COURSE INFORMATION

Text

Matt Ginsberg (1993) Essentials of Artificial Intelligence, Morgan-Kaufmann
(except for Chapters 10,11,12,15,16)

Evaluation

Available grades:

non-completion: Incomplete, Withdraw, etc.

completion: A A- B+ B B-

- A: reserved for superior performance
- A-, B+: expected grade for conscientious performance
- B: adequate performance
- B-: equivalent to failing

Grading Options:

1. Performance Quality: attendance, participation, assigned exercises
2. Grading Contract: specify a set of behaviors and an associated grade.

Course Outline

week	topic	text chapters
1)	Introduction to AI and Overview	1 and 2
2)	Knowledge Representation, Logic	6 and 7
3)	Languages (LISP)	
4)	Reasoning and Deduction	8 and 9
5)	Search	3, 4, and 5
6)	Frames, Objects, and Agents	13
7)	Expert Systems	18
8)	Planning	14
9)	Natural Language	17
10)	Ethics, Review and Discussion	19

FREE LISP ENVIRONMENTS

You will be learning a foreign language called LISP. Here are some pointers to find out more about it:

References

http://www.apl.jhu.edu/~hall/Lisp-Notes/Lisp-References.html
http://www.yahoo.com/Computers_and_Internet/Programming_Languages/Lisp/
http://www.cs.cmu.edu/Web/Groups/AI/html/faqs/lang/lisp/top.html
ftp://rtfm.mit.edu/pub/usenet/news.answers/lisp-faq/
http://www.cambridge.apple.com/other-langs/lisp.html
http://www.elwoodcorp.com/alu/
http://www.ai.mit.edu/projects/iiip/doc/cl-http/home-page.html

http://www.psychologie.uni-trier.de:8000/projects/ELM/elmart.html

Primary LISP Documentation

Guy Steele, *Common LISP the Language, 2nd edition*
Digital Press, 1990 ISBN 1-55558-041-6

online at:

http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ai-repository/ai/html/cltl/cltl2.html

Free LISP

Free LISP environments can be downloaded from one of these sources:

Listing of Free Sites

http://www.elwoodcorp.com/alu/table/systems.htm#free

Windows

Lite version of the company product (formerly Lucid CL)
http://www.harlequin.com/products/ads/freelisp/freelisp.html
Evaluation copy of the company product (Franz LISP)
http://www.franz.com/ "free stuff"
Public domain text-editor (emacs) written in LISP
http://www.cs.washington.edu/homes/voelker/ntemacs.html

MacIntosh

PowerLisp:
http://www.crl.com/~rgcorman/PowerLisp/PowerLisp.html

ftp://ftp.crl.com/users/rg/rgcorman/PowerLisp/

XLisp:

ftp://ftp.uwtc.washington.edu/pub/Mac/Programming/Languages/
ftp://ftp.switch.ch/software/mac/src/think_c/
ftp://ftp.uni-stuttgart.de/pub/systems/mac/misc/
ftp://ftp.leo.org/pub/comp/platforms/macintosh/dev/
ftp://ftp.informatik.tu-muenchen.de/pub/comp/platforms/macintosh/dev/

Or see the instructor.

Unix

Linux:

http://www.franz.com/ "free trial software"

A robsut LISP environment is built into the text editor ***EMACS***, at
http://www.geek-girl.com/emacs

AI Resources

http://www.cs.reading.ac.uk/people/dwc/ai.html
http://www.robotwisdom.com/ai/index.html
http://ai.iit.nrc.ca/ai_point.html
http://tqd.advanced.org/2705/
http://www.gmd.gr.jp/AI.CogSci.Robotics/
http://www.yahoo.com/Science/Computer_Science/Artificial_Intelligence/
http://www.excite.com/xdr/Computing_and_Internet/Computer_Science/Artificial_Intelligence/
http://www.cs.cmu.edu/groups/AI/html/repository.html

AAAI Professional Society

http://www.aaai.org/

Comments on LISP

The "artificial" in Artificial Intelligence means that the "intelligence" is implemented on a computer. Almost all AI implementations are in LISP; that is, LISP and AI are synonymous. The content of AI is defined by what you can write in the LISP language.

In this class, you will learn how to think in LISP, not how to be a LISP programmer. (This has made choosing a LISP primer difficult.) We will use a small subset of LISP called Pure Lisp.

PURE LISP = ideal programming style

Pure LISP excludes all of the programming ideas which lead to bad code. Pure LISP does not allow:

destructive data operations
go tos
explicit pointers and dereferencing
side effects (only the direct results of the function being processed)
unbound and global variables
do loops (use recursion instead, this rule is not firm)
block structure (functions provide grouping)

LISP is a unique language in the following ways:

- symbolic rather than numeric computation. This permits an algebraic approach.

- functional/applicative style rather than procedural style. Program structure is defined by function composition, and by the formal system of lambda calculus. The normal evaluation regime is successive function evaluation, but this can be freely modified

- indefinitely extensible. Most programming language innovations (such as garbage collection, streams, closures and continuations, symbol packages, first-class errors, object orientation, provability) were pioneered in LISP.

- interpreted/interactive rather than compiled. LISP can be compiled when debugging is completed. When you enter an expression into the LISP environment, the evaluation is immediate. LISP programming is a dialog.

- uniform data representation. Programs are data, which means LISP programs can modify themselves at run-time.

- LISP is written in LISP. The process of LISP is written in the same language as the rest of LISP. Bootstrapping means that the LISP evaluation mechanism and compiler are easily available to the programmer for modification and customization. In the days of LISP machines (1980s), the entire operating system was accessible. Emacs is a text editor written in LISP, the entire editing environment is accessible.

- very small kernel. The LISP primitives are:

nil	empty list, false, nothing to return
atom	predicate to determine elementary data

eq	predicate to test equality of atoms
car, cdr	selectors/accessors of data structure
cons	constructor for data structures
cond	basic logic function
eval, quote	controling the difference between program and data

- important functions

setq	setting or assigning names to the results of functions
list	constructing a list
defun	defining named functions
lambda	constructing an unnamed function
let	defining the scope of variables

- debugging tools include

whitespace	ignored by the evaluator
trace	follow the evaluation sequence
pprint	print data in pretty form
read-eval-print	the basic evaluation process

- disadvantages of LISP and their solutions

hard to read syntax with lots of parentheses
redefine the syntax to look the way you want it to

one data type
build the data types you want and wrap them in an abstraction barrier

inefficiency
no longer true, LISP runs at 95% the speed of C. It is possible to write inefficient LISP programs, but the rules to avoid this are straight forward and can be learned with practice. It is easier to write inefficient programs in other languages.

many dialects
the community has standardized on Common LISP. Dialects built from the same foundation are a good idea.

no first class functions
dialects for higher order programming are available (ie Scheme)

How To Think in LISP

What follows is a list of principles and ideas, not a formula for thinking.

- All valid expressions are valid programs. This provides arbitrary granularity.

Programming consists of building up hierarchical languages built on a solid foundation.

- You are always in control of what is data and what is process. Programming is building data, then testing processes on it, then making those processes into data, and so on.

- Computation consists of applying functions to data. Function composition lets you apply functions to the results of other functions, which themselves turn data into different data.

- All defined functions are provable, that is they are data structures you can talk about, and the way to talk about them is to assert their correctness.

- The programmer is always part of the computation. The read-eval-print loop can be seen as an interactive dialog. Read means listen to what the person says. Eval means do what the person asks you. Print means tell the person the results of the request.

- All objects are the same. There are base objects (atomic data) and compound objects built from atomic objects. Atomic objects (atoms) are the pieces of a program, the bricks. Function composition is the cement holding the atoms together. Nothing else is happening. Atoms define your conceptualization, the pieces of the world. Functions just define bigger pieces. Object-orientation is function composition turned inside out.

- Variables are just convenient and arbitrary names for compound objects. So a variable is meaningful only when it is in the same context as the object it names. This is called scoping.

- Function names are also variables. You can rename functions at any time, and you should always use names that are meaningful to you. Write languages not programs. Think like a human, not like a computer and write code that matches human thought.

- Recursion means stating a base case (what is true in the simplest circumstance) and a general case (what holds invariant across all objects being manipulated). The latter is not easy. Recursion makes control structure and data structure look the same, thus all cases of a particular data domain do not need to be examined exhaustively.

- There are always two levels when programming, the syntactic and the semantic: what you see and what you mean. Representation and value. Try to align the two by defining the look of a program to remind you of its meaning.

- Control is just telling the function how to look for what you what. The state space model is the space being searched. You can look in three different ways:
    * depth first (follow the chosen path to its end, then go back to the beginning to try other paths
    * breadth first (look one step down each path choice)
    * dependent (choose a path that meets your current objectives, then go back to the next best path to meet those objectives)

- Formulate knowledge in terms of patterns, and look for those patterns. Patterns can be abstract, with many things of the same class fitting a particular spot.

- Formulate operations as functions. Operations can be abstract, with many functions fitting the same operation. Use operations that address all objects at the same time. For example,

rather than explicitly checking each object for a property (by writing a DO loop), just ask if the property is true for everything (using the function EVERY).

LISP Debugging Techniques

1. Comments are delineated by a semi-colon on a given line. This makes the LISP reader skip to the next line. Larger comments can be delineated by "hash-bar":

```
#| put comments here |#
```

In general the hash-mark, #, is a special token to the LISP reader, telling it that the token or object which follows is a special type.

2. Select the important functions in each program, and use TRACE, PRINT, and BREAK to follow the calls to that function. If recursive programs do not trace in your system, either rename the recursive function with an indirect function name, as in

```
(defun fac (n) (if (= n 1) 1 (* n (fac1 (- n 1)))))
(defun fac1 (n) (fac n))
```

or insert a not-in-line declaration immediately after the argument list in the function definition:

```
(defun fac (n)
      (declare (notinline fac))
      (if (= n 1) 1 (* n (fac (- n 1)))))
```

3. TRACE is a system function which traces and prints out each call to the specific traced function(s). Invoke it by evaluating

```
(trace <function-name>)
```

at the prompt or in the code. Disable it with

```
(untrace <function-name>)
```

or

```
(untrace)
```

for all traced functions. This invaluable tool lets you watch the dynamic calling patterns of functions.

4. Inserting a PRINT statement within the code will print a specific message when the code execution passes the point of the print function.

```
(print (list <item1> .. <itemn>)
```

5. Just like inserting a PRINT statement into a function to see the current bindings of local variables, you can insert

```
(break "message")
```

which temporarily halts execution at the point of insertion. The "message" portion is actually a FORMAT form. FORMAT is an elaborate print statement used for fine grain control of output. All you need to know, though, is that special FORMAT directives within the quoted message automatically bind to values of provided local variables. The directive is written "~A". Experiment with these functions:

```
(break "hello")
(break "the value of foo is ~A" <variable-named-foo>)
(break "values: ~A ~A ~A" <var1> <var2> <var3>)
```

Your system will let you continue from a BREAK with a special keystroke (my MCL-LISP uses *Apple-/*). So BREAK is a way of building custom stepping. I've put some examples of this tool in the Filters code package. It is customary to turn debug tools on and off by putting a comment character ";" in front of them textually and re-evaluating the function. Of course you could use a global debug flag if you like, like this

```
(defvar *debug-flag* nil)   ;put this at the top of the code package
      (when *debug-flag*
      (print 'foo) (break "foo-break"))  ;put this inline
```

A caution: Some of the more sophisticated code in the distribution may contain some built-in functions that may not work on your system. (A previous example of this was the LOOP construct in the ITERATIVE-UNIFY algorithm.) I've attempted to make sure that all code runs in vanilla environments, so bring any obstacles to class.

LISP Program Modification Exercises

The following exercises are in approximate order of difficulty. Students wishing to achieve an "A" grade should attempt most of these exercises. These exercises constitute the bulk of homework and test assignments for the entire course. Most of these exercises ask you to extend the examples. As well as the exercises below, write your own examples. Explore!

All students: Load and run

1. Eliza
2. the recursive Unification algorithm
3. GPS for Blocks World
4. Parcil
5. Streams and filters
6. Logic Programming
7. Object-oriented
8. Parse

The Programs

1. **Eliza**, by Joseph Weizenbaum, transcribed by Peter Norvig.
A famous interactive dialog program, in the form of a Rogerian counselor. Contains a pattern-matcher. Consider how to extend the dialog manager.

2. **Unification and Search**, by Mark Kantrowitz, CMU.
Unification is an extended form of pattern-matching which is integral to inferences engines and query languages. Recursive and iterative versions are included for comparison. read the code, do not worry about the details. The search package is pedagogical, intended to show the similarities across all search algorithms. Although the code is complete, the knowledge representation is missing, so do not expect to run the entire code sample.

3. **GPS**, by Newell and Simon, CMU. Transcribed by Peter Norvig.
The first, famous General Problem Solver. Runs blocks world, so get familiar with the code. Don't forget to use debug mode.

4. **Parcil**, by Erann Gat, JPL.
A C syntax to LISP code recursive descent parser. Not industrial strength.

The next four programs focus on abstraction in coding. They are described in detail in another handout.

5. **Filters and Streams**, by Lugar and Stubblefield, from ideas by Curtis, Waters, and Steele.
Fundamentals of dataflow programming style.

6. **Logic Programming**, by Lugar and Stubblefield, with examples by Bricken.
Fundamentals of declarative logic programming style; an example of meta-linguistic abstraction.

7. **Object-oriented Programming**, by Lugar and Stubblefield.
Fundamentals of object-oriented programming style.

8. **Parse**, by students in Stanford LISP classes.
A recursive descent parser (similar to Parcil), for semi-natural English sentences.

Exercises

1. **Eliza**:

Extend the Eliza program to talk about your favorite subject.

2. **Unify**:

Describe the difference between
a. pattern matching for equality
b. pattern matching with variables
c. unification

3. **GPS**:

Extend the blocks world example to
a. use 4, 5, and 6 blocks
b. have limited table space (the trick is to name blocks which must stay on the table, and disallow moving to the table)
c. do the Towers of Hanoi puzzle
d. use other representations, such as an explicit Hand, an OnTable predicate, a ClearTop predicate.
e. Fix the Sussman anomaly.
f. Use boundary block representations (or others of your choice).

4. **Search**:

Write the missing generic-search functions for Optimal Paths.

5. **(harder) GPS**:

Modify GPS to use smarter search strategies, such as Hill-climbing, Best-first, or Branch-and-Bound.

6. **(harder) Search**:

Use the Kantrowitz generic-search functions to solve Blocks World by providing the knowledge representation as functions for initial-state, goal-p, and children. Evaluate which method is best, either by using the function TIME, or by counting the calls to TRACE.

Use generic-search or GPS to search the state space of some other problem domain, such as Missionaries-and-Cannibals, Sliding-Tiles, Tic-Tac-Toe, Cryptoarithmetic, or N-Queens.

7. (harder) Parcil:

Test Parcil on some real but restricted C code. Extend it to include some high-level C constructs.

8. Streams:

A. Write a stream/filter program to generate the first N prime numbers.

B. (harder) If you are fluent in another programming language, write the same function in that language and compare the ease of writing, the maintainability/extendibility and modularity, and the readability of each version. Write your other language program so that it handles an *arbitrary* N.

C. Change your program(s) to find the first N prime numbers with two occurrences of the digit 9.

D. Change your program(s) to find the first N prime Fibonacci numbers.

9. Logic Programming:

A. Substitute the normal FIRST/REST/CONS/EMPTYP functions for the (simulated) stream functions in the logic code. That is, expand the simulated stream abstraction inline to remove it. Does this improve readability for you?

B. Look at the problem of people liking themselves in the liking examples. Is there a better fix than the one proposed? Write some other liking rules which expose this problem.

C. Add some other facts (not rules) to the three animal databases to exercise the rules which are not used when asking questions about zeke, fred, and tony. What would you do if you wanted to know about types of animals (the animal taxonomy) but not about a particular named animal?

D. There is a major problem with the INFER function. What is it? Simplify the (first, smaller) addition database by removing commutative facts, and add the commutative rule:

```
(rule if ((var a) + (var b) = (var c) (var d))
      then
      ((var b) + (var a) = (var c) (var d)))
```

Figure out what is happening. (harder) Can you fix it?

E. Ask a question about ancestors in the relatives database and figure out what is happening. Ancestor is an example of a transitive relation. (This is related to question D.)

F. Write some other relative rules like GRANDMOTHER, GRANDPARENT, UNCLE.

G. Fix the sibling rule so that a person is not their own sibling.

H. Why is the fact

```
(1 + 0 = 0 1)
```

"out-of-order" in the first addition database? Organize the order of the facts and rules for maximum efficiency.

I. Write an addition rule which abstracts the carry operation out of the database. The trick is to use a carry-flag `((var x) = 1)` which unifies with the fact `(1 = 1)`.

J. (harder) Design an addition rulebase which handles addition of numbers of arbitrary magnitude.

K. (harder) Write a rulebase which solves Cryptoarithms (e.g. SEND+MORE=MONEY). The essence is to add rule(s) which enforce different numbers to be associated with each different letter. Then you might need rules to improve the efficiency of the solution process.

L. (longer but not harder) Write a rulebase for doing multiplication. For doing elementary algebra. For doing differential calculus.

10. Object-oriented Programming:

A. Get other object-oriented examples (from OO textbooks perhaps) and build them in the simple LISP system.

B. (harder) Figure out what you would need to do to include multiple inheritance.

11. Language Parsing:

A. Extend the grammar to include adverbs and pronouns. This will be (harder) if you have difficulty following the lambda forms in the code.

B. Incorporate PARSE-FIND into the code in place of FIND-* and FIND-?.

C. (harder) Write GENERIC-PARSE.

12. Miscellaneous:

A. Write at least five different versions of REVERSE. (harder) Write fifteen significantly different implementations.

B. (harder) Write a semantic net traversal algorithm. This is not hard if you use either the Logic Programming code or the Object-oriented code as a base.

C. Implement the simple robot-in-maze problem. Add the suggested extensions. When you get to adding the Wumpus, it gets (harder).

Artificial Intelligence: Introduction and History

Definition of Artificial Intelligence

Nilsson: the study of intelligent behavior (perception, reasoning, learning, communicating, acting in complex environments) in artifacts.

Ginsberg: the enterprise of constructing an intelligent artifact (a "physical symbol system" that can pass the Turing test)

Disciplines:
software engineering + computer science + philosophy = cognitive science

In reality:
neat explorations into advanced (theoretical, formal) programming techniques

Neats vs Scruffies:
formal logic and proof (the Stanford approach)
vs
experimental programming (the MIT approach)
neural nets, genetic algorithms, simulation

AI Subject Matter (logic + graphs)

Knowledge Representation

formal logic
proof theory
functional and declarative programming
the "Knowledge Level"

Search

non-polynomial algorithms (NP-complete)
state space
brute force (depth first, breadth first)
heuristics
objective functions (hill climbing, best first)
adversary search (taking turns)

History of AI

Aristotle	first formal symbol system	c325 BC
Copernicus	Earth is not the center of the universe	1543 AD
Descartes	Mind =/= Body	1641
Galileo	Math is a world model	1642
Euler	state space	1735
Boole	formal logic	1847
Babbage	analytic engine	1854

Frege	formal computation	1879
Russell & Whitehead	symbolic math foundation	1910
Tarski	theory of reference and meaning	1944
Turing	test of computational intelligence	1950

Formal Knowledge

A **conceptual model** consists of
- discrete objects, presumed to exist: the Universe of Discourse
- interrelations between objects
  - functions: compound names for objects and for unnamed objects
  - relations: truth statements about objects

No matter how the world is conceptualized, there are other conceptualizations that are just as useful.

Information Processing Systems

Information = representation + transformation
(program = data structure + processes)

All representations are content free. A theory of meaning must link representation to reality.

The Physical Symbol System Hypothesis

Newell and Simon, 1972: To resolve the mind-body problem of Descartes:

Minds and computers are physical systems which manipulate symbols.

The ***knowledge level*** is an abstraction layer for Computer Science which unites symbolic computation with world modeling
- hardware
- assembly, microcode, machine instructions
- programming languages
- algorithms and data-structures
- symbol level
- knowledge level

Principles of Representation

Symbol systems = patterns + process

Qualitative (symbolic) information
- not numerical (although this is just a stylistic difference)

Inference of "new" knowledge from a base of facts and rules

General principles of representation
variables, quantification, dynamic binding

Interaction and semantics
inheritance, context, theory of meaning

Meta-reasoning
knowledge about what is known or unknown

New control structures
learning from examples, explanation

Declarative Style

An ***AI program*** consists of
a set of objects
a set of functions (names for compound objects)
a set of relations (facts)
a set of permissible transformations

Objects and relations form a ***state***.
Transformations move between states.
Algorithms explore/search the ***state space***.
Programmers control the search.

State Space

The collection of facts (the database) at one given time defines the **state** of the world.
All possible state configurations define the **state space**.
To move from one state to another, apply a permitted **transformation rule**.
The state space and the moves between states from a **graph**.

Predicate Calculus

A *general purpose* language for describing objects, facts, and transformations for particular domains. Also called **First Order Logic**.

logic	{and, or, if, not, iff} inference, proof
object domains	{<unique atoms>}
quantification	{all.x, exists.x}
predicates	classes and properties
relations	connections between objects
functions	indirect names

Knowledge Representation

Objects: ***Names*** are intended to point ot actual concrete finite objects in the application domain. The choice of names is a part of interface/documentation. The actual names don't matter.

Variables: The ***patterns*** of linkage between variables with the same name determines the meaning of the variable.

Functions: Functions are ***indirect names***. They name objects by telling how to get to them. Functions are compound names.

Knowledge Representation Labels

Constants:

names of specific objects: John, Tuesday, My-Phone-Number
names of specific functions: House-of[x], Phone-of[x], Truth-of[p]
names of specific relations: Likes[Mary, Tom], Phone-Number[Tom, x]

Variables:

refer to sets/classes/domains of objects
always scoped/introduced by a quantifier

Knowledge Representation Atoms

Named objects	(object constants)
Indirect/compound named objects	(functions)
Relations between objects	(facts)

Logical connectives (and, or, not, if, iff) connect atoms.
They cannot be used inside atoms.

yes: eyes-of[John] AND hair-of[John}
no: (eyes-of AND hair-of)[John]
no: hair-of[John AND Mary]
yes: hair-of[John] AND hair-of[Mary]

Knowledge Representation Quantification

Variables name **classes of objects** (all.x) and **arbitrary objects** from a class (exists.x).

Variables are bound to specific objects by the act of **instantiation**.

Quantification provides a mechanism to refer to entire classes and to arbitrary objects.

all.x P[x]	every x for which P[x] is True
exists.x P[x]	some arbitrary x for which P[x] is True

Model Theory

Given an object domain and a collection of functions and relations on objects in that domain, a ***model*** of the domain is defined by the facts:

all atoms (atom-names) are True
all atoms not in the domain are False

Eg: Domain = {Mary, Tom, John} Relation: {Likes}

all possible states:

Likes[Mary, Tom]
Likes[Mary, John]
Likes[John, Mary]
Likes[Tom, Mary]
Likes[John, Tom]
Likes[Tom, John]

all possible models:

1	empty (no relations true)
6	one Likes relation isTrue
15	two Likes relations True
20	three True
15	four True
6	five True
1	six True
64	possible models in total

Entailment (implication) and Computability

P |= Q (double turnstile)

All models for which the collection of facts in P are True imply that the collection of facts in Q are True for every model.

P |- Q (single turnstile)

Using the rules of a given system, we can compute Q from P.

The central issue (1920-1970): Just because we can compute it (P |- Q), does that mean that what we compute matches our model (P |= Q), the ways in which the world can actually be?

Correctness

Sound: if P |- Q, then P |= Q
A sound computation always supports the world model.

Complete: if P |= Q, then P |- Q

A complete computation always generates all possible models.

Sound and Complete: P |- Q = P |= Q
The model and the computation represent the same world.

Decidability

Universal: if it can be stated formally, then it can be stated in First Order Logic.

Decidable: the computational procedure will terminate with a Yes/No result.

Semi-decidable: The computation might halt, but you don't know when. It may never halt if you ask the wrong knd of question. What we can't do is ask if questions which depend on the **failure** to prove something:

No: "Check to see if nothing is wrong"
No: "Prove that this search will fail to find X"

Abstraction Techniques

One way to describe LISP programming (in particular) and AI (in general) is

the art of successive abstraction

That is, LISP programming is writing low level functions, then writing higher level programs which use the lower level functions as basic building blocks, then writing languages which use the high level programs as building blocks, etc.

A primary example of abstraction is the UNIFY algorithm, which takes concrete facts (written in a relational form of predicate calculus) and abstracts them into arbitrary patterns which relate sets of facts. UNIFY does this by introducing pattern variables. The ELIZA program uses pattern variables to turn normal English sentences into an apparently therapeutic dialog. So, for example, the three declarative facts

```
(John likes Mary)
(John likes Tom)
(John likes Sarah)
```

can be abstracted into a single idea of "the people John likes":

```
(John likes ?x)
```

A pattern matching engine collects the basic facts that match the pattern query, saving the user from having to ask "Does John like Mary?", "Does John like Tom?", "Does John like Sam?", etc. Pattern variable abstraction is entirely general, if you know the form of the facts in the database, you can ask for any particulars, including, for example, the general query about all facts relating three labels:

```
(?x ?y ?z)
```

Another example of abstraction is the GENERIC-SEARCH algorithm, which takes concrete state spaces (written as sets of state descriptions in a form of predicate calculus) and concrete state transformations (written as changes to the current set of state descriptions) and abstracts both of them into arbitrary patterns of traversal of state space. GENERIC-SEARCH does this by treating traversal functions as next-state priority queues, and introducing state evaluation functions to decide the traversal priorities. ELIZA, PARCIL and GPS can each be easily modified to search their respective state spaces more intelligently by using the techniques of GENERIC-SEARCH.

Our LISP code distribution contains examples of many different tools for abstraction of programming objectives. The distribution emphasizes several programming techniques for saying more with less. The philosophy is that when the problems are very difficult, it is more important for the program to be understandable than to be efficient. Another deep programming technique, however, is that code optimization itself is an automated transformation. In compiling code, the programmer is not expected to read and debug the resulting machine code. Similarly, in LISP programming, the programmer can express the desired functionality at the highest level of abstraction *without paying a performance cost*. In the language of SE553, the

programmer should focus on expressing the mathematical model of the conceptualization of the problem, building abstraction tools rather than implementing low level optimizations.

AI emphasizes conceptual understanding by providing abstract implementation tools and languages. The LISP code distribution includes the following packages for applying abstraction to programming style.

I. FILTERS AND STREAM FUNCTIONS WITH DELAYED EVALUATION

This package illustrates the techniques of *dataflow programming*. A program is conceptualized as a stream of data which flows through filters that remove unwanted data structures. The "results" flow out the other end of the stream. This technique is incorporated deeply in the Mathematica language, and is essential to the mathematical (or functional) programming style. New programming tools include BACKQUOTE and DEFMACRO.

BACKQUOTE is just like QUOTE in that the form following the backquote is not evaluated, being treated as literal. However inside BACKQUOTE, any forms preceded by the function COMMA are evaluated, in the context that the backquote form is in. This permits passing structured forms, including functional forms, into unknown environments and having them behave appropriately.

DEFMACRO is just like DEFUN in that it defines a function. However, the macro-function being defined is expanded inline; when a function which uses a macro is loaded, the macro code is automatically written in place of the macro name. Macros permit naming and abstraction of code components without the overhead of introducing new function calls.

To make stream objects which generate data items only when needed, BACKQUOTE and DEFMACRO are combined. For instance the macro-function DELAY:

```
(defmacro delay (exp) `(function (lambda () ,exp)))
```

places a function "shell" inline wherever the function DELAY is used. The arguments of DELAY form the body of the function shell. Thus CONS-STREAM

```
(defmacro cons-stream (new stream) `(cons ,new (delay ,stream)))
```

provides the function body STREAM to DELAY. (CONS-STREAM itself is a macro) Like CONS, CONS-STREAM adds a new element to the front of a stream, but the stream itself does not yet exist as data, only as potentia. When CONS-STREAM occurs in a function body, it turns into

```
(cons <new> (function (lambda () <stream>)))
```

with `<new>` and `<stream>` being provided by the function which uses CONS-STREAM. Since CONS-STREAM is a macro, whenever it is used, LISP builds an "anonymous" function in place of the code `(function (lambda () <stream>))`, which makes the code look like

```
(cons <new> #<Anonymous Function #x267F5A6)
```

Finally the function FORCE, called by REST-STREAM, actually evaluates the anonymous function which has been defined in the context of the constructor CONS-STREAM:

```
(defun force (function-closure) (funcall function-closure))
(defun rest-stream (stream) (force (rest stream)))
```

Thus the set of constructor and accessor functions (CONS, FIRST, REST) turn into *generators*. CONS-STREAM builds a "list" of data consisting of a first element and a function which generates the next element. When REST-STREAM is called in the course of a computation, the generator function is evaluated by FUNCALL to build the next data element. Exactly what data element it builds is determined by the function which uses CONS-STREAM. Notice the subtlety that the anonymous generator function itself is not a static entity, it contains within it the code to build the next generator function, not just the next data element.

The stream abstraction permits programs to accumulate or process collections that we do not know how to count at compile time. The code example, for instance, asks for the first N Fibonacci numbers which meet an arbitrary test function, such as the first 20 odd Fibonacci numbers, or the first 8 Fibonacci numbers containing the digit 5 at least three times.

To accumulate items from a stream, we place a filter function in the stream. The final abstraction is the function ACCUMULATE-FILTERED-STREAM, which takes as parameters a stream generating function, a filter function, and a desired number of items which pass the `filter`.

II. LOGIC PROGRAMMING

This package illustrates *metalinguistic abstraction*, or declarative programming. A program is expressed in a descriptive language (in predicate calculus), and the language interpreter applies pattern matching to provide solutions. The abstraction is at the level of language, in that the interpreter is not sensitive to the form or the vocabulary of the language used to represent and describe the problem domain.

Yet another UNIFY algorithm is included in this package. Note that variables are expressed as a list `(var x)`. Of course the `x` can be any label, since the program takes care of providing unique names for all variables

To use the program, type `(logic-shell)` to LISP. A new top-level interpreter is called, which recognizes logical queries. Be sure to load an assertions database before entering the LOGIC-SHELL.

Note: when loading new *ASSERTIONS*, be sure to exit the logic-shell first.
Note: The *DONE* signal alone is returned when there are no solutions.

III. OBJECT-ORIENTED PROGRAMMING

Objects, messages, and inheritance are an alternative declarative organizational structure for programming. By seeing the internal mechanisms of object-oriented programming, you can understand that object-orientation is simply a different way of organizing and accessing a database. The abstraction is in the *programming metaphor*. Our culture is object-oriented (as

opposed to process oriented), so we think more efficiently in terms of objects and their interactions, than in terms of pure functionality.

The model of process as *messages passing between discrete objects* is at the foundations of Shannon's theory of information, which greatly influenced the origins of Computer Science. Object-orientation is excellent for digital systems which have been designed as discrete objects sharing communication lines, but it is weak for process-oriented models (such as a lighthouse sweeping a beam across the ocean and the waves breaking upon the shore) and for all biological and human systems.

The LISP code for object-oriented programming makes extensive use of symbol databases. Every symbol in a LISP environment has its own database. The function

```
(get <object-name> '<property>)
```

retrieves the value of a property associated with the object-name, whether or not the object is an instance or a class or a data-structure. The function

```
(setf (get <object-name> '<property>) <value>)
```

assigns a value to a property of a given object-name. Although we think of objects and classes, the LISP system treats all "things" as symbols, that is, as names. The function

```
(symbol-plist <name>)
```

displays an object-name's properties and their current values.

Methods are treated the same way, as properties associated with symbols. The difference is that the value of a method is a lambda form, that is, an abstract function waiting to have its formal parameters bound to values.

The primary tools of abstraction in object systems are polymorphism and inheritance. Inheritance is a search of each symbol's database, following the ISA property to the database of classes and superclasses. Polymorphism is having a separate name-space for different databases of different symbols.

The function BUILD-ROOT provides some elementary methods (i.e. lambda forms organized under property names) for the root-object ROOT These are

ROOT	
SHOW	<print the ISA, VARIABLES and METHODS properties>
SHOW-PARENTS	<print the ISA values>
SHOW-VALUE	<evaluate the name given as the argument to show-value>
SHOW-ENV	<build the environment of SELF>
SET-VALUE	<given property and value arguments, set the value>

To use an object environment, build the environment, then send messages to the objects about their properties.

IV. SEMI-NATURAL LANGUAGE PARSING

PARSE analyzes an English sentence in the same way that the English classes teach (or at least taught 30 years ago) students how to diagram a sentence. The recursive descent parser uses a technique similar to the one PARCIL uses for the C language. It is an example of top-down programming, each component can be written without knowledge of what the other components do.

English sentences are defined to follow a *grammar*, that is, a theory of structure. The code disassembles its input (i.e. sentences) by following the rules of the grammar. This implementation assumes that an acceptable sentence can be decomposed into compound fragments. The compound fragments can be composed of other compound fragments, but eventually everything is composed of simple words. The words are listed in a *lexicon*, a dictionary which classifies words into their types. (Obviously natural English is far more complex. The code sample generalizes to the construction of arbitrary grammars.)

COMPOUND FRAGMENTS

```
<S>   -->  <NP> <VP>
<NP>  -->  [<detr>] <adjv>* <noun> <PP>*
<VP>  -->  <vaux>* <verb> [<NP>] <PP>*
<PP>  -->  <prep> <NP>
```

`*` means the part may occur zero or more times
`[...]` means the part may occur zero or one time

Abstraction	*Part-of-grammar*	*Function which analyzes the fragment*
`<S>`	Sentence	`(sentence <s>)`
`<NP>`	Noun Phrase	`(noun-phrase <np>)`
`<VP>`	Verb Phrase	`(verb-phrase <vp>)`
`<PP>`	Prepositional Phrase	`(prep-phrase <pp>)`

SIMPLE WORDS

Abstraction	*Part-of-grammar*	*Function which accesses lexicon*
`<noun>`	Noun	`(noun <w>)`
`<detr>`	Determiner	`(detr <w>)`
`<verb>`	Verb	`(verb <w>)`
`<prep>`	Preposition	`(prep <w>)`
`<vaux>`	Verb Auxiliary	`(vaux <w>)`
`<adjv>`	Adjective	`(adjv <w>)`

It is tempting to put the input sentence into a global variable which is used up from left to right by the parser. However the parser needs to access each fragment twice, once to determine its type and once to build the parse tree. Since the type of the fragment is unknown when it is first accessed, this strategy requires a host of other supporting global variables, leading to difficult

to debug dependencies between parsing functions. Failures in guessing the correct type lead to backtracking, that is to unwinding previous results, a nightmare for data structure management. But the major problem is that compound fragments can contain other compound fragments recursively, so that a function which assigns a value to a global variable might be called again within the same sentence, losing the values assigned in the outer context. For example, our grammar permits NP to call PP which may have an NP in it.

Assignment statements (that is, SETQ) work differently each time you use them because the calling context (the binding of local variables) is different. This undermines modularity by creating dependencies and interactions between program segments. The solution is to use LAMBDA forms which do not bind function arguments until the appropriate time (similar to the way STREAMS work, but simpler.) A LAMBDA has the form

```
(lambda (formal-parameters) <body>)
```

It can be conceptualized as an *abstract operator*, independent of its arguments, with the operation defined by the body of the LAMBDA. A *function-call* is the form

```
((lambda (formal-parameters) <body>) <contextual-values>)
```

The evaluation of a functional form consists of evaluating the arguments, then substituting the values of the arguments for their formal parameters in the body of the LAMBDA, then evaluating the LAMBDA body itself. For example, the function PARSE-SENTENCE has the form

```
((lambda (s-pair) <body-which-uses-s-pair>) (sentence s))
```

First the function SENTENCE processes the input sentence, then that result is substituted into the LAMBDA body and the result is evaluated.

Similarly, the function SENTENCE has the form

```
((lambda (np-pair) <body>) (noun-phrase s))
```

This calls NOUN-PHRASE to process the beginning of the sentence. Whatever the result (for instance the sentence may start with a verb phrase and not with a noun phrase), it is handed in an appropriate form to the next function to continue parsing.

The parsed results of a fragment processing function are conveyed to the next processing function simply by CONSing them onto the sentence being processed. This is the meaning of "s-pair", "np-pair", etc. The FIRST of any current fragment is the results of prior processing, the REST is the parts yet to be processed. Thus the output functions can build the results before all of the results are known, and all backtracking is eliminated. And the output functions are precisely the body of each LAMBDA form. Finally, all fragment processors use the same technique, thus all the functions are similar.

The only remaining complexity is to handle the grammar modifiers `*` and `[...]`. The functions FIND-* and FIND-? are abstracted to handle any type of sentence fragment. In the current grammar, `*` applies only to the set {adjective, prepositional-phrase, verb-auxiliary}, while `[...]` applies to the set {determiner, noun-phrase}. Note that `[...]` is renamed `?` since LISP already has an assigned meaning for square brackets. Changing the grammar is easy when the

language features are abstracted away from the particular instances they are used for. Of course, FIND-* uses the same LAMBDA technique as the rest of the parser. FIND-?, which identifies zero or one occurrence, calls the type being tested. If the parse fails, it CONSes NIL to indicate that no occurrence was found.

In summary, here are some things to notice about the PARSE code:

1. No assignment statements, so no variables which you have to track how they are being used.

2. No special treatment for any parsing categories, code uniformity = code maintainability/readability/modifiability

3. No backtracking or storage of temporary results, thus easy tracing and debugging.

4. The * and [...] types are abstracted into FIND-* and FIND-? functions, making extensibility easy.

5. No global variables, so functions have no interaction and can be modified modularly.

Yet more abstraction is available, but not fully implemented. Our grammar has three occurrence types,

`<...>`	for simple words and compound forms
`<...>*`	for sequences of zero or more forms
`[...]`	for zero or one occurrences of a word or form

The function PARSE-FIND removes the code dependency of each special occurrence form, by managing the metalanguage of the grammar. This leads to easy extensibility, not of the grammar, but of the grammar's metalanguage. You may, for example, wish to establish an occurrence operator `!` which calls for exactly one occurrence of a type, zero and more than one occurrence being trapped as ungrammatical. Sentence punctuation requires this occurrence rule.

Finally (finally!) a function which is not difficult to write, but not included, is GENERIC-PARSE, which takes a sentence and a grammar. The grammar data structure may look like

```
(defvar current-grammar
    '((SENTENCE (NP VP))
      (NP   ((? DETR) (* ADJV) NOUN (* PP)))
      (VP   ((* VAUX) VERB (* PP) (? NP)))
      (PP   (PREP NP))
      (NOUN (member-of *nouns*))
      (DETR (member-of *determiners*))
      (VERB (member-of *verbs*))
      (PREP (member-of *prepositions*))
      (VAUX (member-of *verb-auxiliaries*))
      (ADJV (member-of *adjectives*)) ))
```

GENERIC-PARSE is a fully abstract tool for exploring types of parsing grammars. Here is where AI research begins, since the metalinguistic abstractions provided by the code make it easy to experiment with a variety of grammars and parsing metalanguages. A researcher would build a database of sentences, and then build grammars which parsed the database. These parsings would be passed to a statistics collection program which would report of the success of each grammar. The (happy) researcher spends all his/her time at the conceptualization and modeling levels, and before long, a book of results is collected.

V. MISCELLANEOUS

1. Here is the recursive REVERSE which I did not do in class:

```
(defun reverse553 (lst)
      (if (atom lst) lst
      (append (reverse553 (rest lst))
      (list (reverse553 (first lst))))))
```

2. Here is how to build a semantic net in LISP. You might try drawing the graph of this birds and animals example (the spacing is for readability only. LISP ignores whitespace.):

```
(setf (get 'animal  'covering) 'skin)
(setf (get 'bird    'covering) 'feathers)
(setf (get 'bird    'travel)   'flies)
(setf (get 'bird    'isa)      'animal)
(setf (get 'fish    'isa)      'animal)
(setf (get 'fish    'travel)   'swims)
(setf (get 'ostrich 'isa)      'bird)
(setf (get 'ostrich 'travel)   'walks)
(setf (get 'penguin 'isa)      'bird)
(setf (get 'penguin 'travel)   'walks)
(setf (get 'penguin 'color)    'brown)
(setf (get 'opus    'isa)      'penguin)
(setf (get 'canary  'isa)      'bird)
(setf (get 'canary  'color)    'yellow)
(setf (get 'canary  'sound)    'sings)
(setf (get 'tweety  'isa)      'canary)
(setf (get 'tweety  'color)    'white)
(setf (get 'robin  'isa)       'bird)
(setf (get 'robin  'sound)     'sings)
(setf (get 'robin  'color)     'red)
```

3. Here is planning example that is not blocks world. The idea is to find a plan that allows the robot to get the key and take it to the exit. You should be able to program the solution to this in either GPS or the logic engine.

```
(caves c1..c18)          ;a listing of all 18 caves
(tunnel c1  c7)          ;the tunnels between caves
```

```
(tunnel c1  c2)
(tunnel c2  c8)
(tunnel c3  c9)
(tunnel c4  c10)
(tunnel c6  c12)
(tunnel c8  c9)
(tunnel c8  c14)
(tunnel c9  c10)
(tunnel c9  c15)
(tunnel c11 c12)
(tunnel c10 c16)
(tunnel c11 c17)
(tunnel c12 c18)
(tunnel c13 c14)
(tunnel c15 c16)

(robot in 3 at 0)        ;initial locations of robot, key and exit at time 0
(key   in 4 at 0)
(exit in 16 at 0)

(move ci cj)             ;movement action
(grab  x ci)             ;pick up action
(drop  x ci)             ;put down action is not needed in simple game
(open ci)                ;use key action
```

This is the framework of a planning game called WUMPUS. To make a more interesting game, you could extend this framework by

1. have a ladder that is necessary for some (vertical) tunnels
2. make the robot put down the key when holding the ladder
3. add new tunnels (digging) at selected or random times;
   delete some tunnels (cave-ins) at selected times.
4. have another creature (the WUMPUS) moving around the tunnel maze, which the robot can hear when in an adjacent cave. If the WUMPUS is in the same cave, it eats the robot, or steals the key/ladder (putting it in some random place).
5. add gold in certain caves and require the robot to collect an amount of gold before exiting.
6. add teleportation from a specific cave to a random cave
7. add a second robot and have a contest to see who exits first.
8. have an automated robot and a robot moving under user control
9. add friendly natives in some caves which give hints; add some natives which give false hints.

The idea, of course, is that with a free week, class members have sufficient skills to model these extensions and implement WUMPUS within the context of either GPS, or LOGIC ENGINE, or OBJECTS. By combining PARSE and ELIZA, you could even have the robot talking with the natives in the caves.

Issues in Knowledge Engineering

Definition of Artificial Intelligence

Ginsberg: the enterprise of constructing an intelligent artifact
(a "physical symbol system" that can pass the Turing test)

Disciplines:
software engineering + computer science + philosophy = cognitive science

In reality:
neat explorations into advanced (theoretical, formal) programming techniques

Neats vs Scruffies:
formal logic and proof (the Stanford approach)
vs
experimental programming (the MIT approach)
neural nets, genetic algorithms, simulation

AI Subject Matter (logic + graphs)

Knowledge Representation
formal logic
proof theory
functional and declarative programming
the "Knowledge Level"

Search
non-polynomial algorithms (NP-complete)
state space
brute force (depth first, breadth first)
heuristics
objective functions (hill climbing, best first)
adversary search (taking turns)

History of AI

Aristotle	first formal symbol system	c325 BC
Copernicus	Earth is not the center of the universe	1543 AD
Descartes	Mind =/= Body	1641
Galileo	Math is a world model	1642
Euler	state space	1735
Boole	formal logic	1847
Babbage	analytic engine	1854
Frege	formal computation	1879
Russell & Whitehead	symbolic math foundation	1910
Tarski	theory of reference and meaning	1944
Turing	test of computational intelligence	1950

Formal Knowledge

A **conceptual model** consists of
- discrete objects, presumed to exist: the Universe of Discourse
- interrelations between objects
  - functions: compound names for objects and for unnamed objects
  - relations: truth statements about objects

No matter how the world is conceptualized, other conceptualizations that are just as useful.

Information Processing Systems

Information = representation + transformation
(program = data structure + processes)

All representations are content free. A theory of meaning must link representation to reality.

The Physical Symbol System Hypothesis

Newell and Simon, 1972: To resolve the mind-body problem of Descartes:
Minds and computers are physical systems which manipulate symbols.

The **knowledge level** is an abstraction layer for Computer Science which unites symbolic computation with world modeling
- hardware
- assembly, microcode, machine instructions
- programming languages
- algorithms and data-structures
- symbol level
- knowledge level

Principles of Representation

Symbol systems = patterns + process

Qualitative (symbolic) information
- not numerical (although this is just a stylistic difference)

Inference of "new" knowledge from a base of facts and rules

General principles of representation
- variables, quantification, dynamic binding

Interaction and semantics
- inheritance, context, theory of meaning

Meta-reasoning
- knowledge about what is known or unknown

New control structures
- learning from examples, explanation

Declarative Style

An **AI program** consists of
- a set of objects
- a set of functions (names for compound objects)
- a set of relations (facts)
- a set of permissible transformations

Objects and relations form a **state**.
Transformations move between states.
Algorithms explore/search the **state space**.
Programmers control the search.

State Space

The collection of facts (the database) at one given time defines the **state** of the world.
All possible state configurations define the **state space**.
To move from one state to another, apply a permitted **transformation rule**.
The state space and the moves between states from a **graph**.

Predicate Calculus

A **general purpose** language for describing objects, facts, and transformations for particular domains. Also called **First Order Logic**.

logic	{and, or, if, not, iff} inference, proof
object domains	{<unique atoms>}
quantification	{all.x, exists.x}
predicates	classes and properties
relations	connections between objects
functions	indirect names

Knowledge Representation

Objects: Names are intended to point ot actual concrete finite objects in the application domain. The choice of names is a part of interface/documentation. The actual names don't matter.

Variables: The **patterns** of linkage between variables with the same name determines the meaning of the variable.

Functions: Functions are **indirect names**. They name objects by telling how to get to them. Functions are compound names.

Knowledge Representation Labels

Constants:

names of specific objects: John, Tuesday, My-Phone-Number
names of specific functions: House-of[x], Phone-of[x], Truth-of[p]
names of specific relations: Likes[Mary, Tom], Phone-Number[Tom, x]

Variables:

refer to sets/classes/domains of objects
always scoped/introduced by a quantifier

Knowledge Representation Atoms

Named objects	(object constants)
Indirect/compound named objects	(functions)
Relations between objects	(facts)

Logical connectives (and, or, not, if, iff) connect atoms. They cannot be used inside atoms.

yes: eyes-of[John] AND hair-of[John}
no: (eyes-of AND hair-of)[John]
no: hair-of[John AND Mary]
yes: hair-of[John] AND hair-of[Mary]

Knowledge Representation Quantification

Variables name **classes of objects** (all.x) and **arbitrary objects** from a class (exists.x).

Variables are bound to specific objects by the act of **instantiation**.

Quantification provides a mechanism to refer to entire classes and to arbitrary objects.

all.x P[x]	every x for which P[x] is True
exists.x P[x]	some arbitrary x for which P[x] is True

Model Theory

Given an object domain and a collection of functions and relations on objects in that domain, a **model** of the domain is defined by the facts:
all atoms (atom-names) are True
all atoms not in the domain are False

Eg: Domain = {Mary, Tom, John} Relation: {Likes}

all possible states: all possible models:

Likes[Mary, Tom]	1	empty (no relations true)
Likes[Mary, John]	6	one Likes relation isTrue
Likes[John, Mary]	15	two Likes relations True
Likes[Tom, Mary]	20	three True
Likes[John, Tom]	15	four True
Likes[Tom, John]	6	five True
	1	six True
	64	possible models in total

Entailment (implication) and Computability

P |= Q (double turnstile)

All models for which the collection of facts in P are True imply that the collection of facts in Q are True for every model.

P |- Q (single turnstile)

Using the rules of a given system, we can compute Q from P.

The central issue (1920-1970): Just because we can compute it (P |- Q), does that mean that what we compute matches our model (P |= Q), the ways in which the world can actually be?

Correctness

Sound: if P |- Q, then P |= Q
A sound computation always supports the world model.

Complete: if P |= Q, then P |- Q
A complete computation always generates all possible models.

Sound and Complete: P |- Q = P |= Q
The model and the computation represent the same world.

Decidability

Universal: if it can be stated formally, then it can be stated in First Order Logic.

Decidable: the computational procedure will terminate with a Yes/No result.

Semi-decidable: The computation might halt, but you don't know when. It may never halt if you ask the wrong knd of question. What we can't do is ask if questions which depend on the **failure** to prove something:

No: "Check to see if nothing is wrong"
No: "Prove that this search will fail to find X"

Representation =?= Reality

We have seen that

$$|- \;=\; |=$$

that is, that computation and formal models of reality are the same thing.

And that the LISP programming language is the flexible substrate for both representation and computation. When the field of AI began to grow in 1980s, the challenge was to actually express our models of reality in code. This turned out to be exceedingly difficult. It is not that computation is a particularly weak point, it is that our formal languages do not let us speak easily of reality.

Jerome Bruner, in his book ***Actual Minds, Possible Worlds***, shows that the relationships between actual reality, language, and understanding are complex and not formal. He notes that understanding is always in context, meaning is always ambiguous. There is no reality independent of mental activity and symbolic language. We know the world and construct meaning through multiple perspectives, including emotion, culture, language, and stories.

The failure of AI to be able to parse and translate natural languages underscores this problem. What then can we model formally? The field of AI believes that sooner or later, everything can be modelled. What are the technical difficulties in trying to model anything (like making simple stacks of blocks, for example)?

Technical Difficulties in Modeling and Knowledge Representaiton

(using blocks world in LISP as an example)

1. What is important to describe?
   Build little theories of little worlds.

```
(Block A)      (OnTable A)      (Hand Empty)
```

2. How should descriptions be partitioned?
   Functions or Relations, special or general objects?

```
(OnTable A)      (On A Table)      (not (OnTable Table))
```

3. How do we talk about groups and classes of objects?
   Quantification and abstraction

```
(All (x) (Block x))
```

4. How do we address things with no names?

Functions as indirect, compound names.

```
(House-of John)
```

5. How do we handle things with more than one name?
   unique name hypothesis, unification

```
(Uncle John) = (Brother (Father John)) = Bob
```

6. How do we make general rules which define the structure of relations?
   quantification

```
(All (x) (iff (Uncle x) (Brother (Father x))))
```

7. How are typing and filters on domains represented?
   predicates in conjunction

```
(All (x) (and (Person x) (Father x y)))
```

8. How do we join more than one fact?
   conjunction

```
(and (F x) (G x))
```

9. How do we compute with logic?
   inference as natural deduction and as resolution

```
(if (and (P x) (if (P x) (Q x))) (Q x))
```

10. How do we compute with quantifiers and classes of objects?
    implicit universal quantification, Skolemization

```
(Exist (x) (P x)) ==> (P (Sk-1 x))
```

11. What is the difference between a fact and a query?
    query combination rules

    A. conjunction with negated query

```
(and (P x) (not (Q ?)))
```

    B. Skolemization of query variables

```
(Q ?) ==> (Q Sk-1)
```

    C. Facts imply Query

```
(if (P x) (Q ?))
```

    D. The answer predicate

```
(if (P x) (Answer x))
```

12. What kinds of rules do we need for query answering?

    A. definitions

    ```
    (iff (P x) (Q (R x)))
    ```

    B. mathematical structures (symmetry, transitivity, etc)

    ```
    (if (and (if (P x) (Q x)) (if (Q x) (R x))) (R x))
    ```

    C. permissible state transformations

    ```
    (Pick-up x) = (Assert (not (onTable x)))
    ```

13. How can we control the inference/search procedure?

    A. Pre and Post conditions

    B. Compound queries

    C. Searching databases of rules and facts

14. How do we steer the resolution process?

    A. set of support

    B. ordered resoultion

    C. static vs dynamic approaches (compiled vs run-time)

    D. lookahead, cheapest first, dependency directed search

15. How do we express meta-level reasoning (rules about rules)
    measure the savings vs brute force

16. What is the appropriate reasoning strategy?

    look-up tables
    natural deduction
    resolution
    forward or backward chaining
    simulation
    boundary logic

17. What do we do about contradictions and inconsistencies?

    A. Forbid them

B. Default reasoning

C. Exception handlers

D. Three-valued logic (True, False, Inconsistent)

E. Multi-valued logic (eg True, False, Contradictory, Meaningless)

F. Contradiction maintenance

Natural Language Processing

Language

lexicon	the words and their types
syntax	rules for making compound structures with words
semantics	how words and structures map onto their intended meanings, associating the words with the conceptualization
pragmatics	use of words and structures in the context of communication acts

Language structures are not linear, all components interact.

Computational Linguistics (Chomsky) and Syntax

The syntax of a language can be structured using rewrite rules. (See the PARSE program) English language rewrite systems have about 200 rules and about 200 restrictions (special cases, constraints). To address verb tenses, subject number, sentence types and subject-verb agreement, Chomsky developed the theory of deep structure. Deep structure is what holds constant in an utterance independent of the tense, agreements, etc. A parser works with deep structure, then modifies the shallow structure to make the sentence parts agree.

Examples of ambiguity:

"I'll go when I see Sue and John leave(s)."
"The fish love(s) Mary."
"The {patient, janitor} left the operating room in good condition."
"I saw the man in the park with the telescope."
"I saw the ferry flying into Seattle."
"I ate dinner with a {fork, friend}."
"Can companies litter the environment {. ?}"
"Time flies like an arrow. Fruit flies lika a banana."

Semantics

Most grammars determine semantics by mapping the parsed syntax onto Predicate Calculus. Eg: "What is John's grade in Math-101?" ==> (grade ?x John Math-101) The essential question is whether or not what we talk about can be captured in predicate logic. The current prespective in the 1990s is: No, because

1. Most knowledge is uncertain and vague ("France is hexagonal")
2. We know too much, and we know what we don't know.
3. Language is contextual and culturally embedded.
4. Language inherently incorporates multiple perspectives.
5. Meaning is unique, interpretative, normative, interactive, conversational, evolutionary and wholistic.

The referents of pronouns are contextually determined. Examples:

"My name is Tom. I am hungry. I want to eat now."
"John can open Bill's safe. He knows the combination. He's going to change it."

Indexicals

Many words change meaning, or referent, every time they are utterred. Pronouns and words identifying relative time and place provide examples:

"I am hungry."	The 'I' is whomever utters the sentence.
"Give it to me."	The 'me' is whomever utters the sentence.
"Do it now."	The 'now' changes it temporal referent every time it is used.
"It is over there."	The 'there' can be a different place each time it is used.
"I am here; you are there."	'Here' and 'there' are defined by the context of usage.

Pragmatics

Contextual usage is very non-logical and elliptic. Eg:

"Use the racket-wrench. What's that? It's in the drawer."

The dialog aboves implies a model of the situation, something like: to fix an appliance, use a racket-wrench, so find one, recognize it, and ask how to use it when you can see it. More egs:

"Fred went into the market. He found the milk. He gave the clerk some money and left."
"Fred went to the market. He found the shelf with the milk. He paid for it and left."
"Sally needed some money.
{She bought a gun. She phoned her mother. She wrote a proposal for the NSF.}"
"He paid the ten dollars. He wouldn't let her give him five. So she bought the popcorn."
"Can you open the door? I'm cold."
"I want a hamburger. {Yes sir! No, not today. Me too.}"

Dialog Phenomena

anaphora: pronouns and words that refer to something prior in the dialog.

definite noun phrases: compound names with embedded semantics
(eg: outdated math books)

ellipsis: fragments of sentences, missing words in a terse dialog.

extragrammatical utterances: interjections, false starts, etc.

metalinguistic utterances: words about the conversations (eg: "I should have said...")

Taxonomy of Speech Processing

Generation (voice response)
Transmission (digital)
Language recognition
Speaker recognition (verify and identify)
Speaker analysis (speech pathologies, psychological state)
Speech recognition
    isolated word recognition
    word finding
    continuous speech recognition

Transformations to spoken words

microphone or channel characteristics
ambient noise in the environment
articulation
phonemic level
prosodic level (tempo, intonation)
lexicon
syntax level
discourse rules
sematics level
psychological state

Speech Acts

Some sentences are not descriptive or declarative, but are themselves actions. Eg: "I bet you $10 they win."

locutionary acts — making sounds (sighing, grunting, exclaming, etc.)

illocutionary acts — the intent of the sentence (asking, commanding, apologizing, etc.)

perlocutionary acts — the desired result of saying something (forgive me, do it, etc.)

indirect acts — not really questions
(can you pass the salt? will you open the jar?)

Dimensions of difficulty in speech processing

Form of speech	single word	...	continuous	...	conversational
Attitude	cooperative				hostile
Population	one		several		many
Vocabulary size	10		10^3		10^4
Environmental noise	40 db				90 db

Channel characteristics	high quality	radio
Syntactic constraint	simple task grammar	dictation
Semantic constraint	small world model	real world

Levels of Understanding and Utterance

signal	
segment or event	(waveform is always ambiguous for the levels which follow)
phoneme	(transitions between phonemes are difficult to determine)
demi-syllable	
syllable	
word	
phrase	
utterence (sentence)	
discourse	

Experiences of HEARSAY

Large project at CMU in the 1980s with major funding. The hierarchical model

- signal parameters (waveforms)
- normalization
- feature extraction
- lexical constraints
- syntactic constraints
- semantic constraints

did not work, since errors at each level compounded. So the model had to include feedback in a graph network. Evolved into the blackboard model: each level of analysis posted its partial results to a global dataspace, and demons representing each level triggered on posting, returning their own partial analyses. The HEARSAY processing model:

- knowledge sources, each independent and autonomous
- blackboard for communication across knowledge sources
- knowledge sources insert own hypotheses and tests asynchronously
- each knowledge source responds to blackboard postings regarding its own interests
- alternative or competing hypotheses were integrated on the blackboard and could be stated in multiple representational forms (for the different knowledge sources)

AI Exotics

Conventional AI Topics in 1999

knowledge representation and articulation
commonsense and uncertain reasoning, belief structures
inference and deduction, expert systems
search (heuristics, strategies, competitive)
blackboard systems
learning and adaptation
deliberation, planning, and acting
situated modeling
speech and language processing
image understanding and synthesis
manipulation and locomotion (robotics)
autonomous agents and robots
multiagent systems
cognitive modeling
mathematical foundations

Probable Directions of Research over the next few years

representing the environment as messy, changing, and hostile
huge knowledge bases, data mining
interaction with human collaborators
system self-understanding
self-motivated learning and agents
more agents and multiagent systems (the next big thing)

Exotic Topics in AI

neural networks (1985 old, has own technical societies)
multiagent systems (becoming mainstream)
autonomous agents, artificial life
data mining, particularly in dynamic web environmnets
fuzzy logics (1990 old)
constraint reasoning (1980 old)
chaotic systems, cellular automata (1980 old)
genetic algorithms (1990 old)
machine evolution (1996)
reactive machines (1990 old)
robot perception (1980 old but little progress)
diagrammatic systems (1990)
web intelligence

Neural Networks

Densely connected networks with input, middle, and output layers
Inherently parallel architecture
Good for pattern-recognition after training
Sub-symbolic (no map between representation and meaning)
Knowledge (and programming) is in the strength of connections between network units
Each middle unit "averages" inputs, eg: outK@t = Sum(weightJ * inJ@t) + biasK
Weighted summation is a discrete model of differential calculus
Logistic summation: out@t = 1/(1 + e^(sum(weightJ*inJ)))
    simulates perceptual threshhold model
Training is accomplished by modifying weights to meet a performance objective
Programming is reflexive,
    ie the system reprograms its own weights through error feedback
Open issues:
    How to learn to solve a problem without training in the exact problem?
    Are there other useful architectures (compared to three layer approach)?
    Can training be less extensive fro hard problems; does training scale up?
    How can a network generalize to a class of problem when trained in a subset?
    How can you know the state of the system's knowledge? (validation issue)
Experience cautions:
    The training examples must sufficiently constrain the problem.
    Selection of appropriate input data is a design problem.
    Known symmetries must be included in the training set.
    The task needs a probabilistic model.
    Don't train with binary vectors (errors are fatal)
    Course coding (information blurring) is good for pre-training

Cellular Automata

"discrete dynamical systems whose behavior is completely specified in terms of a local relation, much as is the case for a large class of continuous dynamical systems defined by partial differential equations" Tommaso & Toffoli

Simple example is domino runs.
Highly parallel recursive technique
Complex behavior (or is it?) from simple rules
Primary example of self-generating infinite systems
Most simple automata are Turing equivalent
Outcome is not "predictable", since running the automata is the only computational model
Main result: Complexity occurs at phase transitions between simple and random

Agents

Software agents are programs with "attitude", or disposition
Agents act autonomously in support of a user's requirement
Programming techniques for agents include

- reactivity (selective activity depending on context)
- autonomony (goal directed and self-initializing activity)
- collaboration (attunement to work with other agents)
- knowledge-based (use rules for communication protocols)
- inferential (deduce consequences from inputs)
- temporal (persistence of state and identity)
- personality (believable interface with emotional simulations)
- adaptivity (self-changing over time)
- mobility (migrate to different environments, usually in search)

Purposes of agent approaches

- simplify distributed computing
  - automated interoperability
  - resource management
- improve user interface
  - simulate more human activity
  - relieve burden of direct interface
  - indirect management (vague specification)
- system architecture

Genetic and Evolutionary Algorithms

Genetic algorithms are designed using "natural selection" and random permutation.
Code fragments or bit-strings are modified by

- selection (those meeting a criteria are strengthened)
- crossover (two fragments swap pieces at a cut point)
- mutation (parts of a fragment are changed, usually randomly)

Millions of cycles are required for algorithm growth or evolution
Art is a good application.

Artificial Life

Self-replicating programs (viruses)
True autonomy ("Sorry, Dave, I can't let you do that.")
Graphics for Hollywood
Animal construction kits
Cellular autonoma, chaos, and fractal mathematics
Issues:

- physical grounding hypothesis (representation is not needed)
- What is autonomy? What is emergent behavior?
- What is the relationship between behavior and environment?