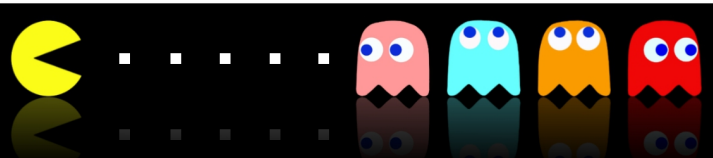
Advanced Game AI

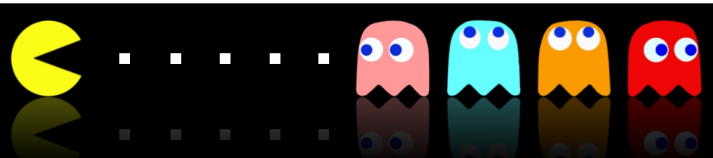
Level 3 – Rule Based AI

Prof Alexiei Dingli



Rule-based AI

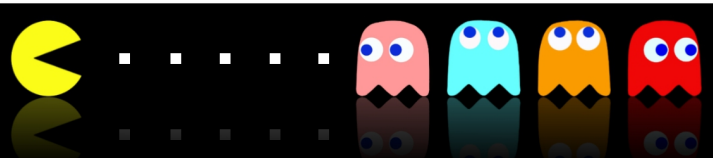
- Generally refer to AI systems that consist of a set of if-then (or if-else) style rules
- Technically, FSMs and DTs are types of rule-based systems since rules are used to handle state transitions and decision nodes
- But more specifically, **“rule-based systems”** are also commonly referred to its usage in **expert systems**



Rule-based Expert Systems

- Common usages/applications in real life:
  - Medical diagnosis, fraud protection, etc

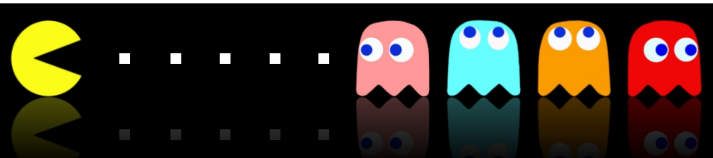
- Advantage:
  - Rule-based systems can mimic the way people **think** and **reason** given a set of known facts and knowledge about a particular domain
  - Fairly easy to program and manage (in a computer application)

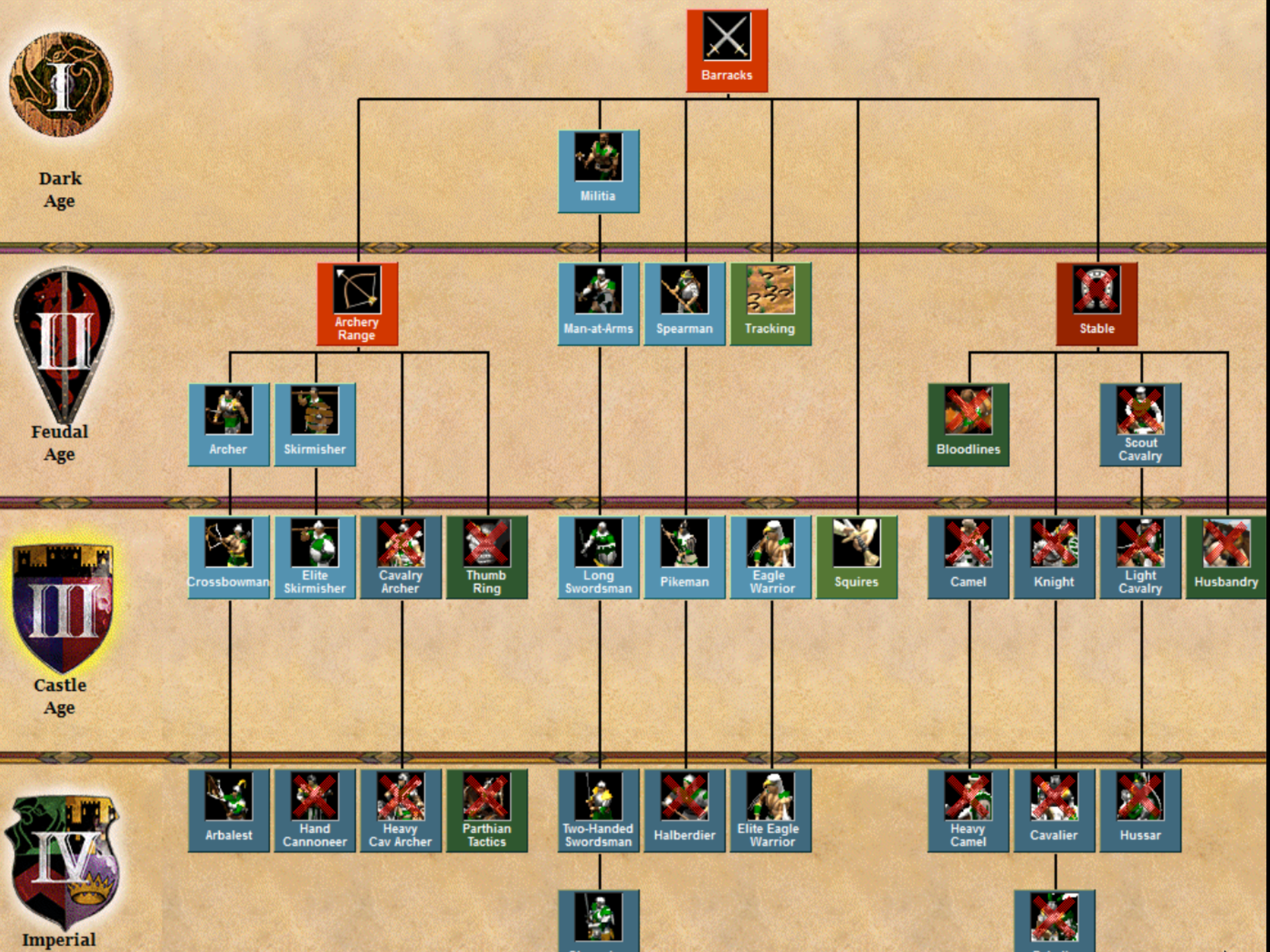


Rule-based Systems for Games

- Rule-based systems are useful in GAMES…
  - Because knowledge encoded in rules is modular
  - Rules can be encoded in any order → flexible for coding and modifying the system at a later time

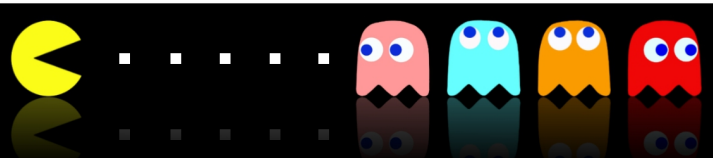
- Let's look at a game example that can use a rule-based system…




Barracks
Militia
Dark Age
Archery Range
Man-at-Arms
Spearman
Tracking
Stable
Archer
Skirmisher
Bloodlines
Scout Cavalry
Feudal Age
Crossbowman
Elite Skirmisher
Cavalry Archer
Thumb Ring
Long Swordsman
Pikeman
Eagle Warrior
Squires
Camel
Knight
Light Cavalry
Husbandry
Castle Age
Arbalest
Hand Cannoneer
Heavy Cav Archer
Parthian Tactics
Two-Handed Swordsman
Halberdier
Elite Eagle Warrior
Heavy Camel
Cavalier
Hussar
Imperial

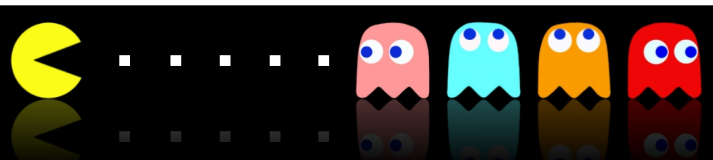
Example: RTS game

- **Technology Tree**
  - An important element in RTS games
  - Shows the links between units to train, facilities to build and resources to harvest in order to expand influence in game



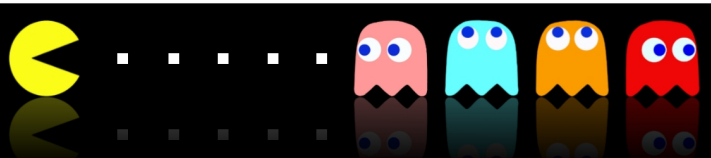
Example: RTS game

- Aim: Enable the computer opponent to keep track of player's current state of technology
  - By collection of knowledge of the player from the world state (resources, units, facilities)
  - "Cheating" and having perfect knowledge will not give fair and realistic AI behaviors
- How to assess state of technology?
  - Sending scouts to collect information and observe (just like what human players do)
  - Make **inferences** via a rule-based system



Rule-based System Basics

- Two main components
  - Working memory – Stores known facts and assertions made by the rules
  - Rules memory – Contains if-then style rules that operate over the facts stored in working memory
- As rules are triggered or *fired,*
  - they can trigger some kind of action (such as in FSM and DT), or
  - they can modify contents of the working memory by adding new information

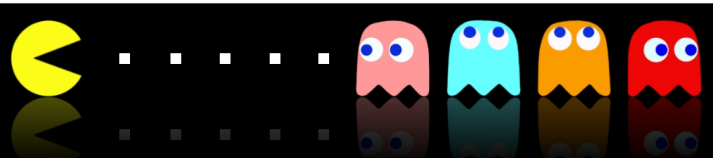


Rule-based System Basics

- Sample working memory

```
enum TMemoryValue{Yes, No, Maybe, Unknown};
TMemoryValue Peasants;
TMemoryValue Woodcutter;
TMemoryValue Stonemason;
TMemoryValue Blacksmith;
TMemoryValue Barracks;
......
```

  - Contains elements that can take any one of the 4 values
  - Idea: Keep track of the current "perception" of the player's state of technology

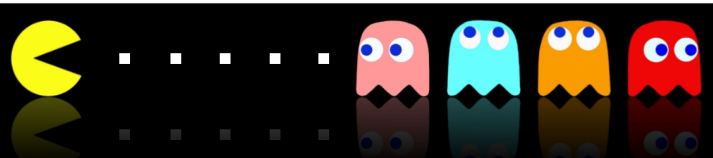


Rule-based System Basics

- Computer can gather facts by sending out scouts to see if a player has built a temple so *Temple* element will be set to *Yes.*
- In another way, we can use a set of if-then style rules to infer the technology that the player has (before a scout confirms it)

- Example “temple” rule:

```
if(Woodcutter == Yes && Stonemason == Yes &&
  Temple == Unknown)
     Temple = Maybe
```

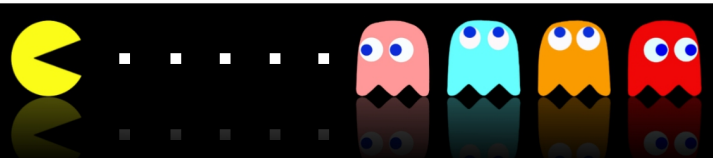


Rule-based System Basics

- Inference can work the other way as well
  - If the player has been observed to have a priest, it can be inferred that the player also must have a temple, therefore, must have a barracks, a woodcutter, and a stonemason

- Example “priest” rule:

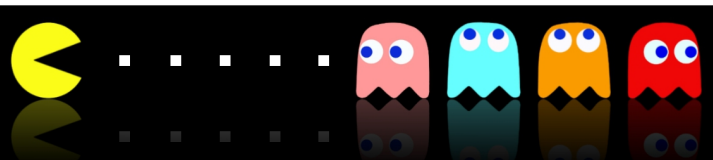
```
if(Priest == Yes) {
   Temple = Yes;
   Barracks = Yes;
   Woodcutter= Yes;
   Stonemason= Yes;
}
```



Rule-based System Basics

- You can have many more rules for this technology tree

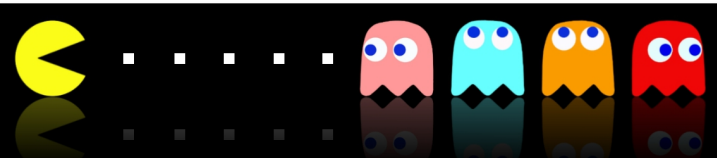
- The main scheme: To write this set of rules and execute them continuously during the game (at each iteration of the game loop or in fixed intervals)
  - Maintain an up-to-date picture of the player's technology capabilities
  - This knowledge can be used to decide when to attack/defend, what to build next and make other tactical/strategic plans



Rule-based System Basics

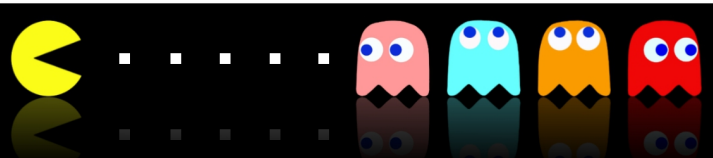
- In reality, developers do not build rule-based systems using actual hard-coded if-statements
  - Some types of inferences are hard to achieve
  - Very inflexible as the rules need to be handcrafted one by one to strike a good balance among them
  - Definitely not efficient for future modifications!

- Developers often use scripting languages or shells to create and modify rules without having to change the source code and recompile



Inference in Rule-based Systems

- Forward Chaining
  - Match rules (if-parts) to facts in working memory
  - If a rule matches, it is fired and its then-part is executed
  - Potentially, if more than one rule matches the given set of facts, *conflict resolution phase* required to figure out which rule to fire
  - Conflict resolution: Many possible ways – first matched rule, random rule, largest weighted rule, etc.



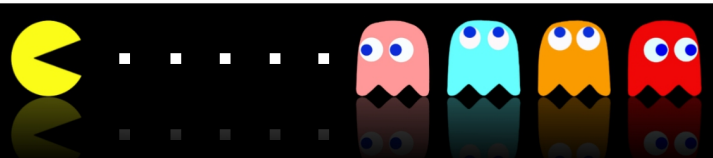
Inference in Rule-based Systems

- Forward Chaining
  - Example

    If working memory indicates Peasants = Yes and Woodcutter = Unknown, the rule:

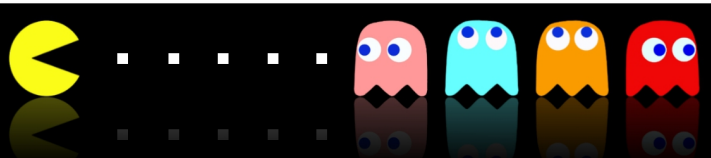
    ```
    if(Peasants == Yes && Woodcutter == Unknown) Woodcutter = Maybe;
    ```

    This rule can potentially be fired depending on which other rules are also matched



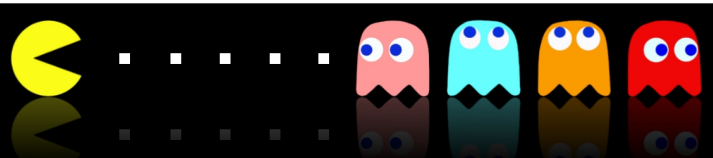
Inference in Rule-based Systems

- Backward Chaining
  - Opposite of forward chaining
  - Match the then-parts, start with outcome/goal, and figure out which rules must be fired to arrive at the outcome/goal
    - E.g. Outcome is that the player has Calvary units. Work backwards – player must have Blacksmith to have Calvary, player must have Barracks to have Blacksmith, player must have Woodcutter to have Barracks and so on



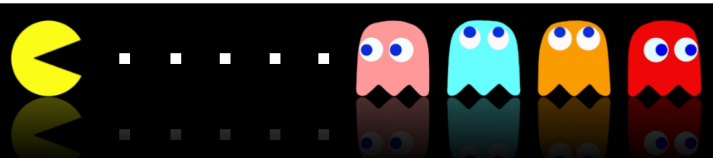
Inference in Rule-based Systems

- Backward Chaining
  - So, all the rules required to reach the outcome are all fired
  - Work the logic backward up the technology tree to the goal
  - In practice, backward chaining is recursive and more difficult to implement than forward chaining



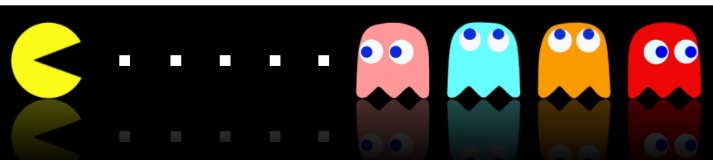
Optimization of RBS

- For small rule sets,
  - Forward chaining is fast
- For large-scale rule-based systems, optimization is essential
  - Many rules may be matched for firing, so conflict resolution phase must be optimized
  - Write your own scripting language (instead of 3rd party ones) to reduce implementation overhead

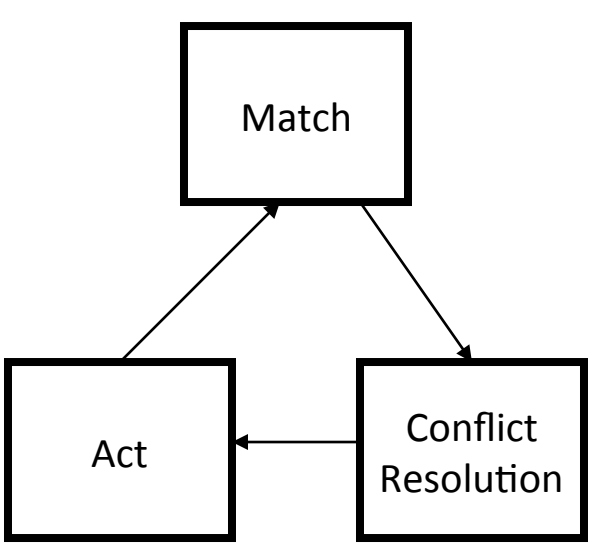


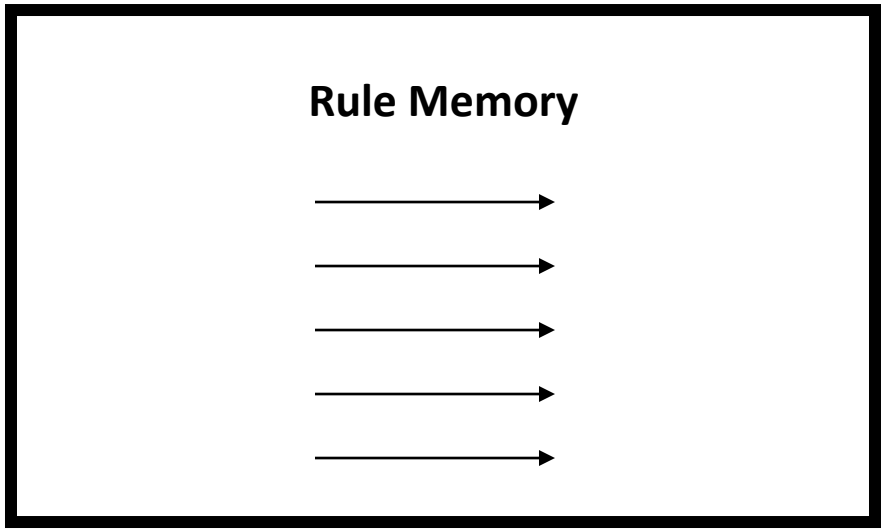
Creating Rule-Based Systems

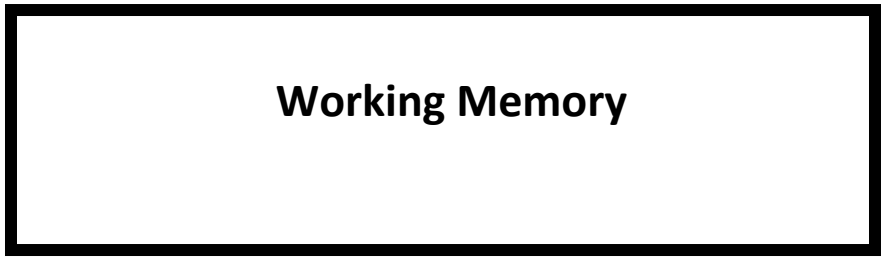
- Decision trees can be converted into rules
  - Just test the disjunction of conjunctions for each leaf
- More general rule-based systems let you write the rules explicitly
- System consists of:
  - A rule set - the rules to evaluate
  - A working memory - stores state
  - A matching scheme - decides which rules are applicable
  - A conflict resolution scheme - if more than one rule is applicable, decides how to proceed
- What types of games make the most extensive use of rules?



Rule-Based Systems Structure







Program

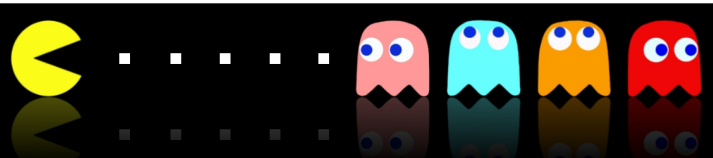
Procedural Knowledge

Long-term Knowledge

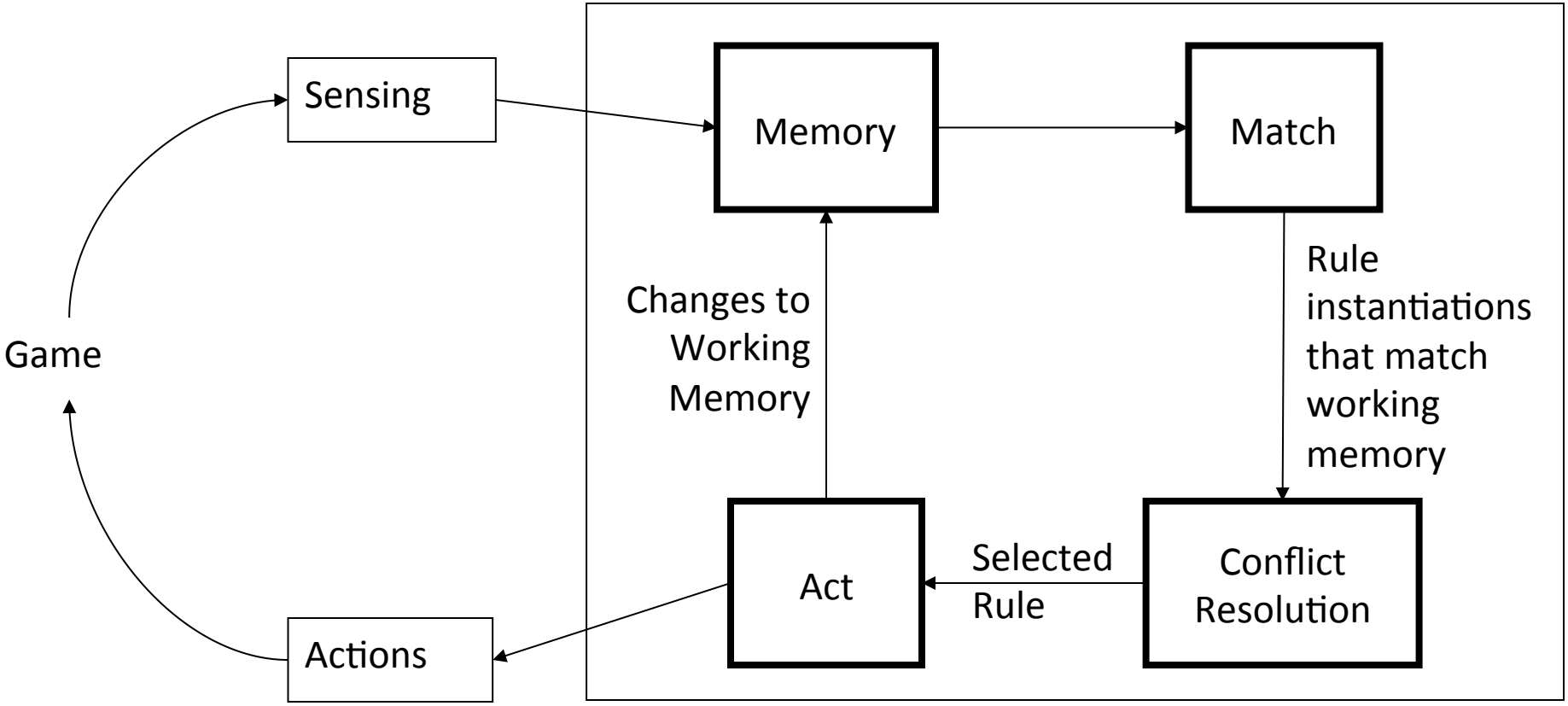
Data

Declarative Knowledge

Short-term Knowledge



AI Cycle


Sensing
Memory
Match
Game
Changes to Working Memory
Rule instantiations that match working memory
Act
Selected Rule
Conflict Resolution
Actions

Civilization: Age of Kings

```
; The AI will attack once at 1100 seconds and then again
; every 1400 sec, provided it has enough defense soldiers.

(defrule
    (game-time > 1100)
=>
    (attack-now)
    (enable-timer 7 1400))

(defrule
    (timer-triggered 7)
    (defend-soldier-count >= 12)
=>
    (attack-now)
    (disable-timer 7)
    (enable-timer 7 1400))
```

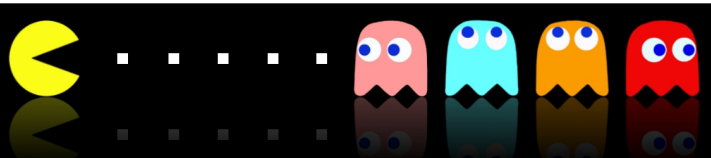
Rule

Action

Age of Kings

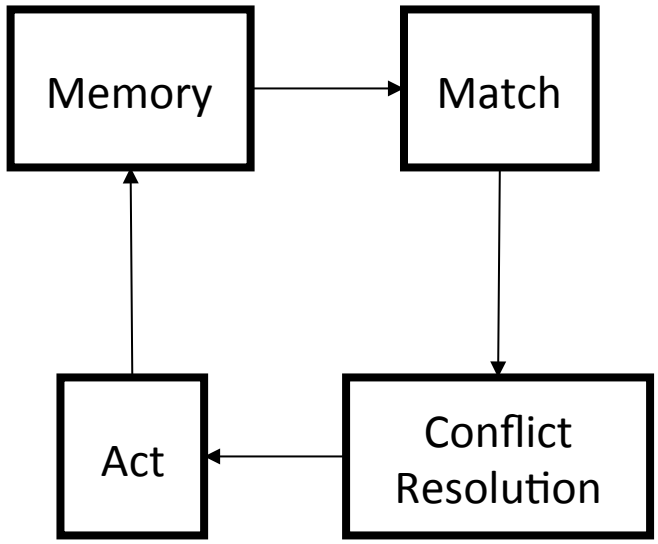
```
(defrule
    (true)
=>
    (enable-timer 4 3600)
    (disable-self))

(defrule
    (timer-triggered 4)
=>
    (cc-add-resource food 700)
    (cc-add-resource wood 700)
    (cc-add-resource gold 700)
    (disable-timer 4)
    (enable-timer 4 2700)
    (disable-self))
```

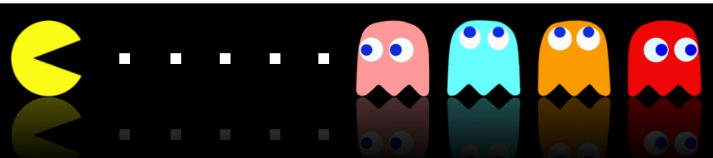
- What is it doing?



Implementing Rule-Based Systems

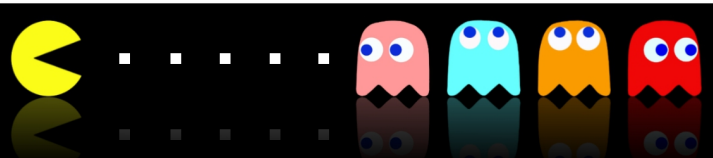
- Where does the time go?
  - 90-95% goes to Match
- Matching all rules against all of working memory each cycle is way too slow
- Key observation
  - Number of changes to working memory each cycle is small
  - If conditions, and hence rules, can be associated with changes, then we can make things fast (event driven)





Efficient Special Case

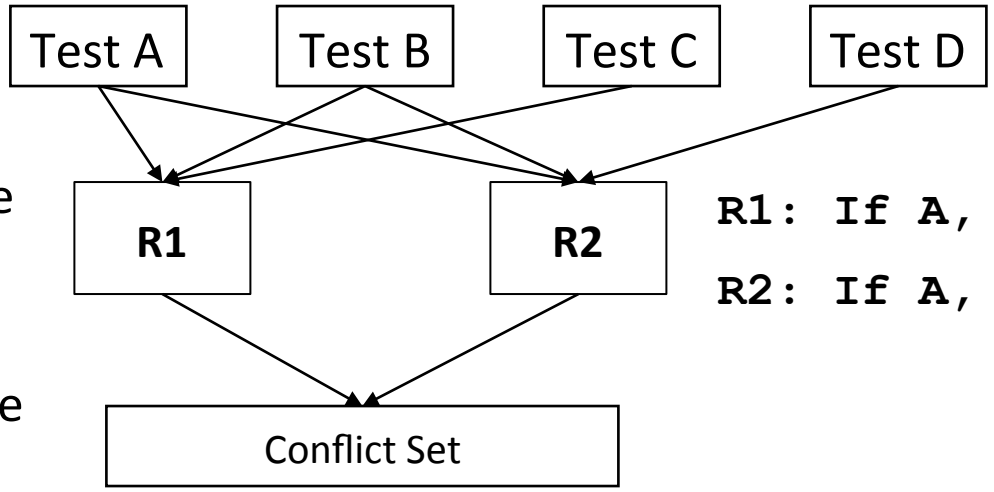
- If only simple tests in conditions, compile rules into a *match net*
- Process changes to working memory
- Associate changes with tests
- Expected cost: Linear in the number of changes to working memory



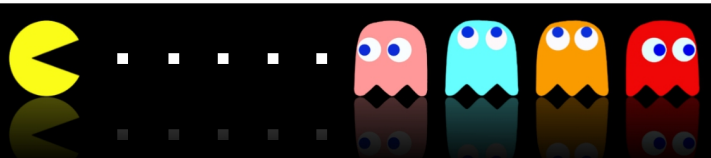
Efficient Special Case

Rules: Bit vectors store which tests are true

Rules with all tests true go in conflict set



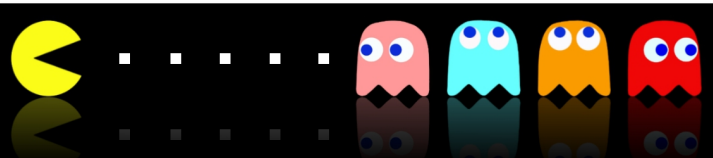

```
R1: If A, B, C, then …
R2: If A, B, D, then …
```



General Case

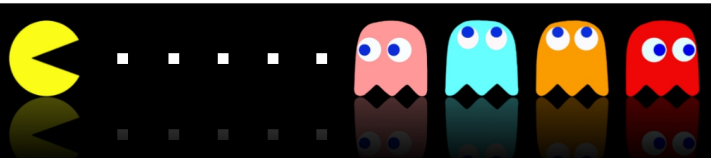
- Rules can be arbitrarily complex
  - In particular: function calls in conditions and actions

- If we have arbitrary function calls in conditions:
  - Can't hash based on changes
  - Run through rules one at a time and test conditions
  - Pick the first one that matches (or do something else)
  - Time to match depends on:
    - Number of rules
    - Complexity of conditions
    - Number of rules that don't match



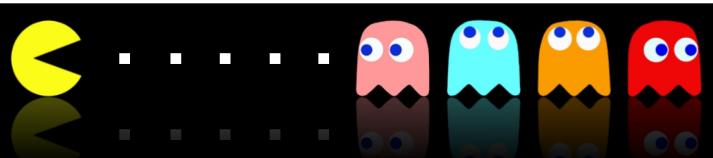
Baulders Gate

```
IF
  Heard([PC],UNDER_ATTACK)
  !InParty(LastAttackerOf(LastHeardBy(Myself)))
  Range(LastAttackerOf(LastHeardBy(Myself)),5)
  !StateCheck(LastAttackerOf(LastHeardBy(Myself)),
              STATE_PANIC)
  !Class(Myself,FIGHTER_MAGE_THIEF)
THEN
  RESPONSE #100
  EquipMostDamagingMelee()
  AttackReevaluate(LastAttackerOf(LastHeardBy(Myself)),30)
END
```



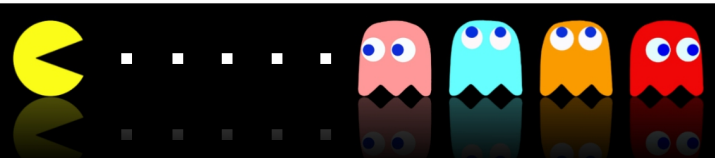
Research Rule-based Systems

- Allow complex conditions with multiple variables
  - Function calls in conditions and actions
  - Can compute many relations using rules



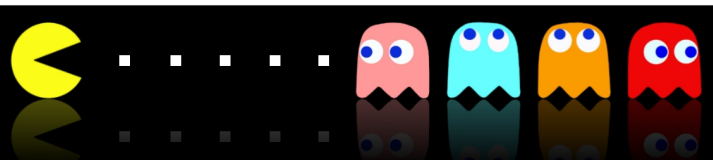
Conflict Resolution Strategies

- What do we do if multiple rules match?



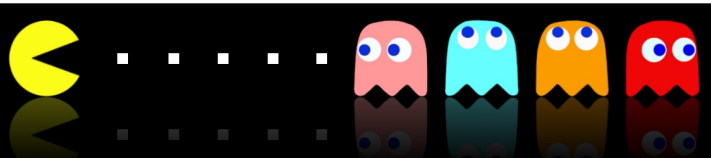
Conflict Resolution Strategies

- What do we do if multiple rules match?
- Rule order – pick the first rule that matches
  - Makes order of loading important – not good for big systems
- Rule specificity - pick the most specific rule
- Rule importance – pick rule with highest priority
  - When a rule is defined, give it a priority number
  - Forces a total order on the rules – is right 80% of the time
  - Decide Rule 4 [80] is better than Rule 7 [70]
  - Decide Rule 6 [85] is better than Rule 5 [75]
  - Now have ordering between all of them – even if wrong



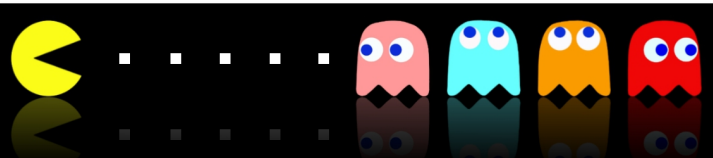
Basic Idea of Efficient Matching

- How do we reduce the cost of matching?
- Save intermediate match information (RETE)
  - Share intermediate match information between rules
  - Recompute intermediate information for changes
  - Requires extra memory for intermediate match information
  - Scales well to large rule sets
- Recompute match for rules affected by change (TREAT)
  - Check changes against rules in conflict set
  - Less memory than Rete
  - Doesn't scale as well to large rule sets
- Make extensive use of hashing (mapping between memory and tests/rules)



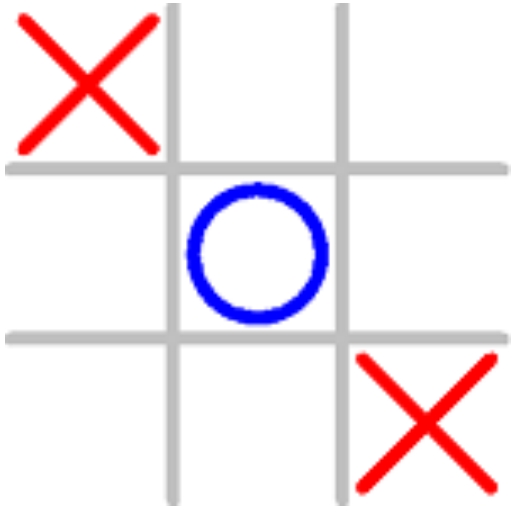
Rule-based System: Good and Bad

- Advantages
  - Corresponds to way people often think of knowledge
  - Very expressive
  - Modular knowledge
    - Easy to write and debug compared to decision trees
    - More concise than FSM
- Disadvantages
  - Can be memory intensive
  - Can be computationally intensive
  - Sometimes difficult to debug

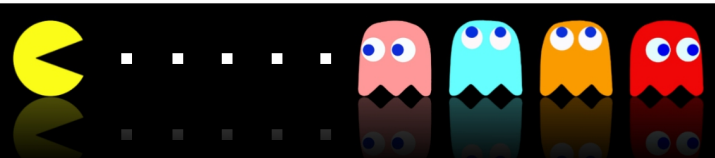
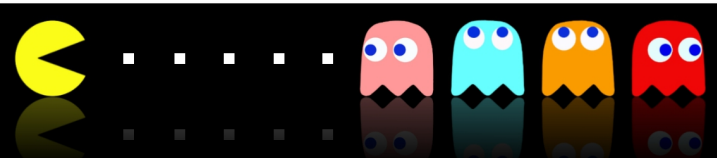


Rule Based Tic-Tac-Toe Exercise

- Write the rules for a computer AI when playing a Tic-Tac-Toe game



Fighting Game Exercise



Questions?