Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

CAIDAS WüNLP

ALGORITHMS IN AI & DATA SCIENCE 1 (AKIDS 1)

Expert Systems

Prof. Dr. Goran Glavaš

22.1.2024

Content

- Knowledge-Based AI
- Expert Systems
- Inference

Based on the materials from Prof. Dr. Jan Šnajder:
https://www.fer.unizg.hr/_download/repository/AI-2022-08-ExpertSystems.pdf

Motivation: An Intelligent Agent

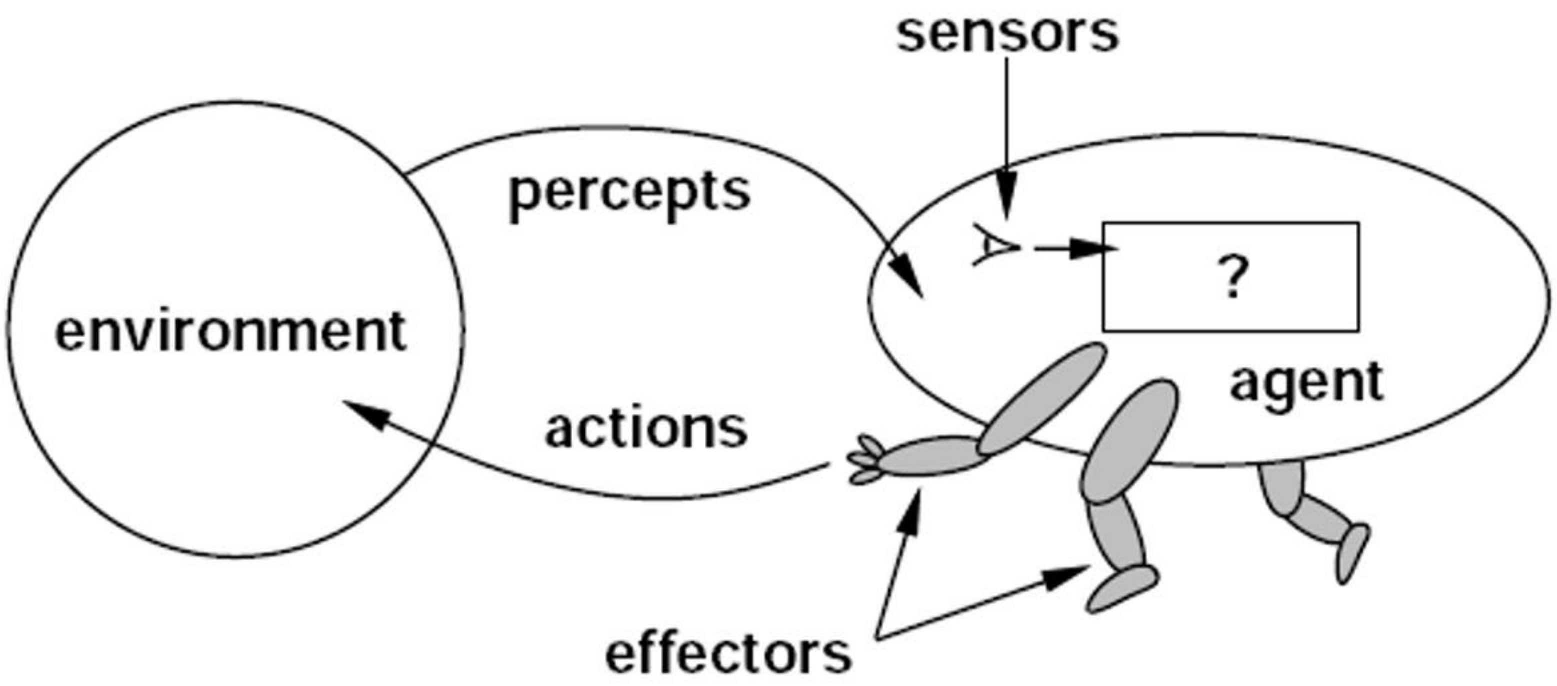


Image from *Russel & Norving. Artificial Intelligence: A Modern Approach*.

AI, Knowledge, and Reasoning

- Humans **know** things and what we know guides how we **do** things
- Human intelligence in big part stems from the ability to **reason** over the internal representations of **knowledge**
  - **Reasoning**: inference of new knowledge from existing knowledge

Knowledge-Based AI

Knowledge-based AI is the body of work in AI that revolves around **knowledge-based agents** which are equipped with two main components: **(1)** the **knowledge base** – a set of **„facts”**, represented in a particular format, and **(2)** the **reasoning engine/mechanism** – an algorithm or set of algorithms that allow for reasoning, i.e., induction of new knowledge from existing knowledge

Knowledge-Based AI

- **Knowledge base**: set of „facts" (sometimes called "sentences", but not in a language sense)
  - **Axioms**: facts taken as given and not derived from other facts
- **Facts** represented in a concrete **knowledge representation language**
  - Knowledge-based AI is sometimes also called **symbolic AI**
  - The KR language has a **vocabulary** – set of atomic elements of knowledge, typically some kind of **entities** and **relations** between them
- **Reasoning mechanism:** a set of rules or operations that **induce new facts** from the existing KB
  - Or check if some proposed facts are consistent with KB, that is, can be induced from the KB facts

Symbolism vs. Connectionism

- Traditional knowledge-based systems represent **symbolic AI**

Knowledge about the external world can be represented with symbols. **Inference** amounts to symbol manipulation. **Intelligent behavior** amounts to inference.

Symbolic AI / Symbolism

- Symbolism is contrasted by **connectionism**

Mental states and **behavior** emerges from the interaction of a large number of interconnected and simple processing units. An **artificial neural network** is a typical example of the connectionist approach to AI.

Connectionist AI / Connectionism

Symbolism vs. Connectionism

- **Symbolic AI** is discrete and inherently (human) interpretable
  - **Knowledge**: given as a KB
  - **Inference**: formal symbolic (rule-based) reasoning over KB
- **Connectionist (neural) AI** is continuous and (mostly) not human interpretable
  - **Knowledge**: learned from (large amounts of) raw data
  - **Inference**: computation in a continuous representation space

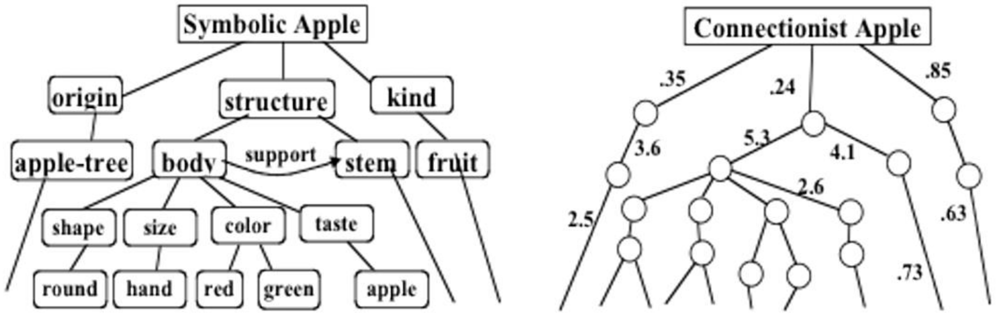


Image from: Minsky, M. (1990). Logical vs. Analogical or Symbolic vs. Connectionist or Neat vs. Scruffy. Artificial Intelligence at MIT. Expanding Frontiers, Patrick H. Winston (Ed.).

Some Knowledge Formalisms

- **Propositional logic**
- **Predicate Logic**
  (aka **First-Order Logic**)
- Temporal Logic
- Description Logic
  (basis of modern ontologies and knowledge graphs)
- Fuzzy Logic
- Modal Logic
- Epistemic Logic
- ...

Image from: https://sites.psu.edu/orenadamrcl/2012/09/30/rcl-4-logic-reduced/

Example: Propositional Logic

- **Symbols** of the propositional logic
  - Propositional variables (vocabulary, atomic formulae): V = {A, B, C, ...}
    - Each (A, B, ...) denotes one knowledge fact. For example, A = „*penguins are birds*"
  - Logical operators (or connectives)
    - Negation (¬), disjunction (OR, ∨), conjunction (AND, ∧)
    - Implication (→), equivalence (↔)
  - Logical (Boolean) constants *True* and *False*
  - Parentheses („(" and „)")

- **Knowledge** (KB) consists of **formulae**
  - Each variable is a **formula**
  - If F is a formula, then ¬F is also a formula
  - If F and G are formulas, then F ∨ G, F ∧ G, F → G, F ↔ G are also formulae

Example: Propositional Logic

- **Reasoning**: Based on the semantics of the logic operators
  - Infer if a formula is True (⊤) or False (⊥) from the truth values of atoms
  - Need **semantics** of the propositional logic

F	G	$\neg F$	$F \wedge G$	$F \vee G$	$F \rightarrow G$	$F \leftrightarrow G$
$\bot$	$\bot$	$\top$	$\bot$	$\bot$	$\top$	$\top$
$\bot$	$\top$	$\top$	$\bot$	$\top$	$\top$	$\bot$
$\top$	$\bot$	$\bot$	$\bot$	$\top$	$\bot$	$\bot$
$\top$	$\top$	$\bot$	$\top$	$\top$	$\top$	$\top$

- **New knowledge**: **logical consequence**
  - Formula G is a logical (semantic) consequence of formulae $F_1,\ldots,F_n$ if and only if every interpretation that satisfies $F_1 \wedge \cdots \wedge F_n$ also satisfies G.
  - In other words, if $F_1 \wedge \cdots \wedge F_n \rightarrow G$ is True for every intepretation

Example: Propositional Logic

Knowledge:

- **Atoms:** P = „Rain falls”; Q = „Cleaners hose the road”; R = „The road is wet”
- **Formulas (KB):**
  - $(P \lor Q) \rightarrow R$ („If rain falls or cleaners hose, the road becomes wet”)
  - R („road is wet”)
  - $\neg P$ („the rain didn’t fall”)

- **Logical inference:**
  - $(((P \lor Q) \rightarrow R) \land R \land \neg P) \rightarrow Q$?

Content

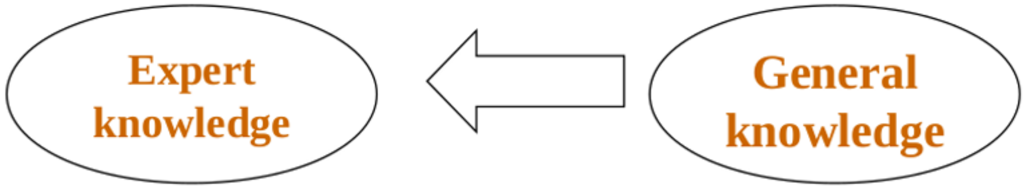
- Knowledge-Based AI
- Expert Systems
- Inference

Expert Systems

- A symbolic AI paradigm for **knowledge representation** and **reasoning**
- Very popular in the 80s – originated from the idea that the majority of human knowledge can be represented in the form of **if-then rules**
  - „If patient's temperature is above 38°C, medications that lower the body temperature should be administered"
  - „If the traffic light is red, then stop"
- First practically successul „AI technology": machines giving an impression of „analyzing and thinking"

General vs. Expert Knowledge

- Obviously, it is impossible to come up with exhaustive if-then rules for all domains of human activity and knowledge
- Impossible to encode **general knowledge** with if-then rules



- **Solution**: narrow down the scope to a specific domain
  - For example: medicine, finances, chess, ...
- Expert systems do not tackle **general problem solving**

Expert systems = Intellectual Cloning

- The overall intent behind **expert systems** is that of **intellectual cloning**

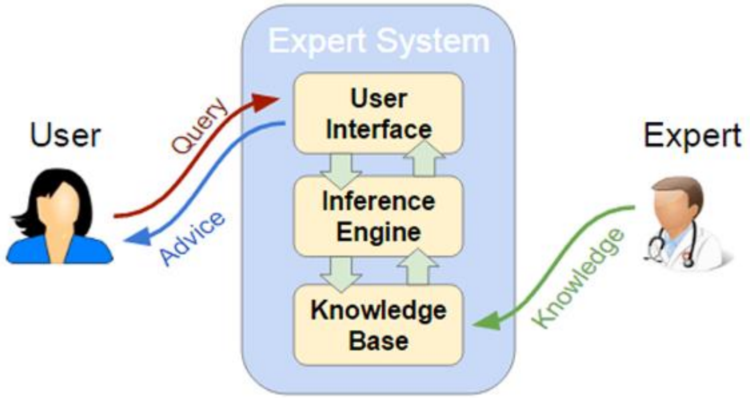
- Find people that have a **reasoning skill** that is important and rare
  - Expert medical diagnostician
  - Expert business analyst

- **Analyze / extract** their knowledge and reasoning and try to embody them in a program
  - In case of **Expert Systems**: as if-then rules

Knowledge base vs. Inference Engine

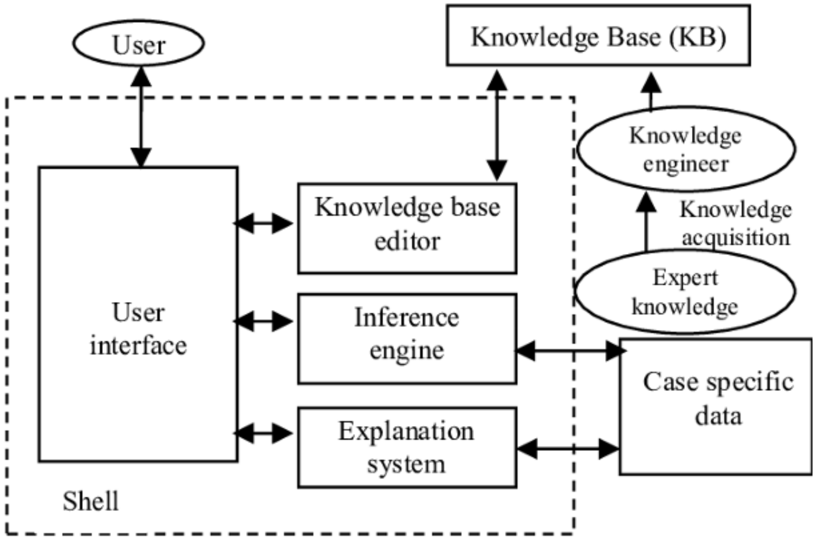
- Different expert systems have differing representation technologies, but all have **two main architectural properties**

1. Distinction between **inference engine** and **knowledge base**
    - IE retrieves rules from the KB
2. Use of **declarative style representations**
    - Rules are data structures with their own semantics, rather than part of the code implementing the inference engine



Expert Systems shell

- **Inference engine** is decoupled from the **knowledge base** → the idea is that IE can operate on any KB that is „plugged in”
- **Expert system shell**: a tool for building expert systems
  - Inference engine
  - Knowledge base editor
  - User interface
  - Explanation module



If-then rules

- Knowledge in ES is represented by the so-called **production rules**
  - Essentially if-then rules
  - **If** [condition/state/premise/antecedent]
    then [action/conclusion/consequent]
- It's quite reminiscent of **implication** in logic (A → B), but there are two key differences
  - In logic, **implication** is a **formula** and as such has a truth value
  - The consequent in implication (B in A → B) is also a formula, whereas the **consequent** in if-then rules of an ES are **actions**
    - Asserting new facts but also
    - **Deleting** facts, executing code, printing on screen, ...

Content

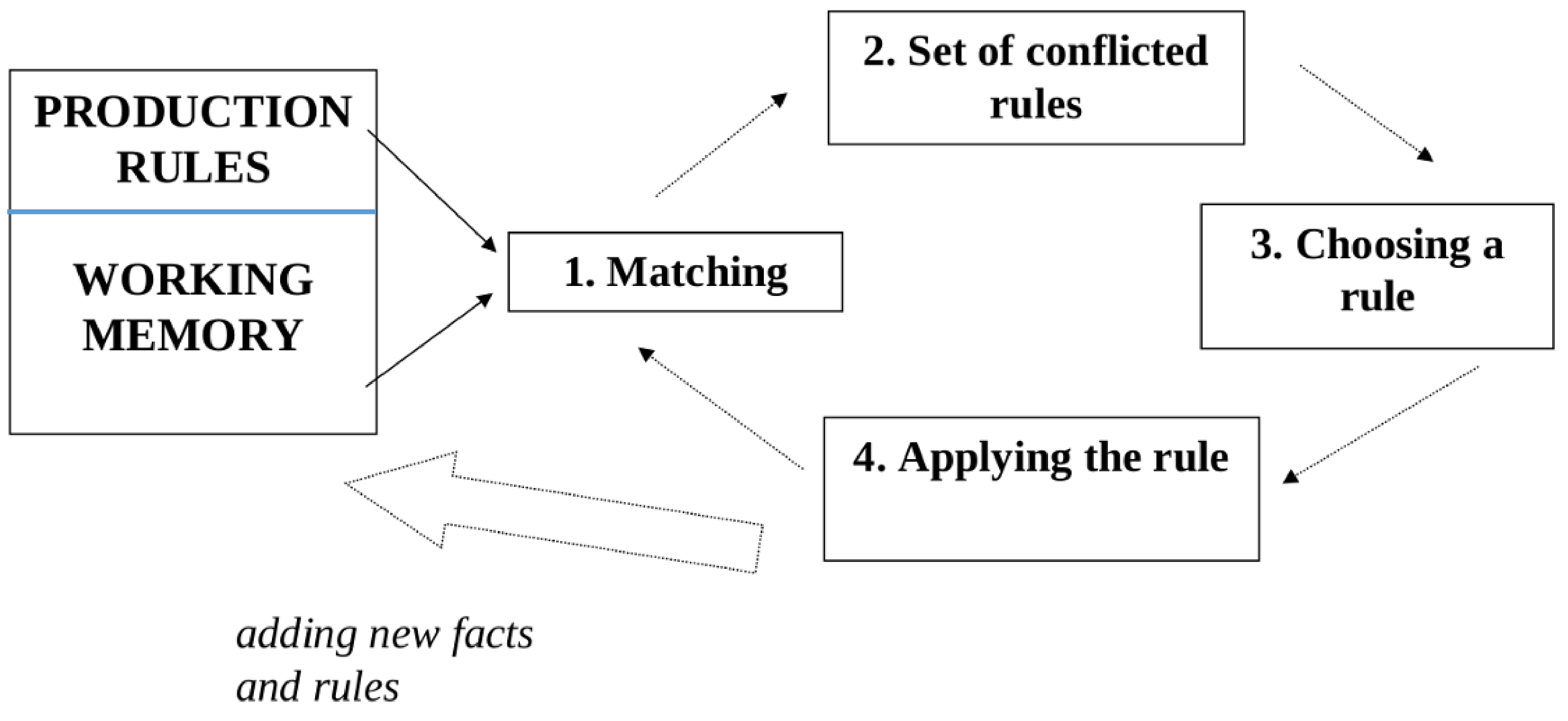
- Knowledge-Based AI
- Expert Systems
- Inference

Inference Components

- **Working memory –** part of the knowledge base that:
  - Stores facts (i) added by the user before inference or (ii) new facts derived during inference
  - Does not store them permanently (akin to short-term memory in humans)

- **Inference engine** – a control mechanism carrying out the following:
  - **Matching** – facts from the **working memory** need to be matched against the left-hand side (LHS or condition) of the if-then rules
  - **Conflict resolution** – if the working memory matches the LHS of more than one rule, need to **select** one of the rules based on some criteria
  - **Rule application** (aka „rule firing”) – executing the action specified by the right-hand side of the rule whose LHS was matched

Inference cycle



Inference in Rule-Based Systems

- Establish a **reasoning chain** which is a sequence of conclusions that link the starting condition to the solution of the problem
  - The reasoning procedure is called **chaining**
- **Forward chaining**
  - Starting with known data and advancing toward a conclusion
  - **To use**: when there is a **small amount of data** and a large space of possible solutions
- **Backward chaining**
  - Choosing a possible conclusion (hypothesis) and trying to prove that it is valid by finding valid evidence
  - **To use:** Not too many possible conclusions, the amount of known data is large
- **Bidirectional inference**
  - Combines forward and backward chaining

Factorization –Variables and Values

- If-then rules in ES will operate on a set of **variables**, each with a **domain**
  - Similar like in Discrete Optimization and Constraint Satisfaction
- The variables and their domains can be referred to as **ontology** of the expert system
  - $O = X_1, X_2, \ldots, X_n$,

    $X_1 \in D_1, X_2 \in D_2, \ldots, X_2 \in D_n$
  - **Rules format**
    - **If** $X_i == x_i$ **and** $X_j == x_j$ **and … and** $X_k == x_k$ **then** $X_m = x_m$

Example

- **Knowledge base** for determining type of fruit
- Ontology (**variables** and **possible values**):
  - **Shape**: elongated | circular | rounded
  - **Surface**: smooth | coarse
  - **Color**: green | yellow | brown-yellow | red | blue | orange
  - **No. seeds:** 0 | 1 | >1
  - **Seed type**: multiple | bony
  - **Diameter**: <10cm | >10cm
  - **Fruit type**: vine | tree
  - **Fruit**: banana | watermelon | cantaloupe | apple | appricot | cherry | peach | plum | orange

Example

- **Knowledge base** for determining type of fruit
- **If-then rules**
  - R_1: **IF Shape** = elongated & **Color** = green | yellow **THEN Fruit =** banana
  - R_2: **IF Shape** = circular | rounded & **Diameter** = >10cm **THEN Fruit Type =** vine
  - R_3: **IF Shape** = circular & **Diameter** = <10cm **THEN Fruit Type =** tree
  - R_4: **IF No. Seeds** = 1 **THEN Seed Type =** bony
  - R_5: **IF No. Seeds** = >1 **THEN Seed Type =** multiple
  - R_6: **IF Fruit type** = vine & **Color** = green **THEN Fruit =** watermelon
  - R_7: **IF Fruit type** = vine & **Color** = yellow & **Surface** = smooth
    THEN Fruit = melon

Example

- R_8: **IF Fruit type** = vine & **Color** = brown-yellow & **Surface** = course
  THEN Fruit = cantaloupe
- R_9: **IF Fruit type** = tree & **Color** = orange & **Seed Type** = bony
  THEN Fruit = apricot
- R_{10}: **IF Fruit type** = tree & **Color** = orange & **Seed Type** = multiple
  THEN Fruit = orange
- R_{11}: **IF Fruit type** = tree & **Color** = red & **Seed Type** = bony
  THEN Fruit = cherry
- R_{12}: **IF Fruit type** = tree & **Color** = orange & **Seed Type** = bony
  THEN Fruit = peach
- R_{13}: **IF Fruit type** = tree & **Color** = yellow | green & **Seed Type** = multiple
  THEN Fruit = apple
- R_{14}: **IF Fruit type** = tree & **Color** = blue & **Seed Type** = bony
  THEN Fruit = plum

Forward Chaining: Example

- **Input (known) data**:
  - **Diameter** = 2cm (<10cm), **Shape**: circular, **No. seeds: 1**, **Color**: red
- **Conflict resolution**: take the rule with smaller number

Step	Working memory	Conflicting rules	Rule that fires
0	**Diameter** = <10cm **Shape** = circular **No. seeds** = 1 **Color** = red	R3, R4	R3 (smaller number)
1	+ **Fruit Type** = tree	~~R3~~, R4	R4
2	+ **Seed Type** = bony	~~R3~~, ~~R4~~, R11	R11
3	+ **Fruit** = cherry	~~R3~~, ~~R4~~, ~~R11~~	**DONE**

Backward Chaining

- Starts with a desired goal (hypothesis) and determines whether the existing facts support proving the goal

- Start with an empty list of facts, the **goal variable** is given
  - We start from all rules that assign a value to the goal variable, and check what is on the LHS
  - If on LHS we have a variable for which we don't have the value yet either, we try to infer it → look for all rules with that variable on LHS, etc.

  - **Last in first out** principle of trying to figure out values for variables
    - **Q:** Which data structure do we need then?

Backward Chaining: Steps

Step 1. Put the **goal variable** onto the (empty) stack

Step 2. Top of stack always the variable for which we need to find the value. Find all rules with the variable from the stack top on **RHS**

- If no rule has the stack-top variable on the RHS, **ask the user**

Step 3. For each such rule:

3a. If LHS satisfied (all variables have correct values in WM),

- apply the rule (place the RHS variable and value into WM)
- remove the curent goal from the stack,
- continue from Step 2

Backward Chaining: Steps

Step 1. Put the **goal variable** onto the (empty) stack

Step 2. Top of stack always the variable for which we need to find the value. Find all rules with the variable from the stack top on **RHS**

- If no rule has the stack-top variable on the RHS, **ask the user**

Step 3. For each such rule:

3b. If LHS not satisfied because of different value of some variable compared to WM, do not apply the rule

3c. If LHS not satisfied because the value of some variable is not in WM at all, then add that variable to the stack

Backward Chaining: Example

- Our fruit example → the goal variable is **fruit**

Step	Stack	Working memory	Conflicting rules	Action
0	**Fruit**		**R1**, R6, R8, R9, R10, R11, R12, R13, R14	**Shape** (LHS of **R1**) not in WM and not on RHS of any rule, ask user
1	**Fruit**	**Shape** = circular	**R6**, R8, R9, R10, R11, R12, R13, R14	**Fruit Type** (LHS of **R6**) not in WM but exists on RHS of rules, add to stack
2	**Fruit Type** **Fruit**	**Shape** = circular	**R2**, R3 (**Fruit Type** on RHS)	**Diameter** (LHS of **R2**) not in WM and not on RHS of any rule, ask user
3	**Fruit Type** **Fruit**	**Shape** = circular **Diameter** = <10cm	R3	LHS of R3 is satisfied (all variables with correct values in WM), add RHS to WM and pop the stack

Backward Chaining: Example

Step	Stack	Working memory	Conflicting rules	Action
4	**Fruit**	**Shape** = circular **Diameter** = <10cm **Fruit Type** = tree	**R6**, R8, R9, R10, R11, R12, R13, R14	The LHS of R6 is in conflict with WM, proceed to next rule
5	**Fruit**	**Shape** = circular **Diameter** = <10cm **Fruit Type** = tree	**R8**, R9, R10, R11, R12, R13, R14	The LHS of R8 is in conflict with WM, proceed to next rule
6	**Fruit**	**Shape** = circular **Diameter** = <10cm **Fruit Type** = tree	**R9**, R10, R11, R12, R13, R14	The LHS of R9 has **Color** which is not in WM, and not in RHS of any rule, ask user
7	**Fruit**	**Shape** = circular **Diameter** = <10cm **Fruit Type** = tree **Color** = red	**R11**	The LHS or R11 has Seed Type which is not in WM but exists in RHS of another rule, push **Seed Type** to stack

Backward Chaining: Example

Step	Stack	Working memory	Conflicting rules	Action
8	**Seed Type** **Fruit**	**Shape** = circular **Diameter** = <10cm **Fruit Type** = tree **Color** = red	**R4**, R5	R4 has **No. Seeds** on LHS, which we don't have in WM nor do we have any rules with it on RHS, ask user
9	**Seed Type** **Fruit**	**Shape** = circular **Diameter** = <10cm **Fruit Type** = tree **Color** = red **No. Seeds** = 1	**R4**, R5	LHS of R4 is satisfied, we add the RHS to WM and pop the stack
10	**Fruit**	**Shape** = circular **Diameter** = <10cm **Fruit Type** = tree **Color** = red **No. Seeds** = 1 **Seed Type =** bony	R11	LHS of R11 is satisfied, add the RHS (**Fruit** = cherry) to WM and pop the stack → stack will be empty → **DONE**

Backward Chaining: Algorithm

- Let's write the pseudocode for **backward chaining**, using appropriate data structures and in a **modular fashion**!

- **Data structures:**
  - **Q:** How to represent the ontology (variables and allowed values for each)?
  - **Q:** How to represent the working memory?
  - **Q:** How would you represent a rule?

  - **Ontology** and **rules** are static
  - **Working memory** changes, but can only grow
    - And we know its maximal size (number of variables) in advance

  - A **lot of „reading"** into all three, not much writing
  - No similarity or neighbourhood required (*min*, *max*, *previous*, *next*, ...)

Backward Chaining: Algorithm

- **Ontology**
  - **hash table** of **hash tables**
  - **Keys:** variables, **Values:** hash table with allowed values for the variable
  - When user provides a value, we need to check if it's allowed for the variable
- **Rule:** has LHS and RHS, assume RHS always has only one variable
  - **LHS**: **hash table** (Key: variable, Value: value)
  - **RHS:** pair (tuple) – variable, value
- **Working Memory**: **hash table**

```
value_valid(ont, var, val)
   vals = ont[var]
   if val in vals # hashtable lookup
     return True
   else
     return False

rule_status(rule, wm)
  for var in rule.LHS
    if var not in wm
      return var # not in wm
    elif rule.LHS[var] ≠ wm[var]
      return False # in wm, wrong val
  return True

apply_rule(rule, wm)
  var = rule.RHS.var
  val = rule.RHS.val
  wm[var] = val
```

Backward Chaining: Algorithm

```
backward_chain(ont, rules, goal)
  s = [] # empty stack
  s.push(goal)
  wm = {} # empty hash table

  while not s.is_empty()
    goal = s.peek()
    matches = find_rules(rules, goal)

    if len(matches) == 0 # no rule with stack-top variable on RHS
      val = ask_user(goal)
      if value_valid(ont, val, goal)
        wm[goal] = val
      else
        return „error”

    for m in matches
      status = rule_status(m, wm)
      if status == True # LHS satisfied
        apply_rule(m, wm) # RHS added to wm
        s.pop()
        break
      elif status == False # LHS in conflict with wm
        continue
      else # status is a variable not in wm
        s.push(status)
        break
  return wm[goal]
```

```
find_rules(rules, goal)
  matches = []
  for rule in rules
    if rule.RHS.var == goal
      matches.append(rule)
  return matches
```

- Execution **stack**
  - Function `peek` just reads the value from the top, without removing it
- This basic variant of the algorithm is quite inefficient
  - **Q:** How would you speed it up?
- **Q:** how would you implement backchaining without (an explicit) stack?

Questions?

ਸਵਾਲ? Küsimusi?
Turite klausimų?
Awọn ibeere?
Pitanja? Sorusu olan? Dúvidas? 質問は？
有问题吗？ Fragen?
¿Preguntas?
Frågor? Questions? Pytania? Domande?
Питання? Vragen? Ερωτήσεις;
Porandukuéra? คำถาม? أسئلة؟