Rev. 2. **Released:** Wednesday, Mar 11 2009. **Due:** Wednesday, Mar 18 2009, 7:00pm.

0.1 Goals

Our goal with this lab is to make you familiar and confident with using tools from different sub-fields in artificial intelligence so that you can think concretely about how they should work together in an autonomous problem solving agent.

In this first laboratory, you will learn about ways to represent, learn and use knowledge to solve problems. We will present the problems within complete problem solving frameworks, beginning with the very simplest.

0.2 Setting up the Lab

This project will use the Funk 2 programming language, developed by teaching assistant Bo Morgan. This section will walk you through running and using the language. Funk has the following properties:

Causal tracing of the events while executing multiple processes. For example, comparing two different uses of the same functionality in the system.

See the results of executing a process without affecting the system. Funk can *simulate* the execution of a process without making changes to the system, which is necessary (but not sufficient) for the imagination involved in the deliberate planning of actions.

Remember historical values for arbitrary memory objects in the system. Remember what the state of the system was at a previous time, this is not possible in other languages either.

0.2.1 Running Funk 2 on Athena

Registered students with an MIT Athena account can ssh to `linux.mit.edu` and run Funk2. After you have logged in:

```
attach 6.868
/mit/6.868/funk2/bin/funk2
```

The second command will execute funk (it takes a while to start up).

0.2.2 Installing and configuring Funk 2 on your computer.

If you want to run Funk2 from your own computer, you must be running a Linux or OS X based operating system and 2Gb of free disk space.

1. To obtain Funk 2, download the latest version and extract the tarball:

```
wget http://neuromin.de/rct/downloads/funk2-latest.tar.gz
tar xvfz funk2*.tar.gz
```

2. Next, you must configure, compile and generate the funk bootstrap image. *Note, this will take a long time.*

```
cd funk2                    # hit tab to complete this directory name
make clean
make configure
make
```

3. Start up Funk and enter the **read-evaluate-print loop** (REPL):

```
bin/funk2
```

0.2.3 Using Funk2

Knowing scheme or lisp will help a lot in learning Funk2 because the syntax is very similar. We have tried to make a Funk2 equivalent for every scheme operation. Here is a simple "Lab 0" for learning Funk2:

1. The "print" operation allows you to print a piece of memory data to the screen in a user-friendly format:

```
[print 'Hello world!']
```

2. You can use the "quote" operation in order to specific a part of the code that should be treated as Data and not executed (yet).

```
[quote [print 'Hello world!']]
```

3. You can use the "list" operation to construct a new list of data. This list command gives the same result as the "quote" command immediately above.

```
[list [quote print] [quote 'Hello world!']]
```

4. You can execute a list of data by using the "eval" operation.

```
[eval [quote [print 'Hello world!']]]
```

5. You can get a single element of a list of data by using the "elt" function with an index. Indexing for lists starts with 0, like C (and not 1, like some other languages). This returns the first element of the quoted list:

```
[elt [quote [print 'Hello world!']]  0]
```

6. This "elt" command returns the second element of the list:

```
[elt [quote [print 'Hello world!']]  1]
```

7. You can define a global variable by using the "globalize" operation. This example creates a global variable named "my-variable":

```
[globalize my-variable [quote [print 'Hello world!']]]
```

8. You can refer to your global variable "my-variable" by simply entering the symbol directly:

```
my-variable
```

9. The "elt" operation can be used to access the first and second elements of any data, including user-defined variables, as in the following two examples:

```
[elt my-variable 0]
[elt my-variable 1]
```

10. You can define functions using the "defunk" operation. The following command defines a function named "my-funktion" that takes one argument, "x", and this new function simply prints the value of the local variable, "x".

```
[defunk my-funktion [x] [print x]]
```

11. You can call a user-defined function in the same way as any other function, such as the following example:

```
[my-funktion 3]
```

12. You can define functions that take multiple arguments, "x" and "y" in this case, and does simple arithmetic, "+" in this case:

```
[defunk my-add [x y] [+ x y]]
```

13. Here is how to execute the "my-add" function that we just defined above:

```
[my-add 3 4]
```

14. You can set the value of variables by using the "set" operation, as in the following example:

```
[set my-variable [list 1 2 3]]
```

15. Notice how this has changed the value of the variable, "my-variable":

```
my-variable
```

16. The "mapc" operation will execute a function for every element in a list. This example shows how to map the function, "my-funktion", over every element of the list stored in the variable, "my-variable":

```
[mapc &my-funktion my-variable]
```

17. You can generate a completely new list from an existing list by using the "mapcar" operation with a function and a list as arguments. For example, the return values of the function, "my-funktion", for every element of the variable, "my-variable", are stored in a new list in this case:

```
[mapcar &my-funktion my-variable]
```

18. Here is an example of assigning a new list mapping to a global variable, "my-mapped-variable":

```
[globalize my-mapped-variable [mapcar &my-funktion my-variable]]
```

19. And again, referring to our new variable value:

```
my-mapped-variable
```

1 Building a Reactive Agent in Blocks World

"You can't think about thinking without thinking about thinking about something."
– Seymour Papert

Following Papert's advice, we will think about thinking about problems in the famous *microworld*, Blocks World. In this world, you are a disembodied claw and can interact with the items in the environment: a table and colored blocks. Planning problems in blocks world are typically represented as a **goal state**, whereby the agent has to move the blocks one at a time to change the world from its **current state** to reach the goal state. Planning in Blocks World is tractable, but optimal planning (constructing the shortest sequence of actions) is NP-hard [2].

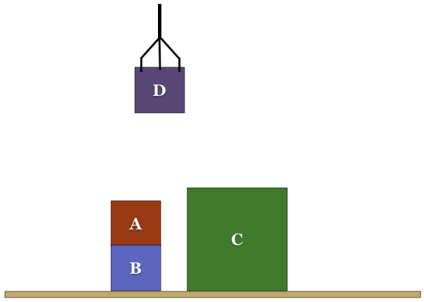

Figure 1: Blocks World is a highly constrained microworld that still is rich enough to allow us to think about planning problems.

After you have installed Funk2, you can initiate the blocks world REPL loop by typing:

```
[blocks_world-test]
```

There are three commands in the relational (FOL) blocks world `[pick-up X]`, `[drop]` and `[move-to X]`. (To exit the blocks World REPL, type `[exit]`).

§ Question 1

Experiment with blocks world. Find a command that fails to perform your action, and find one that succeeds. Describe the conditions under which the command fails and those under which it succeeds.

1.1 Reactive Agents

A **reactive agent** observes the world but only needs to represent the world's state enough to determine which action it should do next. The most basic type of reaction is *instinctive*, an inborn rule that tells the agent how to act in a situation. You can think if *act* generally, not just limited to external observable behaviors, but also including cognitive changes—turn on a planner, increase adrenaline production, etc.

See §5.1 and §5.2 of *The Emotion Machine* for an explanation of 'instinctive' versus 'learned' reactive agents.

These rules can be represented in terms of If-Do control rules. When the antecedent (the expression contained after the If) matches its precepts, then a specific action is performed. In order to solve more difficult problems, we need our agent to be able to predict its environment. It must still detect action preconditions and select actions but now *also* predict the outcomes of these actions using If-Do-Then rules.

§ Question 2

Come up with 5 If-Do[-Then] instinctive rules that you think would be needed in an intelligent agent.

A general way to think of an agent is to divide its processing into a loop of three functions: Sense, Plan and Act. In terms of **inputs** and **outputs**:

- Sense is given **perceptual data** from sensors and transforms these into a meaningful representation of the **world state**, typically comprised of objects, their properties, and relations between objects.
- Plan receives the representation from Sense and uses this along with its **background knowledge** to produce a **plan**, usually a description of intended actions and predictions of future world states, in order to achieve its **goals**.
- Act takes the plan description and *executes* the actions at the right times, while Sense monitors the changes produced in the environment, generating more **perceptual data** and repeating the agent loop.

At first in this lab, our plans will barely be representations at all: we will focus on learning simple If-Do rules, in particular, learning the conditions for the rule to engage, in a process called *concept learning*. In section 1.3, we will learn predictive If-Do-Then rules that describe actions and their expected outcomes.

Even simple reactive agents can still be goal-based. There are two reasons for it to take an action: **exploration**: to learn more about its environment; and, **exploitation**, to maximize its (expected) rewards. In reinforcement learning, these are considered a trade-off: either the agent is building its model (exploration) or using the model to predict the reward of its future actions (exploitation). Rewards are external indicators of goal attainment.

1.2 Learning an Reactive Agent with If-Do rules

In this section of the lab, we are going to learn the If-Do rules of a reactive agent. Much of the discipline of machine learning is focused on learning problems that are similar to our problem of learning an action-selection function; it falls into the general problem of learning a *classifier* that maps between an input vector (sensations) onto a set of category labels (actions).[1]

If we limit ourselves to the task of learning a rule to classify "relevant" versus "not relevant" preconditions for an action to take place, then we do not really need a description of the rule's antecedent. In fact, we could throw away the representation and focus only on the data's behavior, by learning a *discriminative classifier*. Instead, we will learn the full function, not just the description, so that we can reason about this knowledge later.

We are not learning a rule to select the best action from a set of actions, we are learning a rule for *each* action that determines whether or not it is relevant. This is a problem of **concept learning**, learning a binary valued function from examples of *input* and *output* pairs. Later in section 3 lab, you will learn more details about concept learning.

This lab is interdisciplinary and sometimes non-linear. You can skip to section X to learn more about concept learning.

§ Question 3.1

Using the limited propositional representation, modify the code in `som-2009/lab1-prop.fu2` to given an agent the enabling conditions for the `[pick-up the-red-block]` action. Blocks World will tell you when the action fails, and you imagine the agent gets punished when it does the action at the wrong time and fails (*i.e.*, a huge cost on performing this action that is only outweighed by the action's success).

§ Question 3.2

Not all learning experiences are equal. There are two 5 step learning experiences (precondition state, result) that the agent may take when learning the If-Do rule for `[pick-up the-red-block]`. They both result in very different learned problems. Given the two learning sequences of actions in `learning-exploration-1` and `learning-exploration-2`, what are the different kinds of antecedent rules learned by the program?

§ Question 3.3

Which one of those rules is better (define what *better* means)? Explain why these learning experiences, both an equal number of events, have lead to very different learned rules.

[1] In machine learning terminology, *discriminative classifiers* learn the decision rule, the conditional probability, without representing the rule's antecedent. *Generative classifiers* learn the entire joint probability distribution, which includes a description of the underlying rule.

1.3 Learning an Reactive Agent with If-Do-Then rules

Now we want to extend our action descriptions to also include a description of their effects. (This question has been removed from the lab.)

2 Learning Knowledge Representations for Problem Solving

> "Cases where the situation is different... require particular consideration if they are not to plunge all our hard-won distinctions back into confusion." – Edmund Husserl

A **knowledge representation** (KR) is a data encoding denoting something that is believed by an agent. Although KRs are used to varying extents in all areas of AI, unfortunately, the area does not have clearly defined goals, and consequently there are many different ways to say the same ideas and some irrelevant distinctions where there should not be. Knowledge representations are tightly related to **reasoning** or **inference** techniques (often these are representations themselves), that can be used to derive new knowledge from existing knowledge. Knowledge representations are also related to **machine learning**, which deals with learning representations or assigning values to variables. This description of learning overlaps with that of inference[2] and in the problem solvers we would like to build, learning processes will involve background knowledge and themselves be representations that are subject to reasoning and learning.

To protect your mind from the noise, when learning a new knowledge representation you should ask:

- *What is it?* Translate your answer to basic, well-understood mathematical concepts *i.e.*, a definition involving the precise meanings of set, relationship, sequence, orderings, properties of relationships, and so forth.[3]
- *How is it used by the agent?* What are the problems that can be solved by this knowledge?
  - *How can it be programmed?*
  - *How can it retrieved or indexed when it is needed?*
  - *How can it be learned?* How can these representations be "generated"? How are two representations of this kind compared?

Unfamiliar with these math concepts? Take a look at this webpage by John Sowa and some of the materials on course website.

Next we focus on two systems of logic: propositional logic, and a popular extension of it, first-order logic.

2.1 Propositional Logic

Logic is a knowledge representation that predates computer science. Logic evolved out of attempts to formalize a system for making valid inferences, and the first one dates back to Aristotle

[2] In logics, for example, the distinctions between these types of learning procedures have their own names: *deduction*, using inference rules to produce new assertions, and *induction*, creating new rules that cover and extend the observed examples

[3] It is helpful to use the language of mathematics to describe declarative knowledge representations. In AI, exotic representations are commonly discussed, but rarely are representations ever compared. For example, *n-grams* can be considered to be *mixtures of 1...n-order Markov models*, each of which can thought of as *finite state machines*, propositional states with a probability distribution on transitions between them. With a common language, representations can be compared on their own merits and the religious wars between **symbolic** and **not-symbolic** approaches evaporates: both are represented by symbols, so the distinction is meaningless. Instead of "system X is sub-symbolic", one should say "system X annotates its learned data structures with values from the set of real numbers, $\mathbb{R}$, and uses a linear order on $\mathbb{R}$ to compare them."

(c. 300BC). While propositional logic is limited in its expressibility,[4] it is very widely used in machine learning today.[5]

Propositional logic represents the world in terms of simple assertions. These are technically called **sentences**, which is similar but not the same as natural language sentences. Here are two examples:

The **power set** of a given set S, denoted $\mathscr{P}(S)$, is all possible subsets of S. If $S = \{a, b, c\}$, then $\mathscr{P}(S) = \{\}, \{a\}.\{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}$ and $\{a, b, c\}$.

- p = The red block is on the blue block.
- q = The green block is on the blue block.

Each of these assertions is represented by a propositional symbol (*e.g.*, p, q, r, s) that makes a claim about something that can be *true* or *false* in (a model of) the world. A system of logic has:

1. **Syntax**: Specifications to construct and recognize valid formulas in the language (usually specified by a grammar: symbols and transformational re-write rules).
2. **Semantics**: The relationship between the system and its **models** or **interpretations**. Logical semantics are tied to model theory, giving it a precise definition of semantics: mapping well-formed assertions to models. The definition of "semantics" outside of logic is much more general and connotes anything related to meaning.

A semantic interpretation involves assigning a truth value, {*true*, *false*}, or equivalently, {*1*, *0*},[6] to every proposition. For the example above, one interpretation is p = 0, and q = 1.

In general, if there are n propositions, then there are 2^n possible interpretations—unique ways to assign *true* or *false* to the propositions. In other words, each member of the power-set of the proposition symbols represents all of the *true* propositions in one possible configuration of our world state. We say a hypothesis/state s_1 is *more general* than s_2, denoted $s_1 \preceq_G s_2$, if s_1 is a *subset* of s_2. For example, the set {a,b} is more general than the set {a,b,c}, and the empty-set {} is the most-general category of them all. We can add domain theories to restrict our interpretation or introduce new propositions into the world.

$|S|$ denotes the size of a given set S. The size of a power set $|\mathscr{P}(S)| = 2^{|S|}$. You could think of 2^n as the generative process of asking n sequential 'yes' or 'no' questions about whether or not each component is in the set, *i.e.*, $|\{yes, no\}|^n$.

p	q	¬(p)	∧(p, q)	∨(p, q)	→(p, q)	↔(p, q)
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

Table 1: Sentential connectives, in infix notation, along with the **truth table** for two propositions, p and q.

The operators in the table above are used to define relations between propositions, to build larger units like p ∧ q and p → q. These larger units themselves resolve to truth values, which percolate up from their constituents. One interpretation is commonly confused: the → implication operator defines a relationship that is only *false* when the antecedent is true and the consequence is false. So p → q is completely the same (has the same constraints on the interpretation models) as ¬p ∨ q. Therefore, if p = "1 + 1 = 3", then p → q always evaluates to *true* because no models should claim 1 + 1 = 3.

[4] Some functions cannot be represented by propositions, like the *parity function*: $f(x) =$ `mod(length(x),2)`.

[5] Most representations in machine learning use probabilistic extensions of propositional representations; instead of mapping to binary values $f(\mathsf{p}) \rightarrow \{0, 1\}$ representing belief in *true* or *false*, they map to a real number, $f(\mathsf{p}) \rightarrow [0, 1]$ representing *degree* of belief.

[6] Assigning binary numbers to propositions can make automated inference easier. ¬(p) = 1-p, ∧(p, q) = `min`(p, q) and ∨(p, q) = `max`(p, q).

We can use these *sentential connectives* to form a Boolean hypothesis space, $\mathcal{H}_{bool}$, to describe all of the possible combinations of world states—with a size of 2^{2^n}. As discussed later in section 4.2, we can shift our description level and start treating our objects as *features* and our sentential connectives as a simple kind of relation between these features, and then form *larger* "objects" out of them! This, again, is the problem of *concept learning*. If we limit ourselves to conjunctions (the and: $\wedge$), our hypothesis space $\mathcal{H}_{\wedge}$ allows us to make only 2^n different concepts. Unfortunately, this hypothesis space could not express concept descriptions like "blue or not green". However, by using *all* of the relational connectives we can combine our n proposition symbols and our hypothesis space, $\mathcal{H}_{bool}$ can now form 2^{2^n} different concepts: the power-set of the power-set of n.

2.1.1 Inference in logic

Inference rules in logic take a very similar form as the knowledge: they are represented with variable symbols and sentential connectives. However, they are part of a **meta-logic** system, and are used to detect patterns between sentences and to then perform *'truth-preserving transformations'* in order to *deduce* new logical assertions.

If you do not know the difference between **deduction** and **induction**, see this.

§ **Question 4.1**

An argument in logic is a set of premises from which a conclusion logically follows. Propositional Logic is far from being as expressive as a natural language, like English. When translating an English statement to propositional sentences, a lot of information is lost. For example, all of the subtle distinctions between connectors "while", "because", "is", "since", "although" are replaced by the same conjunctive $\wedge$ operator. Translate these English statements into logical expression. Suggested propositional symbols are given at the end (in parenthesis).

1. If each man had a definite set of rules of conduct by which he regulated his life he would be no better than a machine. But there are no such rules, so man cannot be machines.—A.M. Turing, "Computing Machinery and Intelligence", Mind Vol. 59 1950. (rules,manmachine,b)
2. Either logic is difficult, or not many students like it. If mathematics is easy, then logic is not difficult. Therefore, if many students like logic, mathematics is not easy. (d,l,m)
3. Everyone who smokes marijuana goes on to try heroin. Everyone who tries heroin becomes hopelessly addicted to [drugs]. therefore everyone who smokes marijuana becomes hopelessly addicted to [drugs]. (m,h,a)
4. The after-image is not a physical space. The brain process is. So the after-image is not a brain process.—J.J. Smart "Sensations and Brain Processes" 1959. (a,p,b)
5. If Kurzweil is right, then the singularity is near. If the singularity is near, then we stop doing research. We are either doing research or working in finance. We do not work in finance, therefore Kurzweil is wrong. (k,s,r,f)

§ **Question 4.2**

Go through the previous list and test whether the argument is *valid*, by comparing its structure to propositional logic's inference rules. An argument if valid if and only if the premises are true then the conclusion is always true. You are evaluating the structure, not the semantic content, of the variables.

2.2 Relational Logic

Propositional logic is clearly limited for the kind of expressions we want in the blocks world. We want to be able to easily transfer knowledge about what we have learned from dealing with the *red block* to blocks with other colors.

There is a strong association between logical representations and databases. In a relational database, a relation is a tuple.

Some people [3] find it helpful to think of propositional logic as the analog of a single table in a database (attribute-value representations can be mapped to propositions) and first-order logic as the same representation as relational databases. Relational logics allow us to talk about objects, their properties, and relations.

There are many kinds of relational logics. One of them is **First-order logic** (FOL), also called **predicate calculus**, and it has become something of a standard. FOL has its origin in work by Frege (1879) and was designed to formalize all of mathematics. Since the work of John McCarthy [4], it has been used in AI. Restricted forms of FOL, like **clausal logic** allow deductions to be easily made with automated theorem provers[7].

FOL is worth learning about because it is a widely used, expressive and well-understood way to represent, reason and learn relational knowledge. A machine learning community addresses the problem of learning FOL predicates and goes by the name *Inductive Logic Programming* (ILP). Recently, some members of the (previously disjoint) areas ILP and statistical machine learning have formed a disorganized field called *Statistical Relational Learning*. There are other logical formalisms, such as the **event calculus**, which extends FOL with ways to avoid many problems related to reasoning about events.

2.2.1 First-order Logic

First-order logic has three kinds of symbols: predicates, functors, and variables. A set of **predicate symbols**, $\mathtt{P} = \{\mathtt{p}_i/\alpha_i, ...,\mathtt{p}_n/\alpha_n\}$, along with their *arities* (the number of *terms* they take) describes *relationships* between constants and can be assigned *true* or *false* values. For example, the predicate `on/2` and `heavy/1` can be used to assert that `on(blue-block,table)` and `heavy(blue-block)`. When a predicate is true we say the relationship "holds". There are special names for predicates with arities 0, 1 and 2:

- 0-term predicates are the same as **propositions**. (Yes, FOL is an extension of propositional logic.)
- 1-term predicates are called **properties** or *unary* relations, like `heavy(blue-block)`.
- 2-term predicates are *binary* relations, like `on(blue-block,table)`.

The size of the set of possible world states `S` is based on `P` and `C`:

$$|\mathtt{S}| \quad = \quad \prod_{i=1}^{n} \left(2^{|\mathtt{C}|}\right)^{\alpha_i} \tag{1}$$

A predicate's *terms* can either be functors or variables. **Functor symbols**, $\mathtt{F} = \{f_i/\alpha_i, ..., f_n/\alpha_n\}$, do not define relations as predicates do, instead they are functions that map constants to constants. **Constant symbols**, the set of which is denoted `C`, are the common case of functors with 0 arity, and these denote *objects* or *items* in the domain of interest. For example, in our blocks world we have $\mathtt{C} = \{$red-block, blue-block, yellow-block, green-block, gripper, table$\}$. A functor f with arity k is a mapping from $\mathtt{C}^k$ to `C`. For example, `mother(dustin)` maps to the constant symbol sharon. We can get rid of most functors $\mathtt{f}/\alpha$ by converting them into a relation $\mathtt{r}/\alpha+1$, for example `motherOf(dustin,sharon)`. An exception is functors that are defined recursively: for example, with the constant `0` and the successor functor `succ/1`, we can define all of the natural numbers: `0`, `succ(0)`, `succ(succ(0))`...

[7] Prolog is a standard logic programming language that uses clausal logic, the technique of *unification* to fill variables in with constants and *resolution* to make inferences.

Variables, usually denoted with capital letters, specify an undefined constant that can be queried: `on(blue-block,X)`, and an inference engine should return false or one or more substitutions like `X/table`. A term with no variables is called a *ground atom*.

Terms can be joined together using the same Boolean connectives that are used in propositional logic, but in addition, FOL provides the ability to quantify variables. These two operators, called the **existential quantifiers**, are $\forall$ "for all" and $\exists$ "there exists". With these, we can write formulas that make claims about *some* members or *all* members[8] of a set:

$$\forall \mathtt{X}\ \mathtt{block(X)}\ \wedge \mathtt{on(X,gripper)} \rightarrow \mathtt{inGripper(X)} \tag{2}$$

Equation 2 makes a claim about all blocks, telling us that if the block is in the gripper then we should give it the `inGripper` property.

$$\forall \mathtt{X} \forall \mathtt{Y}\ \mathtt{inGripper(X)} \wedge \mathtt{inGripper(Y)} \rightarrow \mathtt{X} = \mathtt{Y} \tag{3}$$

Equation 3 uses the results of equation 2 and is a convoluted (logical) way of saying "only one block can be in the gripper at once", ruling out all interpretations of models where there there are two distinct constant symbols that have the `inGripper/1` property simultaneously.

A FOL representation of one state of blocks world looks like:

```
on(red-block,table)
on(blue-block,table)
on(yellow-block,blue-block)
```

§ **Question 5**

Using the notation of FOL, write down the a) **antecedents** (preconditions) and the b) **effects** for the actions:

1. `pick-up(X)`

2. `drop(X)`

3 Concept Learning

Concepts are representations that map the world into two classes—those in the concept and those not in the concept; $f \rightarrow \{true, false\}$. Concept learning is the problem of learning a description of a concept, usually with labeled examples. Here are many alternative definitions of a concept:

- Membership function that maps to *true* or *false*.
- Cognitive representation that divides the world into *things in the concept* versus *things outside the concept*.
- Intentional definition of a set, *e.g.*, $\{\mathtt{x} : \mathtt{blue(x)} \wedge \mathtt{has(x,feathers)}\}$.

However the term has a shared history in psychology as well as AI. This section will cover human concept learning.

[8] Or we can define our own category using combinations of properties of variables, as we do by enforcing the `block/1` property in Equation 2, so the rule to only applies to blocks.

3.1 Concept learning in cognitive sciences

In the 1950s, Bruner et al [1] studied human subjects' ability to learn concepts given positive and negative examples of the concept. Instances were cards containing combinations of four features: Borders $= \{1, 2, 3\}$, Shape $= \{cross, circle, square\}$, Number $= \{1, 2, 3\}$ and Color $= \{black, red, blue\}$. A training example is a card and being told whether the card was in the concept or not. A subject is given a sequence of examples and time after each to describe the concept.

Bruner et al. learned that conjunctive concepts were generally easier to learn than disjunctive concepts. Although this is generally true, exceptions have been found where disjunctive concepts are easier to learn when background knowledge is salient (Pazzani 1991).

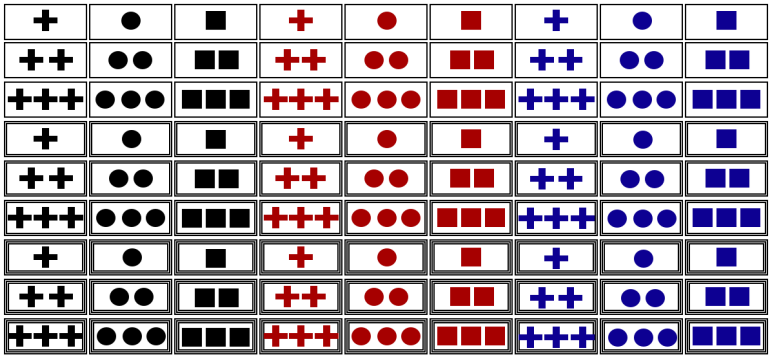

Figure 2: The perceptual world of 81 distinct items that Bruner et al. studied. |Borders| $\times$ |Shapes| $\times$ |Number| $\times$ |Color| $= 3 \times 3 \times 3 \times 3 = 81$.

Do Bruner's results generalize to all concepts? This work is criticized for assuming that every category was represented by a set of necessary and sufficient features (a logical hypothesis space) and for not studying real-world concepts.

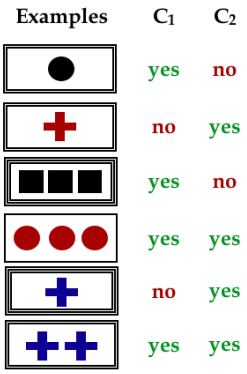

Figure 3: Example training data, and two columns for different concepts.

§ **Question 6** You are a subject of Jerome Bruner who presents you with the example concept instances in Figure 3 and tells you whether they are examples of the concept or not. Learn a concept description for C1 and C2. To correctly simulate the experiment, imagine that you are presented with each example (reading the column top to bottom) sequentially, and are given a moment to re-construct your description of the concept based on the label "yes" it is an

example of the concept or "no". After, try to comment on the *hypothesis space* from which your description was generated.

3.2 Inheritance and the Representation-Inference Trade-Off

In the late 1960s, Ross Quillian developed a taxonomic semantic network, a hierarchical model of human memory. His computer program would efficiently organize knowledge using the idea of *inheritance*: properties can be stored at one location—their most general categories, and then 'inherited' to their more specific category members by an inference process when needed. For example, the property `canFly(.)` would only be located at `birds`, and then be *deduced* to the members of the category when needed: Does a robin fly? `canFly(bird)` $\wedge$ `isA(robin,bird)` $\rightarrow$ `canFly(robin)`.

Qulllian's taxonomic model of semantic memory explained many experimental findings. For example, people take longer to answer questions like "Does a robin have a spine?" than they do with questions that rely on local features like "Does a robin fly?". In addition, people were faster at accessing words when they had been primed with a semantically similar word.

3.3 The typicality effect and prototypes

People are more quick to affirm that `IsA(robbin,bird)` than `IsA(chicken,bird)`, and this is called the **typicality effect**. (Robins are more representative of birds than are chickens.) How can this be if both `chicken` and `robin` both have features that match the definition of the `bird` category? A similar problem was pointed out by Wittgenstein; he used the concept of a `game` as an example where he could not think of a logical combination of sufficient and necessary features that describe the wide class of activities that could be called games (Try it!). Rosch [6] showed how **prototypes** could be used to explain this graded membership.

- the **prototype** view holds that each category has a representative *prototype* that new objects are compared with when they are being classified.
- **exemplars** are like the prototype system, except for that there are multiple representative members of each category.

Rosch, unlike Quillian, did not come up with a representation to support her prototype theory. Maybe the solution is in the index? AI researchers presented **decision trees**, which were a way to learn a sequential classification rule by using criteria to pick the most salient attributes at each step in the classification process. In other words, this does away with the idea of having all knowledge at the global scope.

What happened next? See an overview of the problem of representing concepts in cognitive science by Doug Medin [5].

4 Problems Going Forward

4.1 Reasoning about actions

Events change the state of the world—after an event, predicates can switch from true to false or false to true. Logics have some problems expressing this change, including:

The Frame Problem. The frame problem involves accounting for what has and what has not changed in the world after a given action is executed. In particular, the *unintended* effects of an action are often difficult to model. Many action formalisms, including STRIPS, assume the **commonsense law of inertia**: properties in the world do not change unless they are explicitly mentioned in the effects of the actions.

The Ramification Problem. Similar to the frame problem, the ramification problem is the problem of representing all of the derived consequences of an action. Noted by Minsky in 1961 "One problem that has been a great nuisance to us arises in connection with non-mathematical problems in which actions affect the state of some subject domain... one must then deduce all of the consequences of an action in so far as it affects propositions that one is planning to use." Ramification problems come out of properties that are derived from domain theories, such as the `inGripper/1` property.

Naive solution: run the theorem prover after considering each action.

The Qualification Problem. Actions have preconditions and, like the effects of actions, these are hard to learn and author by hand.

Key point: These problems all point to the issue of determining which knowledge is *relevant* to the task. As the knowledge-base expands and we move outside of toy domains, irrelevant features become more of a problem for learning and reasoning. It is helpful to know, for example, that picking up an object will not affect the items color, weight, or size. Consequently, learned knowledge should be retrieved by causal relevance to the particular events that it can affect. To give the agent a reasonable learning bias, we can assume events are only represented by the agent when they are involved with predicting the outcome of some goal-driven plan.

4.2 Scoping of knowledge representations

> After seeing the classrooms, the student center, the dorms, and the dining hall, the confused tourist asked "I see these buildings, but where is *the* university?" – paraphrase of W.V.O. Quine's category problem.

Humans naturally think about the world in terms of objects and relationships between the objects. Imagine you are walking around Memorial and Massachusetts Ave. The *things* you encounter include buildings, trucks and people, and you can reason about their properties and relationships: *Gerry is standing in front of the student center. An ambulance is parked behind the Greek food truck.* When you enter building 5, the environment changes: the *things* you encounter are now class rooms, offices, and people. You enter an office and now the things you represent are desks, papers, chairs, etc. You look in an administrator's drawer and now you start representing items like pens, white-out, rubber cement and paperclips.

The above scenario illustrates that way you can shift representations. At one point, you are reasoning about items and relationships between them, and the next moment, the you are reasoning about *things inside* those items! When you are outside of the item, it is helpful to hide most of the details and deal with the item as if you were acting through an interface. People familiar with *object-oriented programming* will understand this concept—the inter details of a data structure can be hidden from the programmer who is using the object.

Ignoring these object barriers can lead to absurd mistakes. OpenMind[9] once deduced "a toe is part of society", making an inference on the transitive relations `PartOf(toe,person)` and `PartOf(person,society)`. You could think of an item as some combination of its properties/features, and then go on to talk about relationships between items. But each of those properties/features could also be reasoned about as if it were an item.

Key point: When you look into the semantics of most common programming languages, many of them pay significant attention to scoping issues: reusing/overloading symbols and hiding most of details when possible. Most (all?) logics avoid this important issue. One of the powers of logic stems from the fact that it is a declarative representation: each fact is self contained (modular) and comprehensible. We can simply copy the FOL predicates from one KB and plop them in another—so long as their symbols conflict in a way that causes rules to contradict each other. Human knowledge is much more context sensitive than logic: symbols

[9] OpenMind commons: http://commons.media.mit.edu

have different meanings depending on the problem solving context. Knowledge depends on hidden assumptions about the context. For example, take the assertion "birds fly": what about penguins, ostriches, dead birds, toy birds, caged birds, gestures involving the middle finger, badminton targets, Larry Bird, and Charlie Parker? They don't fly; yet this knowledge is still useful for our default reasoning. Think about how this relates to the problem in section 4.1.

References

[1] Bruner, J., Goodnow, J., and Austin, G. A study of thinking. *books.google.com* (Jan 1986). 11

[2] Chenowet, S. On the np-hardness of blocks world. *Proceedings of AAAI-91.* (1991). 4

[3] de Raedt, L. Logical and relational learning: Ch1. 9

[4] McCarthy, J. Programs with common sense. 9

[5] Medin, D. L. Concepts & categories. 12

[6] Rosch, E. Principles of categorization. *MIT Press* (Jan 1999). 12