

Agenda

1. **Administrivia**
2. **Rete, rete**
3. **Language: Why language is special; from Words to Meaning**
4. **Syntactic nets to semantic nets**
5. **The Blue Room: The Language of Thought?**

1. **Administrivia: lunch?**

2. **Rete. Key ideas:**

- If-then forward chaining rules can be reorganized for efficient pattern matching.
- The RETE algorithm creates a decision tree that combines the patterns in all the rules of the knowledge base.
- Once it has been determined which patterns have been matched by facts, comparisons of variable bindings across patterns must be checked in a relational database table that 'remembers' what partial matches have already been tested.
- The key to its efficiency is AWP again: to do partial (incremental) matching - it remembers past test results across iterations of the rule matching loop. Only new facts are tested.
- **Variable bindings are saved and reused, rather than recomputed.**

Recall that we talked about **forward chaining** as a procedure that loops through a set of rules, searching a database of assertions for variable bindings that match rule antecedents. When matches are found, rules are triggered and fired, producing new assertions (or removing assertions). When new assertions are added, the loop through a rule set repeats. The Rete algorithm takes advantage of two observations: (1) not many assertions are added or changed when a rule fires, and the changed assertions do not affect many rules; and (2) many rules share antecedents (i.e., their 'if' components are similar). The disadvantage of Rete is: it takes memory to store all these pre-recorded variable bindings/matches, and we must expend effort to remove (failed) match information.

Each node in a Rete network (graph) represents the set of variable bindings that match an assertion or a collection of assertions. The Rete algorithm works by moving assertions through the graph, like a pinball machine, saving incremental match information as it goes. A path through the graph to a leaf represents the bindings that match the antecedents to a rule (take a look below). You can think of the root node as a node that connects subgraphs for all antecedents of a particular length (which is why we can use relational databases to implement them - each node, except the root node, is a table representing one relation, and each row of the table records one assertions. The match procedure uses a database select operation to define the match, then a project operation to find specific matches in the set of all assertions. To collect all antecedents, we use a (sequence of) database join operations, with a final execute action that is a database project to get the subset of all matching variables needed for the consequent.)

To build a rete:

- For each antecedent, create an 'alpha' node, aka a 'match' node (these are the root nodes)
- Join a first antecedent and a second antecedent to create a 'beta' node, aka a 'merge' node
- Join each subsequent antecedent with the previous merge node to create a new merge node
- For each consequence, create a terminal node, aka **execute node**, via **project**, that carries the consequent.

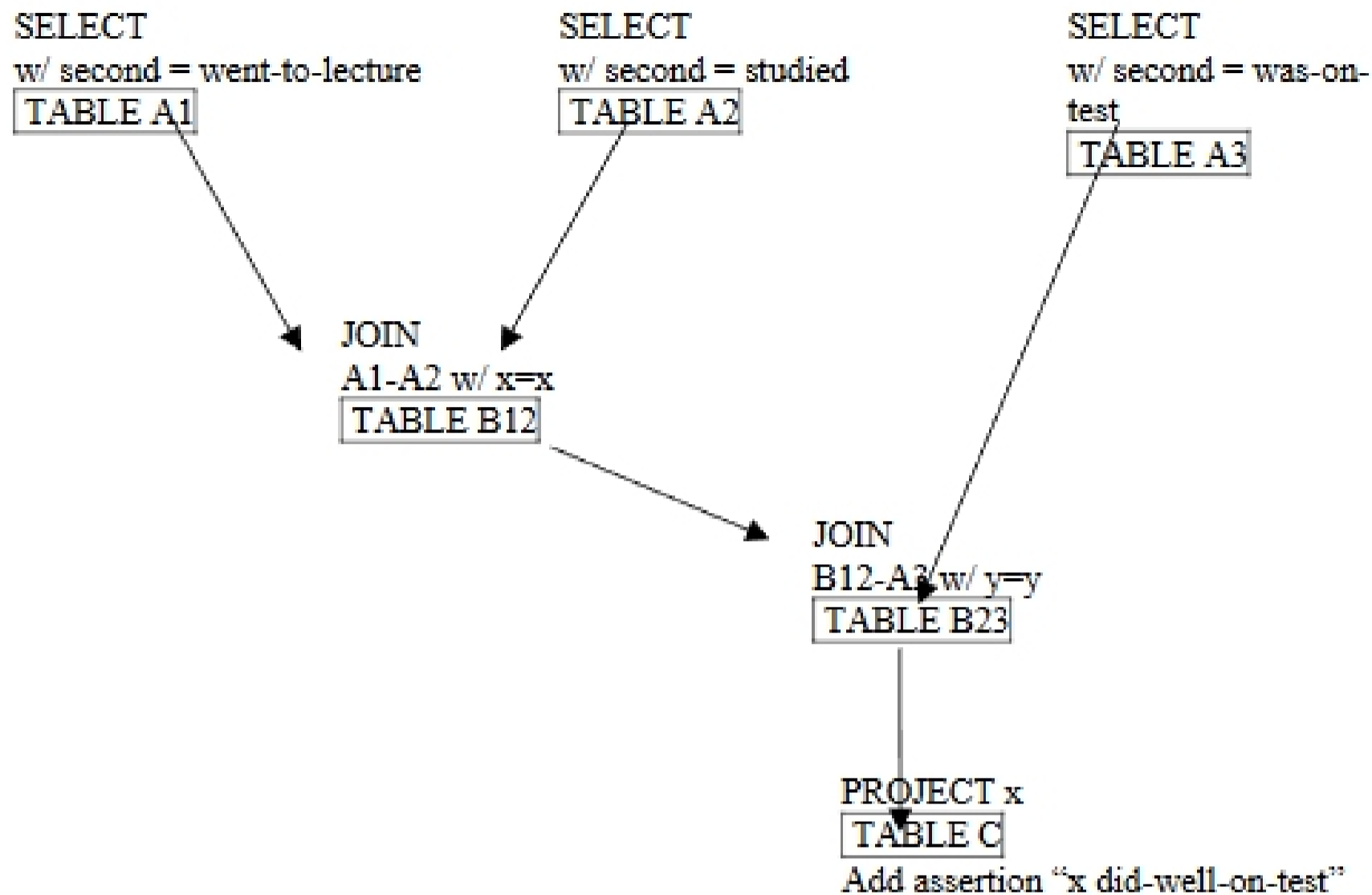
The Rete algorithm keeps up to date the information associated with the nodes in the graph. When a fact is added or removed from working memory, a token representing that fact and operation is entered at the root of the graph and propagated to its leaves modifying as appropriate the information associated with the nodes. When a fact is modified, say, the age of John is changed from 20 to 21, this is expressed as a deletion of the old fact (the age of John is 20) and the addition of a new fact (the age of John is 21).

RULE:

IF (?x went-to-lecture)
 (?x studied ?y)
 (?y was-on-test)
THEN (?x did-well-on-test)

ASSERTIONS:

You went-to-lecture
You studied nothing
Nothing was-on-test



3. Language: Why language is special; from Word strings to Meaning (Syntax to semantics)

Language is special: The Twain Test (Unsupervised learning; very small sample complexity – 1-5 examples; no Wall Street Journal subscriptions), and “the Burst Effect”. At age 1 yr, 1-11 months or so, these are the kinds of utterances children produce:

Ride papa's neck; this my rock-baby; mama forget this

By the age of 3, this is what they produce. What's the difference?

You match me open sandbox; Papa, you like this song? I won't cry if mama wash my hair

The same difference shows up in this way.

Pop-quiz (multiple choice): who produced the following ‘sentences’ (Names changed to protect the innocent):

(1) I see red one (2) P. want drink (3) P. open door (4) P. tickle S. (5) I go beach (6) P. forget this (7) P said no (8) P. want out (9) You play chicken

Multiple choice: (a) Pidgin speakers; (b) apes (signing); (c) Feral child Genie; (d) ordinary children [Hint: recall the Burst effect plus the general rule about tricky 6.034 questions]

Question: How can we *represent* knowledge of language? How can we *compute* with it? (Like data structure + algorithm question). Answer to first part (representation) – this is what linguists do for a living. What are the *underlying representations* that constitute our knowledge of language? Answer to the second part (computation) – this is what syntactic and semantic transition trees partially accomplish, mapping strings of words into procedural operations on a database (“meaning”).

As to the first part, look at how much we implicitly ‘know’ about a language. Take even a simple sentence like *the cats ate ice-cream*.

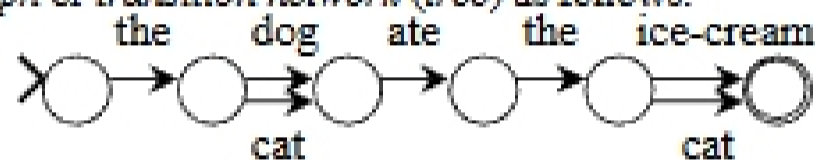
- We know how each word sounds and whether it's an English word at all. *Ca* begins a valid English word, but no English word will start out *ptk* (but could in Polish). Further, the *s* on *cats* marks it as plural.
- We know that the words must appear in a certain order. *Ice-cream ate the cats* means something very different from *The cats ate the ice-cream*.
- We know "who did what to whom," a kind of mental snapshot: *the cats* is the agent of the action *ate* while *the ice-cream* is the thing eaten, or the affected object. (These are the *thematic roles* or *slots in a thematic role frame*). And so on...

It is conventional to divide the study of the representation of linguistic knowledge into two parts: syntax, the study of word arrangements without regard for meaning – this derived from the Greek word *syntaxis*, literally 'to arrange together' and semantics, the study of meaning.

Syntax determines the allowable combinations of primitives in a language, where one notion of primitives are word categories, and further, combinations of word categories. For example, let us suppose that we had sentences like these:

The cat ate the ice-cream
The dog ate the ice-cream
The dog ate the cat

If we take these words for the moment as primitives, we can simply write down these possible linear patterns as a *directed graph* or *transition network (tree)* as follows:



Here, the 'start' state is marked by an incoming arrow, and the final (goal) state is a double circle. Given a transition network and a string of words (a sentence), we can *check* to see if the string of words can be 'let through' the network, or *generated* by it. To do this, we start by looking at the first word in the sentence can let us move from one state to the next via a *transition* – does the label on the arc match the current word in the sentence we are looking at? (See how we can march through the first example sentence this way. If we arrive at the end of the sentence and are in the goal state, we win.)

Of course, it would get very boring (and wrong) to just keep writing down words as arc labels to encompass more and more sentence patterns: *a dog ate the ice-cream*; *a cat ate the ice-cream*; *a dog ate the cat*; *the dog ate an ice-cream*; etc. Instead, we can *collapse* collections of words such as {dog, cat, ice-cream, tree,...} into a *word category* called 'Nouns' and words like {a, the, an...} into a category called 'Determiners' (because they 'determine' in some sense the multiplicity of the Noun that comes later). That gives us the following, more compact network:



Of course, now this transition tree (network) will allow some 'funny' word sequences – but we did say this was *syntax*, not semantics. Only form matters.

Finally, since we are good programmers, we notice a final redundancy in the network: the pattern "determiner noun" is repeated, once as the 'Subject' of the sentence, and once as the 'Object'. So, why not turn that pattern into a subroutine? We can call the repeated pattern a 'Noun Phrase' and splice out the repeated pattern (see the network below.) Of course, since this is a subroutine, we must have a way to *call* and *return* from subroutines generally (i.e., we must use a *stack* and sentences are now *recursive*). Now we will have two networks: the main Sentence network and a 'subnetwork' called a Noun Phrase. We need a new label in the Sentence network that is called 'Noun Phrase', and marking that at the beginning of a valid sentence, we must find a Noun Phrase:

