some remarks

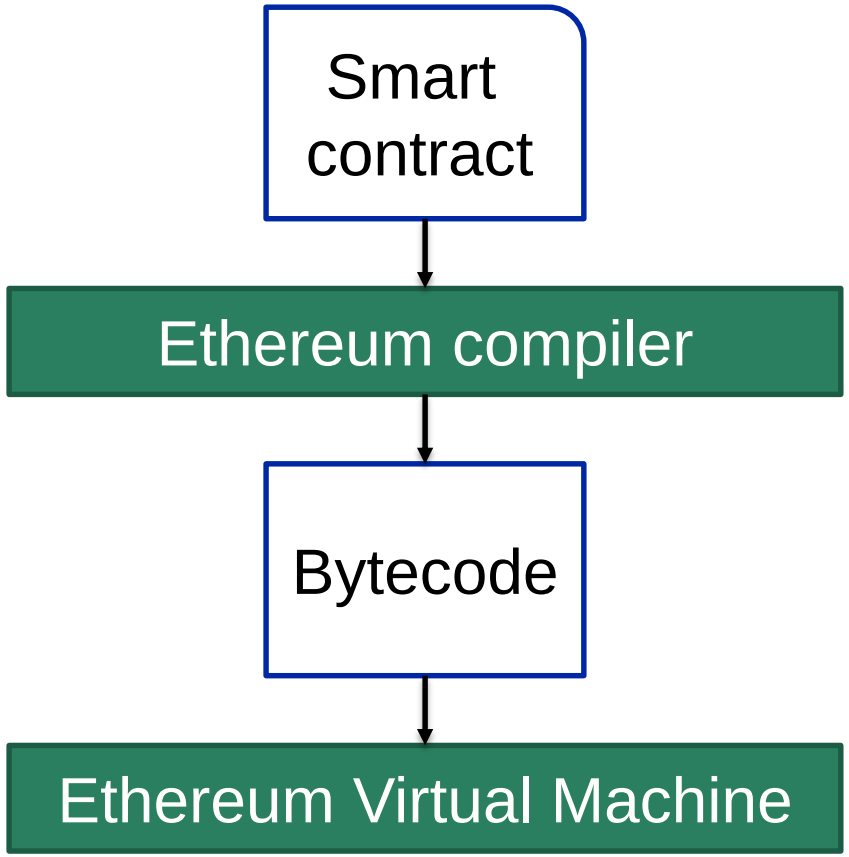
- Follow the Keep It Simple principle
  - Design a contract focusing on one specific operation
  - Write simple functions, keep the code readable
    - Put all the state variables at the beginning of the contract
    - Order functions in the following order: constructor, external functions, public functions, internal functions, private functions
  - Do not put code which is not strictly required in the contract
  - Control the input arguments
    - Exploit function modifiers
- Remember that what is in the blockchain is visible to everyone
  - Make a good use of the visibility operators
  - Off-chain vs in-chain data

Compiling and deploying the smart contracts

Course material based on:
- Ramamurthy, 2019
- Akhtar et al., 2019
- Solidity, 2019

From the Solidity file to the bytecode

- Now that the smart contract is written, we should compile it
- We are interested in extracting two elements
  - The bytecode: a long text, starting with 0x, with all the instructions encoded in a binary format
  - The Application Binary Interface (ABI): a JSON file describing the interface of the smart contract:
    - The public and external methods which can be invoked by other users/contracts



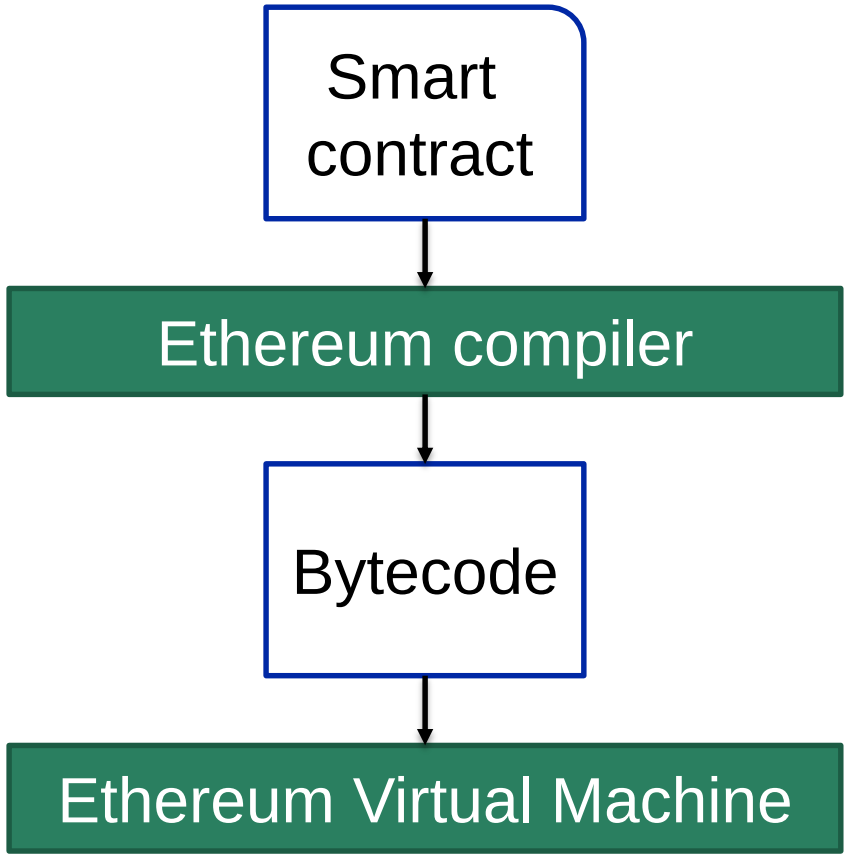
How to generate the bytecode and the ABI

- Short and easy way :
  - Remix embeds a compiler which can extract the bytecode and the ABI from the smart contract
- Long and geek way:
  - We can use a framework to compile the smart contract from command line (e.g. solc)

```
$ solc --abi contract.sol

========== contract.sol:MyContract ==========
Contract JSON ABI
[{“constant”:false,...

$ solc --bin contract.sol

========== contract.sol:MyContract ==========
Binary:
60805943095893 40...
```

Deploying the smart contract

- The last step is deploying the smart contract
- We can do that by submitting the ABI and the bytecode to the Ethereum Virtual Machine
- In addition, we should provide other information, such as:
  - Who is deploying the contract
  - The maximum amount of gas
- If successful, the operation returns an address, which can be used to invoke the smart contract methods



How to deploy the smart contract

- Short and easy way:
  - Remix can deploy the smart contract to the EVM (live demo in a few minutes)
- Long and geek way:
  - We can interact with the EVM API and submit the request programmatically. We can for example use web3-js, a Javascript library to interact with an EVM, and execute the following code:

```
contract = web3.eth.contract([{“constant”:false,...); // pass the ABI as the argument
contractInstance = contract.new({
                from: 0x1ED13B2108B2c43e7e6Dbf7f089205A3A3b2A69D, //submitter address
                data: 0x6080601640500082348015610005 7..., //contract bytecode
                gas: ”4700000”
        }, function(e, contract) {
                console.log(“contract address: ” + contract.address);
        });
```

Use Case 1: Auction

An auction on blockchain

- We want to make an auction on the blockchain
- One person (the owner) deploy the auction contract
- Everyone can bid
- When the auction closes, the person who bid highest wins
- No one can bid afterwards

- NOTE: Building a complete auction contract is beyond the scope of the class, the today example is meant to show some of the Solidity features

An auction on blockchain
0) Defining the methods

– See Example1.0.sol

An auction on blockchain
1) Implementing the first version of the contract

– See Example1.1.sol

An auction on blockchain
2) Manage the validation of the inputs with require

– See Example1.2.sol

An auction on blockchain
3) Replace name with the sender address

– See Example1.3.sol

An auction on blockchain
4) Using ether for bidding

– See Example1.4.sol

An auction on blockchain
5) Avoid uninfluential bids

– See Example1.5.sol

An auction on blockchain
6) Only the owner can close the auction

– See Example1.6.sol

An auction on blockchain
7) Give back money to the losers

– See Example1.7.sol

Use Case 2: Lottery in Solidity (see Example2.sol)

A lottery in the blockchain

– Start from Example2.0.sol and solve the following tasks

1. Replace the explicit getter methods with the intrinsic ones offered by Solidity
2. Fill the addGuess function. Decide how much one should pay to get a “lottery ticket”
3. Fill the decideWinner function. Decide how much one should win. Remember that the contract should have enough money to pay all the winners. The owner should cover the expenses if there are too many winners

Smart Contracts Differently: Imperative Programming and Contracting

Slides based on:

- Russel and Norvig, 2009

Programming Paradigms

Imperative Programming

- **procedural**, which groups instructions into procedures,
- **object-oriented** which groups instructions together with the part of the state they operate on,

Declarative Programming

- **functional** in which the desired result is declared as the value of a series of function applications,
- **logic** in which the desired result is declared as the answer to a question about a system of facts and rules,
- **mathematical** in which the desired result is declared as the solution of an optimization problem

Programming Paradigms

Imperative Programming

- Identify problem
- Assemble information
- Figure out solution
- Program solution
- Encode problem instance as data
- Apply program to data
- Debug procedural errors

Declarative (Logic) Programming

- Identify problem
- Assemble information
- Tea break
- Encode information in KB
- Encode problem instance as facts
- Ask queries
- Find false facts

Used in:
- Databases
- Constraint satisfaction
- Optimization
- …

Should be easier to debug *Capital(NewYork,US)* than *x:= x+2*!

Smart Contracts Differently: Logic and Rules – A Primer

Slides based on:

- Russel & Norvig, 2009

Overview: Smart Contracts Differently: Logic and Rules – A Primer

- Knowledge Bases
- Logic in general
  - Forward chaining
  - Backward chaining
- First Order Logic (FOL)
  - Why FOL?
  - Examples of FOL
  - Forward chaining
  - Backward chaining
- Logic programming

Knowledge Bases

- Knowledge base = set of *sentences* in a *formal* language
- **Declarative** approach to building an agent (or other system):
  `Tell` it what it needs to know
- Then it can `Ask` itself what to do – answers should follow from the KB
- Agents can be viewed at the *knowledge level*
  i.e., what they know, regardless of how implemented
- Or at the *implementation level*
  i.e., data structures in KB and algorithms that manipulate them

A simple knowledge-based agent

- The agent must be able to:
  - Represent states, actions, etc.
  - Incorporate new percepts
  - Update internal representations of the world
  - Deduce hidden properties of the world
  - Deduce appropriate actions

function KB-AGENT(*percept*) **returns** an *action*
  static: KB, a knowledge base
    t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(*percept*, t))
  $action \leftarrow$ ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE($action$, t))
  $t \leftarrow t + 1$
  return *action*

Logic in general

Logics are formal languages for representing information such that conclusions can be drawn

Syntax defines the sentences in the language

Semantics define the „meaning“' of sentences; i.e., define *truth* of a sentence in a world

E.g., the language of arithmetic

- $x+2 \geq y$ is a sentence; $x2+y>$ is not a sentence
- $x+2 \geq y$ is true iff the number x+2 is no less than the number y
- $x+2 \geq y$ is true in a world where $x= 7, y= 1$
- $x+2 \geq y$ is false in a world where $x= 0, y= 6$

Inference

- $KB \vdash_i \alpha$ = sentence α can be derived from *KB* by procedure *i*
  - Consequences of *KB* are a haystack; α is a needle.

- ***Soundness***: *i* is sound if
  whenever $KB \vdash_i \alpha$, it is also true that α can be derived from *KB*
- ***Completeness***: *i* is complete if
  whenever α can be derived from *KB*, it is also true that $KB \vdash_i \alpha$
  - Preview: we will define a logic (first-order logic) which is expressive enough to say almost anything of interest, and for which there exists a sound and complete inference procedure.
    That is, the procedure will answer any question whose answer follows from what is known by the *KB*.

Propositional logic: Syntax

Propositional logic is the simplest logic – illustrates basic ideas

The proposition symbols P_1, P_2 etc are sentences

If S is a sentence, $\neg S$ is a sentence (*negation*)

If S_1 and S_2 are sentences, $S_1 \wedge S_2$ is a sentence (*conjunction*)

If S_1 and S_2 are sentences, $S_1 \vee S_2$ is a sentence (*disjunction*)

If S_1 and S_2 are sentences, $S_1 \Rightarrow S_2$ is a sentence (*implication*)

If S_1 and S_2 are sentences, $S_1 \Leftrightarrow S_2$ is a sentence (*biconditional*)

Propositional logic: Semantics

- Each model specifies true/false for each proposition symbol
- E.g. P1,2 P2,2 P3,1
  false false true
- (With these symbols, 8 possible models, can be enumerated automatically.)
- Rules for evaluating truth with respect to a model m:

$$
\begin{array}{rlllllll}
\neg S & \text{is true iff} & S & \text{is false} & & & & \\
S_1 \wedge S_2 & \text{is true iff} & S_1 & \text{is true and} & S_2 & \text{is true} & & \\
S_1 \vee S_2 & \text{is true iff} & S_1 & \text{is true or} & S_2 & \text{is true} & & \\
S_1 \Rightarrow S_2 & \text{is true iff} & S_1 & \text{is false or} & S_2 & \text{is true} & & \\
\text{i.e.,} & \text{is false iff} & S_1 & \text{is true and} & S_2 & \text{is false} & & \\
S_1 \Leftrightarrow S_2 & \text{is true iff} & S_1 \Rightarrow S_2 & \text{is true and} & S_2 \Rightarrow S_1 & \text{is true} & &
\end{array}
$$

Simple recursive process evaluates an arbitrary sentence, e.g.,

$$\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1}) = true \wedge (false \vee true) = true \wedge true = true$$

Truth tables for connectives

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
$false$	$false$	$true$	$false$	$false$	$true$	$true$
$false$	$true$	$true$	$false$	$true$	$true$	$false$
$true$	$false$	$false$	$false$	$true$	$false$	$false$
$true$	$true$	$false$	$true$	$true$	$true$	$true$

Logical equivalence

Two sentences are *logically equivalent* iff true in same models:

$\alpha \equiv \beta$ if and only if $\alpha \models \beta$ and $\beta \models \alpha$

$$
\begin{array}{rcll}
(\alpha \wedge \beta) & \equiv & (\beta \wedge \alpha) & \text{commutativity of } \wedge \\
(\alpha \vee \beta) & \equiv & (\beta \vee \alpha) & \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) & \equiv & (\alpha \wedge (\beta \wedge \gamma)) & \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) & \equiv & (\alpha \vee (\beta \vee \gamma)) & \text{associativity of } \vee \\
\neg(\neg\alpha) & \equiv & \alpha & \text{double-negation elimination} \\
(\alpha \implies \beta) & \equiv & (\neg\beta \implies \neg\alpha) & \text{contraposition} \\
(\alpha \implies \beta) & \equiv & (\neg\alpha \vee \beta) & \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) & \equiv & ((\alpha \implies \beta) \wedge (\beta \implies \alpha)) & \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) & \equiv & (\neg\alpha \vee \neg\beta) & \text{De Morgan} \\
\neg(\alpha \vee \beta) & \equiv & (\neg\alpha \wedge \neg\beta) & \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) & \equiv & ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) & \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) & \equiv & ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) & \text{distributivity of } \vee \text{ over } \wedge
\end{array}
$$

Proof methods

- Proof methods divide into (roughly) two kinds:

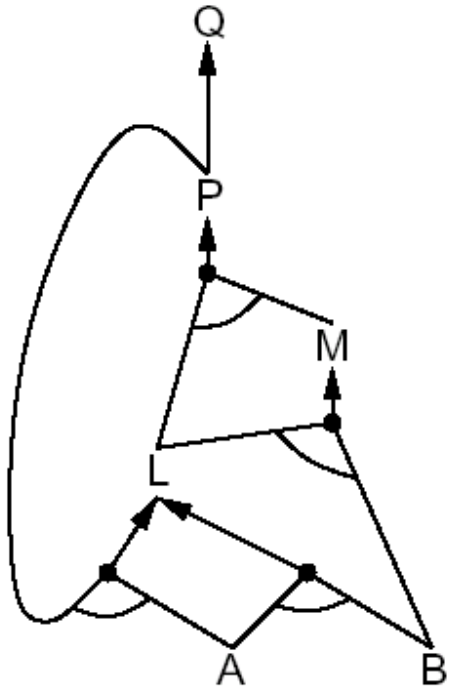
- *Application of inference rules*
  - Legitimate (sound) generation of new sentences from old
  - *Proof* = a sequence of inference rule applications
    Can use inference rules as operators in a standard search alg.
  - Typically require translation of sentences into a *normal form*

- *Model checking*
  - truth table enumeration (always exponential in *n*)
  - improved backtracking, e.g., Davis-Putnam-Logemann-Loveland
  - heuristic search in model space (sound but incomplete)
    e.g., min-conflicts-like hill-climbing algorithms

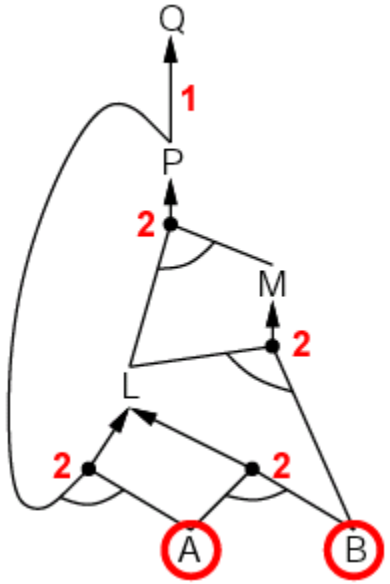
Forward chaining

Idea: fire any rule whose premises are satisfied in the *KB*, add its conclusion to the *KB*, until query is found

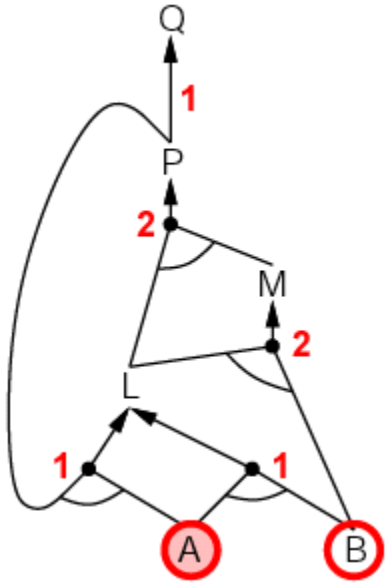
$$P \Rightarrow Q$$
$$L \wedge M \Rightarrow P$$
$$B \wedge L \Rightarrow M$$
$$A \wedge P \Rightarrow L$$
$$A \wedge B \Rightarrow L$$
$$A$$
$$B$$



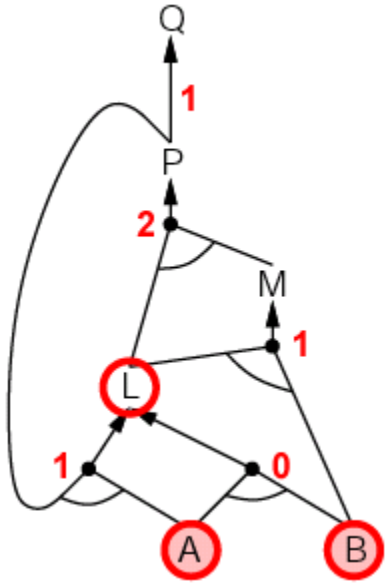
Forward chaining example



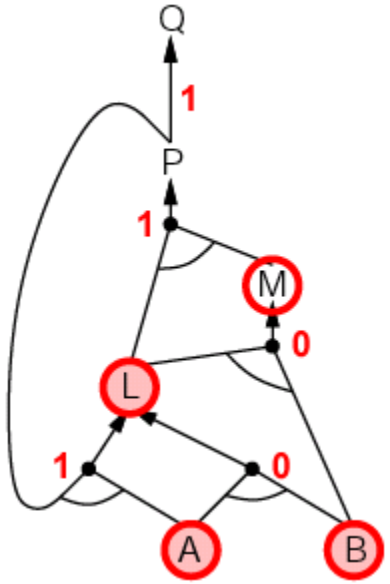
Forward chaining example



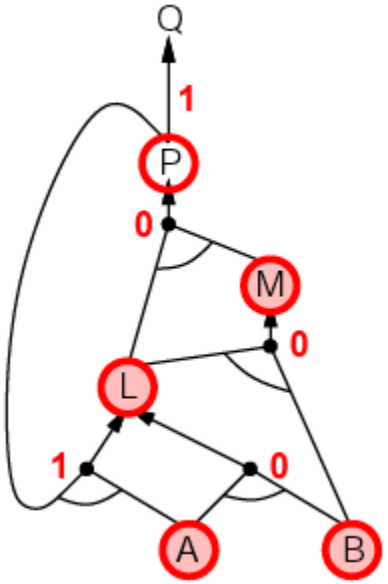
Forward chaining example



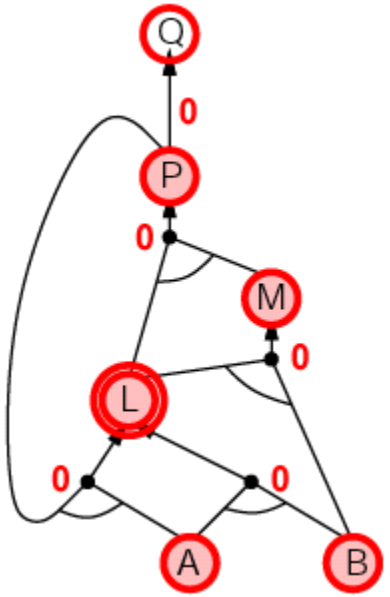
Forward chaining example



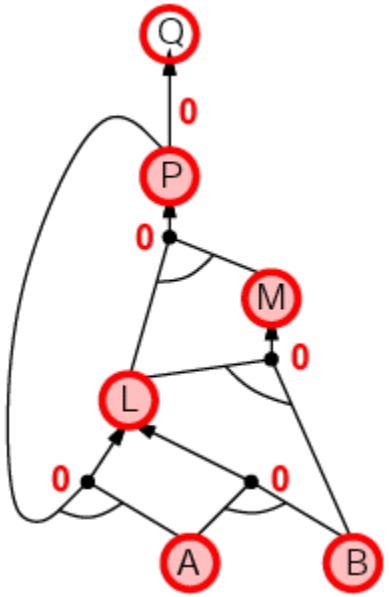
Forward chaining example



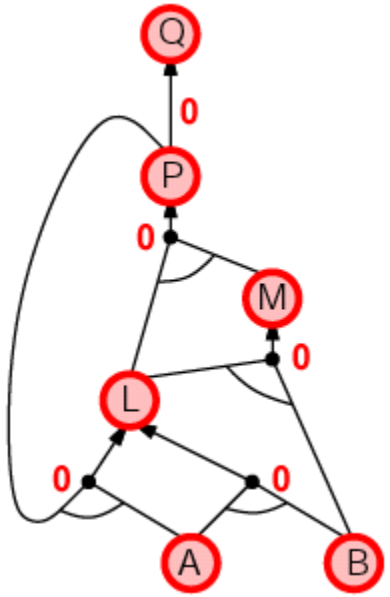
Forward chaining example



Forward chaining example



Forward chaining example



Proof of completeness

FC derives every atomic sentence that is entailed by *KB*

1. FC reaches a *fixed point* where no new atomic sentences are derived
2. Consider the final state as a model *m*, assigning true/false to symbols
3. Every clause in the original *KB* is true in *m*

   Proof: Suppose a clause $a_1 \wedge \ldots \wedge a_k \Rightarrow b$ is false in *m*

   Then $a_1 \wedge \ldots \wedge a_k$ is true in *m* and *b* is false in *m*

   Therefore the algorithm has not reached a fixed point!
4. Hence *m* is a model of *KB*
5. If *KB* |= *q*, *q* is true in ***every*** model of *KB*, including *m*

Backward chaining

Idea: work backwards from the query *q*:

to prove *q* by BC,

check if *q* is known already, or

prove by BC all premises of some rule concluding *q*

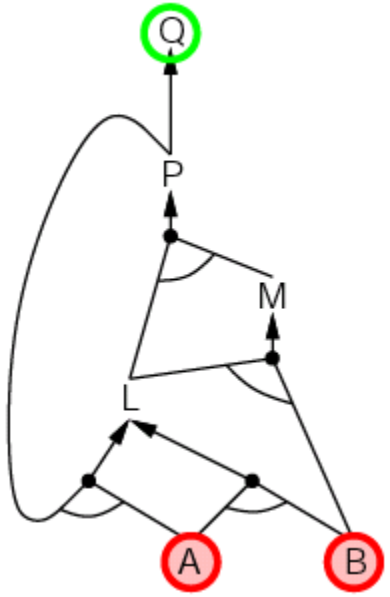
Avoid loops: check if new subgoal is already on the goal stack

Avoid repeated work: check if new subgoal

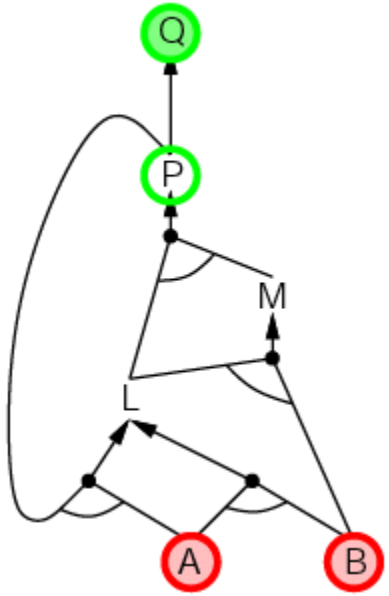
1) has already been proved true, or

2) has already failed

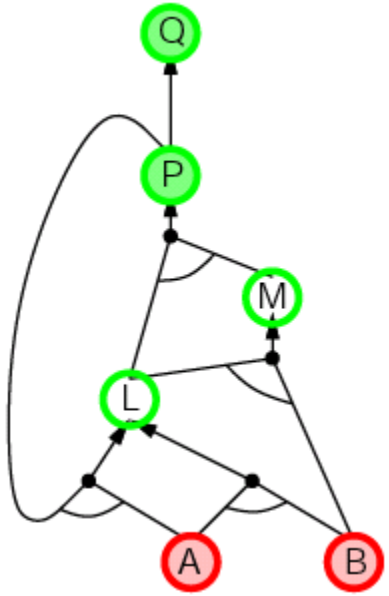
Backward chaining example



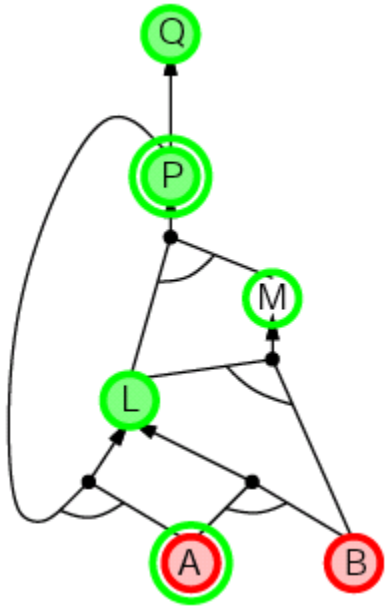
Backward chaining example



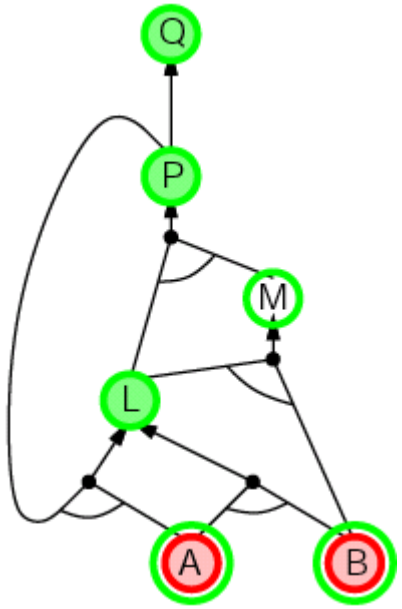
Backward chaining example



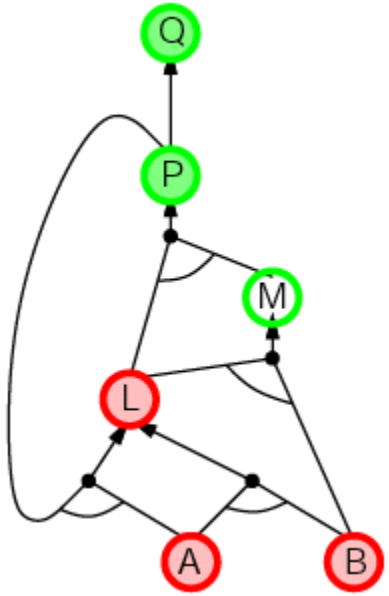
Backward chaining example



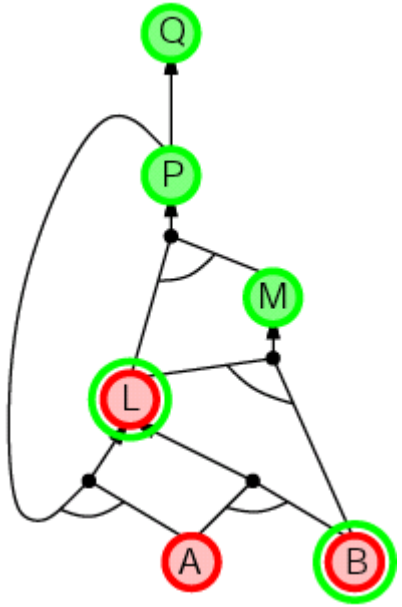
Backward chaining example



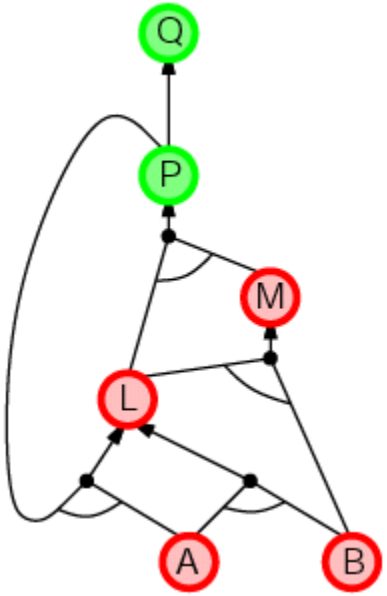
Backward chaining example



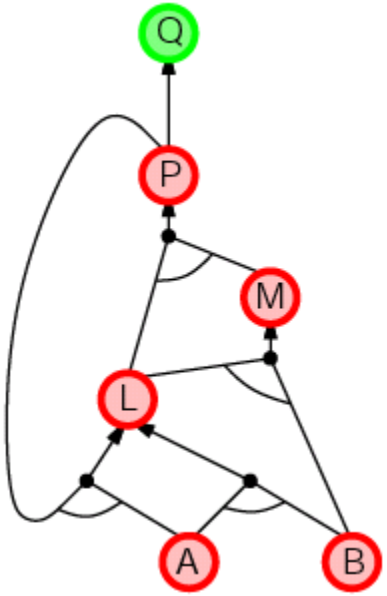
Backward chaining example



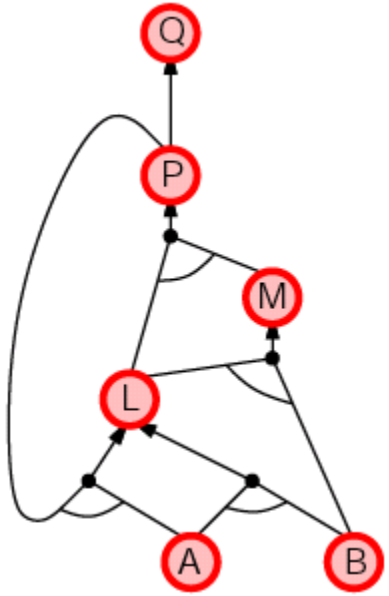
Backward chaining example



Backward chaining example



Backward chaining example



Forward vs. backward chaining

– FC is *data-driven*, cf. automatic, unconscious processing,
  – e.g., object recognition, routine decisions

  May do lots of work that is irrelevant to the goal

– BC is *goal-driven*, appropriate for problem-solving,
  – e.g., Where are my keys? How do I get into a PhD program?

  Complexity of BC can be ***much less*** than linear in size of KB

Summary

- We can apply *inference* to a *knowledge base* to derive new information and make decisions
- Basic concepts of logic:
  - *syntax*: formal structure of *sentences*
  - *semantics*: *truth* of sentences wrt *models*
  - *entailment*: necessary truth of one sentence given another
  - *inference*: deriving sentences from other sentences
  - *soundess*: derivations produce only entailed sentences
  - *completeness*: derivations can produce all entailed sentences
- Forward, backward chaining are linear-time, complete for Horn clauses
- Propositional logic lacks expressive power

Overview: Smart Contracts Differently: Logic and Rules – A Primer

- Knowledge Bases
- Logic in general
  - Forward chaining
  - Backward chaining
- First Order Logic (FOL)
  - Why FOL?
  - Examples of FOL
  - Forward chaining
  - Backward chaining
- Logic programming

Pros and cons of propositional logic

☺ Propositional logic is *declarative*: pieces of syntax correspond to facts

☺ Propositional logic allows partial/disjunctive/negated information
(unlike most data structures and databases)

☺ Propositional logic is *compositional*:
meaning of $B_{1,1} \wedge P_{1,2}$ is derived from meaning of $B_{1,1}$ and of $P_{1,2}$

☺ Meaning in propositional logic is *context-independent*
(unlike natural language, where meaning depends on context)

☹ Propositional logic has very limited expressive power
(unlike natural language)
E.g., cannot say “insurance policies cost premiums” except by writing one sentence for each insurance

First-order logic

Whereas propositional logic assumes world contains ***facts***
first-order logic (like natural language) assumes the world contains

- ***Objects***: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries …
- ***Relations***: red, round, bogus, prime, multistoried, …,
  brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, …
- ***Functions***: father of, best friend, third inning of, one more than, beginning of …

Logics in general

Language	Ontological Commitment	EpistemologicalCommitment
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief$\in [0,1]$
Fuzzy logic	facts + degree of truth $\in [0,1]$	known interval value

FOL examples

- Brothers are siblings
  - $\forall x,y\ Brother(x,y) \Rightarrow Sibling(x,y)$
- "Sibling"' is symmetric
  - $\forall x,y\ Sibling(x,y) \Leftrightarrow Sibling(y,x)$
- One's mother is one's female parent
  - $\forall x,y\ Mother(x,y) \Leftrightarrow (Female(x) \wedge Parent(x,y))$
- A first cousin is a child of a parent's sibling
  - $\forall x,y\ FirstCousin(x,y) \Leftrightarrow \exists p,ps\ Parent(p,x) \wedge Sibling(ps,p) \wedge Parent(ps,y)$

Interacting with FOL KBs

- Suppose a "wumpus-world" agent is using an FOL KB and perceives a smell and a breeze (but no glitter) at *t=5*:
  - ***Tell****(KB,Percept([Smell,Breeze,None],5))*
  - ***Ask****(KB, ∃a Action(a,5))*
    I.e., does the KB entail any particular actions at *t=5*?
  - Answer: *Yes, {a/Shoot}* ← *substitution* (binding list)
- Given a sentence *S* and a substitution σ,
  - *S*σ denotes the result of plugging σ into *S*; e.g.,
    - *S = Smarter(x,y)*
    - σ = *{x/Hillary, y/Bill}*
    - *S*σ = Smarter(Hillary, Bill)
  - ***Ask****(KB,S)* returns some/all σ such that *KB* |= σ

Example knowledge base

– The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

– Prove that Col. West is a criminal

Example knowledge base contd.

… is a crime for an American to sell weapons to hostile nations:

$American(x) \wedge Weapon(y) \wedge Sells(x,y,z) \wedge Hostile(z) \Rightarrow Criminal(x)$

Nono … has some missiles, i.e., $\exists\, x : Owns(Nono,x) \wedge Missile(x)$

$Owns(Nono,M_1) \wedge Missile(M_1)$

… all of its missiles were sold to it by Colonel West

$\forall\, x: Missile(x) \wedge Owns(Nono,x) \Rightarrow Sells(West,x,Nono)$

Missiles are weapons:

$Missile(x) \Rightarrow Weapon(x)$

An enemy of America counts as “hostile”:

$Enemy(x,America) \Rightarrow Hostile(x)$

West, who is American …

$American(West)$

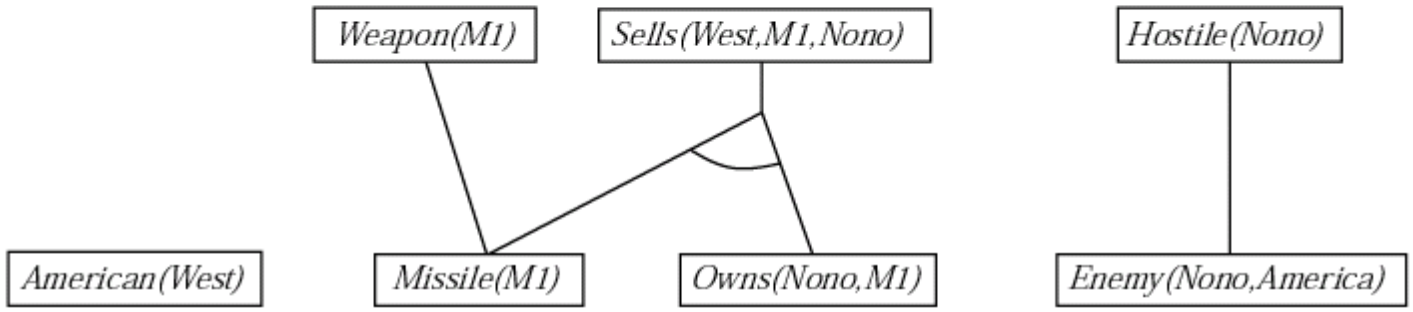
The country Nono, an enemy of America…

$Enemy(Nono,America)$

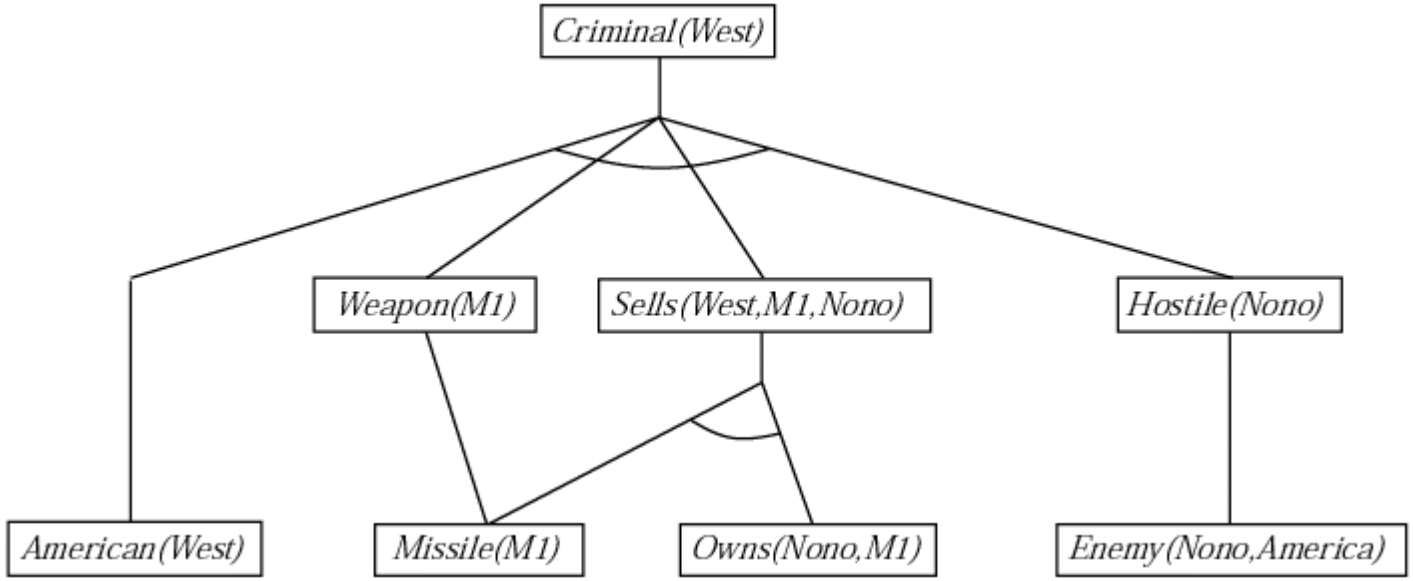
Forward chaining proof

American(West) *Missile(M1)* *Owns(Nono,M1)* *Enemy(Nono,America)*

Forward chaining proof



Forward chaining proof



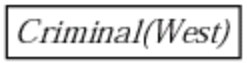
Properties of forward chaining

- Sound and complete for first-order definite clauses (proof similar to propositional proof)
- *Datalog* = first-order definite clauses + *no functions*
  (e.g., crime KB)
- FC terminates for Datalog in poly iterations: at most $p \cdot n^k$ literals
- **May not terminate in general if α is not entailed**
- This is unavoidable: entailment with definite clauses is semidecidable

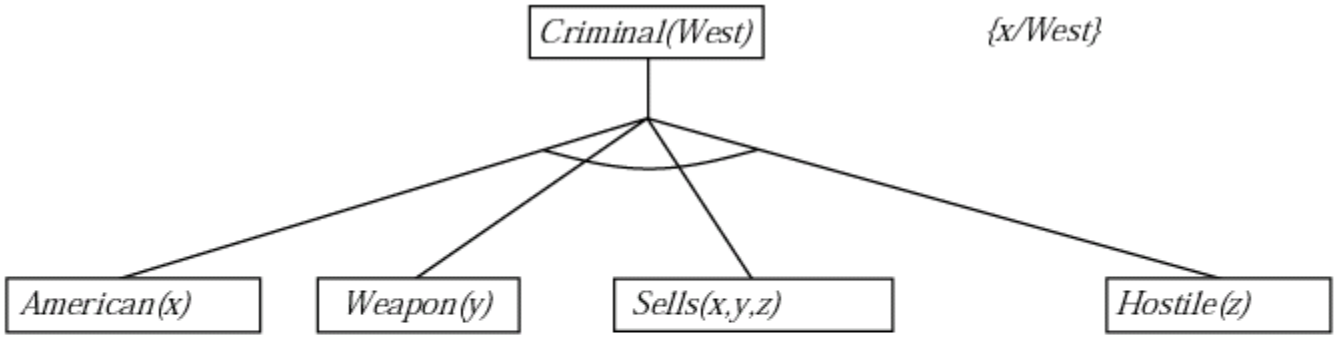
Efficiency of forward chaining

- Simple observation: no need to match a rule on iteration *k*
- if a premise wasn't added on iteration *k-1*
  $\Rightarrow$ match each rule whose premise contains a newly added literal
- Matching itself can be expensive
- *Database indexing* allows *O(1)* retrieval of known facts
  - e.g., query Missile(x) retrieves Missile(M_1)
- Matching conjunctive premises against known facts is NP-hard
- Forward chaining is widely used in *deductive databases*

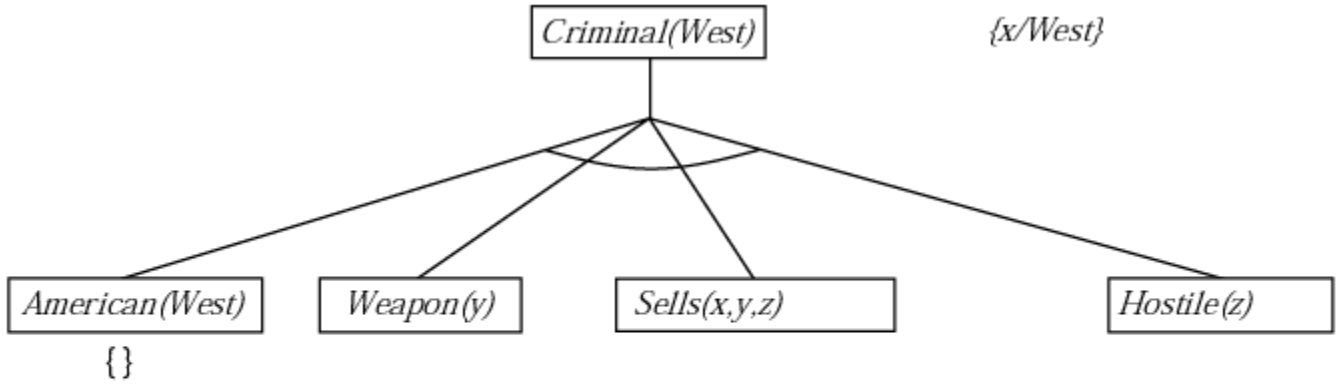
Backward chaining example



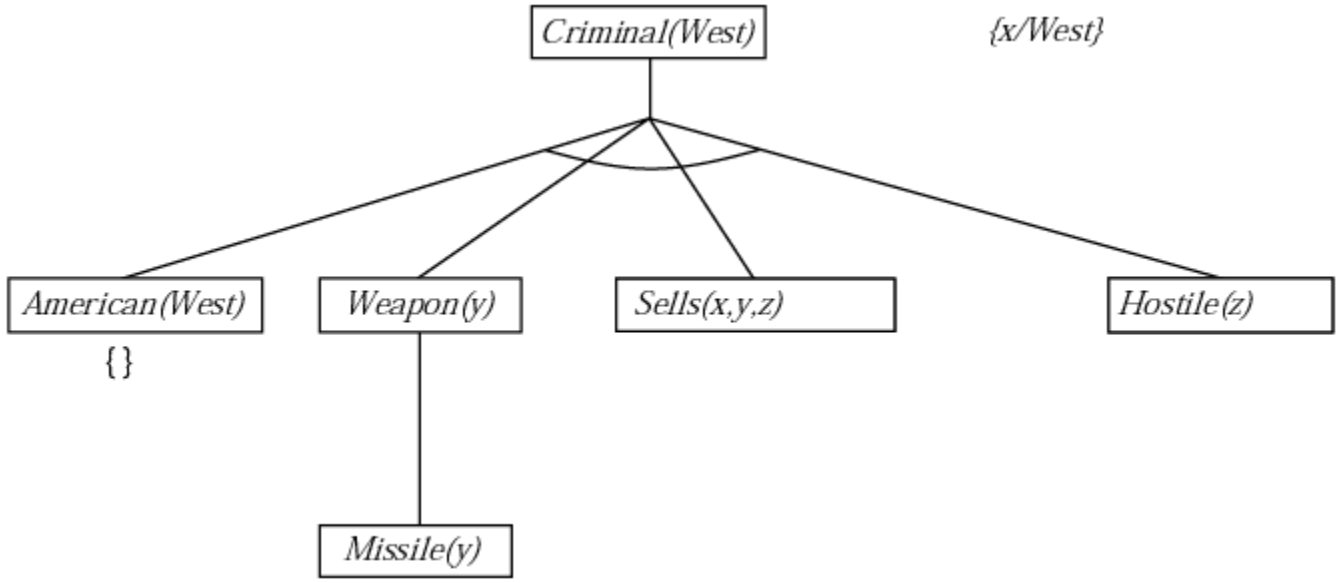
Backward chaining example



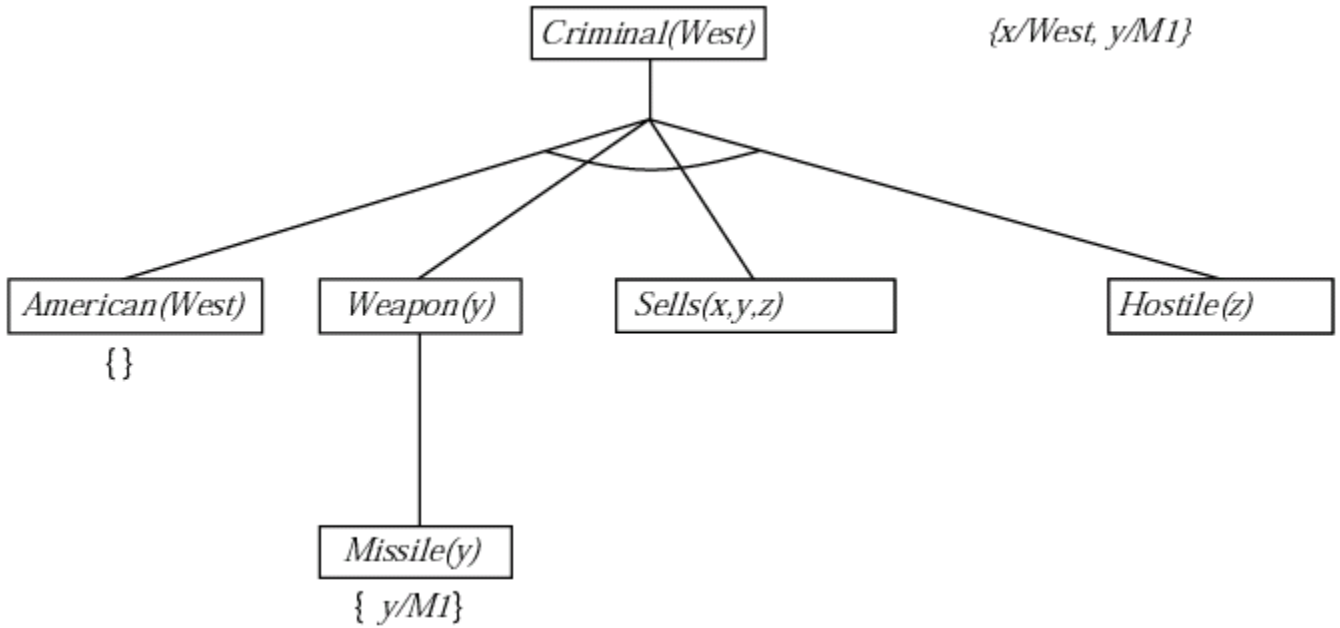
Backward chaining example



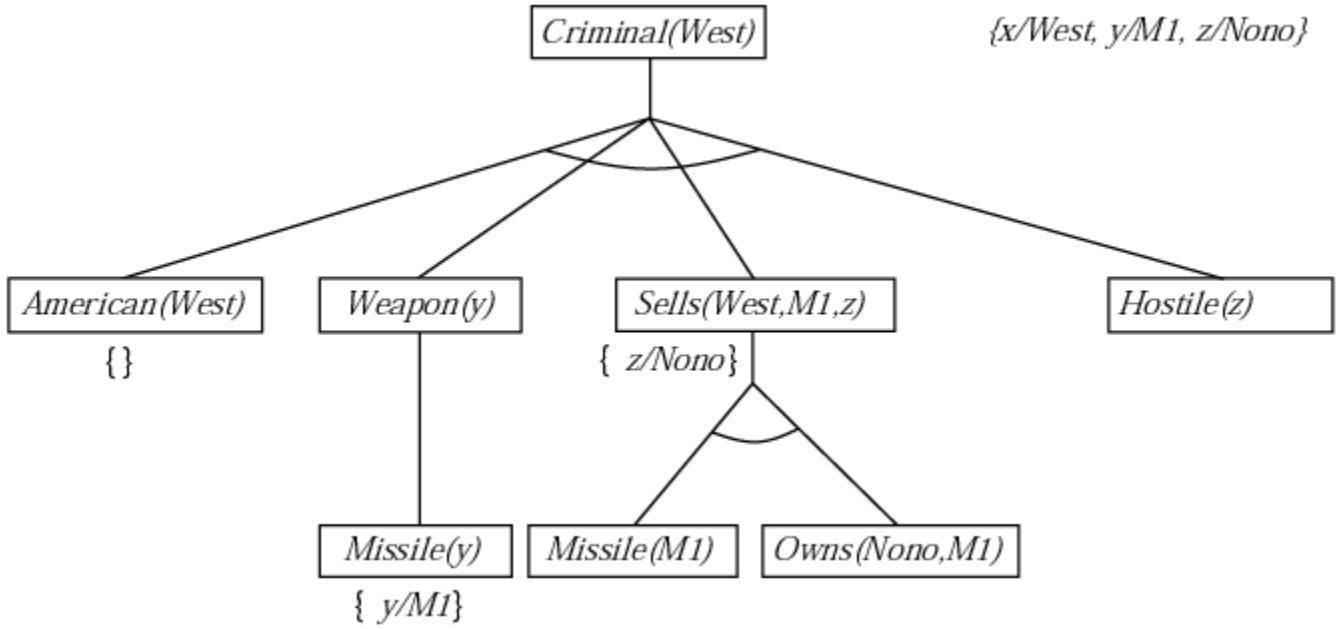
Backward chaining example



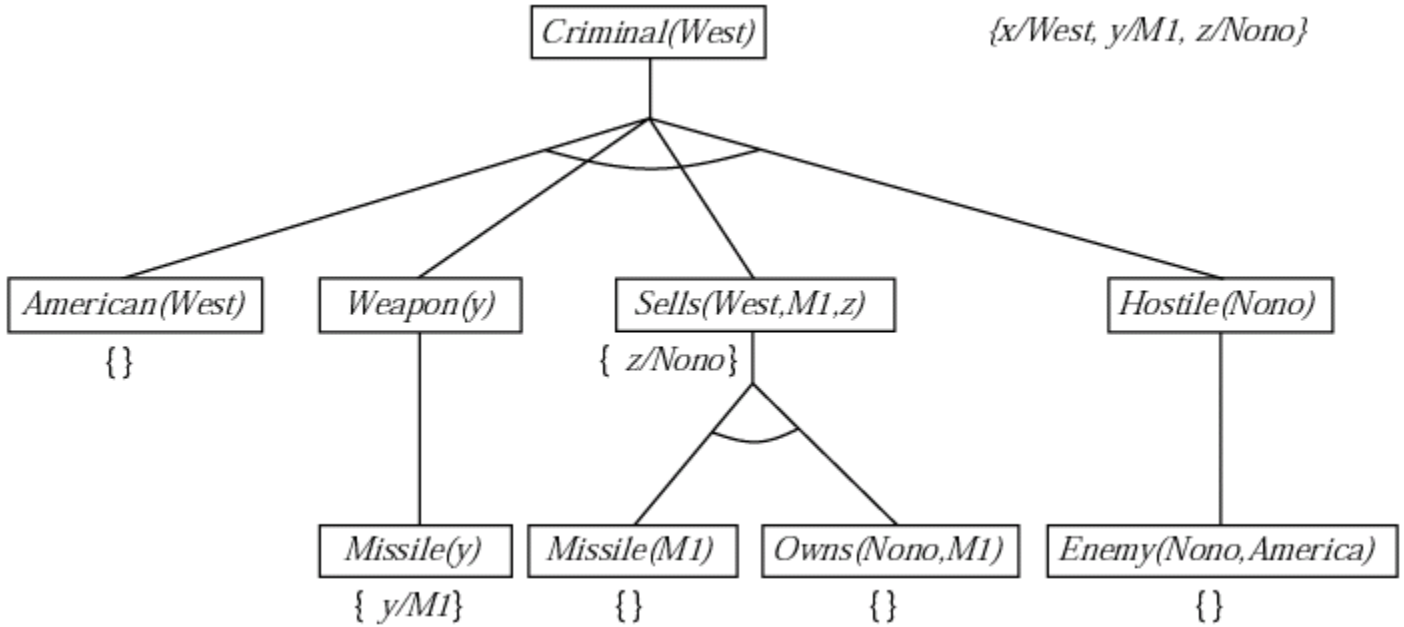
Backward chaining example



Backward chaining example



Backward chaining example



Properties of backward chaining

- Depth-first recursive proof search: space is linear in size of proof
- **Incomplete due to infinite loops**
  $\Rightarrow$ fix by checking current goal against every goal on stack
- Inefficient due to repeated subgoals (both success and failure)
  $\Rightarrow$ fix using caching of previous results (extra space!)
- Widely used (without improvements!) for *logic programming*

Overview: Smart Contracts Differently: Logic and Rules – A Primer

- Knowledge Bases
- Logic in general
  - Forward chaining
  - Backward chaining
- First Order Logic (FOL)
  - Why FOL?
  - Examples of FOL
  - Forward chaining
  - Backward chaining
- Logic programming

Logic programming

Sound bite: computation as inference on logical KBs

	Logic programming	***Ordinary programming***
1.	Identify problem	Identify problem
2.	Assemble information	Assemble information
3.	Tea break	Figure out solution
4.	Encode information in KB	Program solution
5.	Encode problem instance as facts	Encode problem instance as data
6.	Ask queries	Apply program to data
7.	Find false facts	Debug procedural errors

Should be easier to debug *Capital(NewYork,US)* than *x:= x+2*!

Prolog systems

- Basis: backward chaining with Horn clauses + bells & whistles
- Widely used in Europe, Japan (basis of 5th Generation project)
- Compilation techniques $\Rightarrow$ 60 million LIPS
- Program = set of clauses = **`head :- literal`**$_1$**`, …, literal`**$_n$**`.`**

  `criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).`
- Efficient unification by *open coding*
- Efficient retrieval of matching clauses by direct linking
- Depth-first, left-to-right backward chaining
- Built-in predicates for arithmetic etc., e.g., **`X`** is **`Y*Z+3`**
- Closed-world assumption (“negation as failure”)
  e.g., given **`alive(X) :- not dead(X). alive(joe)`** succeeds if **`dead(joe)`** fails

Prolog examples

– Depth-first search from a start state X:

```
dfs(X) :- goal(X).
dfs(X) :- successor(X,S),dfs(S).
```

– No need to loop over S: **successor** succeeds for each

– Appending two lists to produce a third:

```
append([],Y,Y).
append([X|L],Y,[X|Z]) :- append(L,Y,Z).

query:   append(A,B,[1,2]) ?
answers: A=[]    B=[1,2]
         A=[1]   B=[2]
         A=[1,2] B=[]
```

Rule-based systems in blockchain

Slides based on:
- Lity, 2019

Logic rules and blockchain

- Programming through logic rules moves the smart contract development from an imperative to a declarative paradigm
  - There are theoretical guarantees on the complexity of the smart contract (e.g. will the execution end?)
- There are a number of projects which are trying to combine these two worlds, e.g.
  - Lity: an extension of Solidity with support to logic rules
  - Wonka: a rule engine for Nethereum (a .NET library for Ethereum)

Lity

- We use Lity as a good example of how blockchain can support and enable rules
- Lity extends Solidity
  - Every Solidity program is a Lity program, but not the opposite
  - It combines imperative and declarative programming
  - Lity does not run on Ethereum nodes – it uses functions which are not supported
    - It requires CyberMiles, a blockchain infrastructure developed by the CyberMiles startup

The Lity smart contracts

- We can ideally see a Lity smart contract as a smart contract embedding a knowledge base
  - When the knowledge base is empty, we have a normal smart contract (compatible with Solidity)
- We can make three operations on the knowledge base
  - Adding facts
  - Removing facts
  - Executing rules on top of it

Facts

– Any variable where the data type is defined through `struct` can be a fact

```
struct Person {
        address a;
        bool eligible;
}
```

– We can insert new facts in the knowledge base with the Lity command `factInsert`

```
factInsert Person(0x1ED13B2108B2c43e7e6Dbf7f089205A3A3b2A69D, true);
factInsert Person(0x3A3b2B2cA69D843bf7f089205Ae7e6D1ED13B210, false);
factInsert Person(0x13B210cA3bf7Ae7e6D1Ef089269D843A3b05D2B2, false);
```

– And remove facts with the Lity command factDelete

```
factDelete Person(0x13B210cA3bf7Ae7e6D1Ef089269D843A3b05D2B2, false);
```

Rule structure

- Defining a rule in Lity requires three components:
  - A name
  - A body with a set of patterns
  - A head with the operations to be executed, should the body find a match

```
rule “name” when {
        // body patterns
} then {
        // head operations
}
```

Patterns

– A body pattern is defined by an identifier, a `struct` data type, and a set of conditions on the contained variables

```
p: Person(eligible == true)
```

– This pattern matches all the facts stored in the knowledge base of type Person, where the eligible variable is set true

– p identifies the matching fact

Rule

– We have now all the elements to write a Lity rule:

```
rule "pay" when {
        p: Person(eligible == true);
} then {
        p.a.transfer(10);
        p.eligible = false;
}
```

– We can start the rule execution with the command `fireAllRules`

Competing rules

- What does It happens when a smart contract has two or more rules working on the same data?
  - By default, there is not a predefined order on which rules are executed
  - We may end up with a result very different from the expected one!
- A usual way to solve this issue is to prioritise rules
  - Rules with higher priority will be executed first
  - Lity implements this strategy through the salience keyword
    - Rules with high salience will be executed first

```
rule “rule1” salience 10 when { ...} then { ... }
rule “rule2” salience 30 when { ...} then { ... }
```

Use Case 3: Lottery with Rules (see Example3.lity)

A rule-based auction on blockchain

– Start from Example3.0.lity and:

1. Prepare the facts that needs to be stored in the knowledge base. Remember that fact must be represented by a struct variable
2. Add the facts in the knowledge base
3. Prepare the rule to pay the winners
4. Prepare the rules to give back to the owner the extra money. Remember to set priority among rules, if needed!

Where to go from here…

Websites, Books, and other Interesting Sources (1)

[Akhtar et al., 2019] N. Akhtar, R. Lin and M. Wang. Blockchain Fundamentals, UC Berkeley, 2019 https://www.edx.org/professional-certificate/uc-berkeleyx-blockchain-fundamentals

[Antonopoulos & Wood, 2018] A. M. Antonopoulos and G. Wood. Mastering Ethereum, O'Reilley, 2018. https://github.com/ethereumbook/ethereumbook

[Coulouris et al, 2012] G. Coulouris. Distributed Systems: Concepts and Design, Addison Wesley, 2012.

[Drools, 2020] Drools: https://docs.jboss.org/drools/release/7.1.0.Final/drools-docs/html_single/#_rule_engines_and_production_rule_systems_prs

[Lity, 2019] Lity: https://www.litylang.org & https://www.litylang.org/getting_started/

Websites, Books, and other Interesting Sources (2)

[Ramamurthy, 2019] B. Ramamurthy. Blockchain Specialization, The University at Buffalo, 2019. https://www.coursera.org/specializations/blockchain

[Russel & Norvig, 2009] S. Russell & P. Norvig. Artificial Intelligence: A Modern Approach, Pearson, 2009.

[Solidity, 2019] Solidity documentation: https://solidity.readthedocs.io/en/v0.5.11/index.html

[Tanenbaum & van Steen, 2007] A. Tanenbaum and M. van Steen. Distributed systems: Principles and Paradigms, Pearson, 2007.

[Wonka, 2020]: https://github.com/Nethereum/Wonka