1. The Cyc Philosophy

1.1. Thin Ice

In July of 1962, some Lincoln Labs scientists, urged on by Marvin Minsky, decide to play a rather interesting practical joke on the Computing Center. They write a Begging Program, a program whose job control card (remember batch?) says to give it two minutes of CPU time. All the program does is wait 90 seconds, then print a message on the operator's teletype, "I am almost finished; I need 60 more seconds, please." Then it waits 45 seconds and prints "Almost done; one more minute, please." And so on. After twenty minutes, the operator kills the job. Totally deadpan, the programmer complains when he gets the printout showing he was charged for twenty minutes:

**Programmer:** The JCL card says 2 minute time limit!

**Operator:** Yeah, but the program kept asking for more time.

**Programmer:** So what? It always does that!

In early 1971, a patient talks to the ELIZA psychiatric counseling program.

**Patient:** My dog's mother died last week.

**Eliza:** Tell me more about your mother.

The patient is impressed with the subtlety and perception of the program at digging into his subconscious. Of course, what the program did was to simply ignore most of the words because it did not understand them. It really just understood him to say "My . . . mother . . .", and it printed out its standard response for that case.

In July of 1976, a medical diagnosis program is told about a follow-up visit by a patient on whose case it had worked the previous
week. The program asks for the patient's blood pressure and temperature again. It then asks for the patient's birthdate and gender this week.

In February of 1980, a skin disease diagnosis program is told about a "patient" that is a 1969 Chevrolet.

Program: Are there spots on the body?
User: Yes.
Program: What color spots?
User: Reddish-brown.
Program: Are there more spots on the trunk than elsewhere?
User: No.
Program: The patient has measles.

In June of 1983, a machine learning program is given hundreds of concept definitions, from half a dozen different fields, and begins to look for interesting new discoveries. It finds a few, but, in each case, begins to thrash after a while. It's almost as if there were some quantity of discovery-energy stored up in the program; this energy is quickly used up, after which the program seems to wind down.

In October of 1987, an expert system authorizes a car loan to someone who stated, on his application, that he'd worked at the same job for twenty years. A good risk? Perhaps, but the individual also stated he was 18 years old.

In March of 1989, a natural language researcher is worrying about how to get her program to figure out correctly which kind of "pen" is meant in these two sentences:

The pig is in the pen.
The ink is in the pen.

She realizes that, for the examples her system is supposed to handle, she can just distinguish the two cases on the basis of containing solid versus liquid objects. This is only a fair-weather solution, however (that is, after it rains there may be puddles in the pen, and ink clots in the other kind of pen).

In January of 1993, a minor earthquake shatters a glass-lined drum of virulent material at Oak Ridge National Labs (ORNL). The program that guides spill-handling procedures there asks numerous questions and recommends dumping a particular chemical on the spill. It asks for the name of the spilled material, so the user assumes it knows what the stuff is. Unfortunately, the program was written in the 1980s, before there were any biologically active materials stored at ORNL. The program does use the compound's name, of course, in some way: it appears here and there on its printed report the next morning.

In June of 1995, at 4:20 a.m., a warning bell sounds at a nuclear power plant just outside Paris. A tired technician, about to go off-shift, is confronted with a program's recommendation to close valve #802 immediately. People often go along with a fallacious argument, nodding continuously, if the argument has some plausible justification for each step. The technician does just this, as he reads the accompanying paragraph-sized justification that is displayed with the program's recommendation. He then goes off to close the valve.

Some of the preceding examples are true; some are apocryphal. The point is that computer programs are being given ever more responsibility, ever more complex tasks, and ever more sophistication. But their apparent intelligence and sophistication still vastly exceed the true depth of their understanding and the breadth of their capabilities. They are especially susceptible to failure when confronted with novel or unexpected situations. They are, if not complete idiot-savants, at least extremely brittle.

1.2. Overcoming Brittleness

Why aren't human beings brittle in the way computer programs are? How do we cope with novelty? Largely by finding some related case and propagating the differences to this new one. That is, we do one of the following:

- Match some similar situation and adjust it slightly (= remember)
- Match some apparently far-flung situation (= analogize)
- Fall back on general knowledge (= use common sense)
- Try to learn more about this new situation (= recur)

That fourth case is perhaps a special case of the third. In any event, it is a recursion, in which our "problem" changes from "X" to "learn
more about X," and any of the above four methods can now be tried
on that new problem. (Yes, any of these four methods! You can spend
time solving X, or learning more about X, or learning more about how
to better learn about problems like X, or . . . .)

The first three cases above (and hence, ultimately, the fourth) de-
pend on having a large base of both general and specific knowledge to
consult. As Marvin Minsky said in his afterword to Vinge's True
Names, "the more we know, the more we can learn." Unfortunately,
the flip side of that is: "If you don't know much to begin with, you
can't learn very much very quickly." That flip side comes into play
every time we build and run a program that doesn't know too much
to begin with, especially for tasks like semantic disambiguation of sen-
tences, or open-ended learning by analogy.

Expert systems finesse this need for knowledge; they restrict their
tasks so much that they can perform relatively narrow symbol manipu-
lations that nevertheless are interpreted meaningfully (and, we admit,
usefully) by human users. But having just a thin veneer of competence
and understanding is the cause of their brittleness; it's why they can
make mistakes that no human being ever could, often without even
knowing that they're out of their range of competence. It's also why
they can't readily communicate and cooperate with one another.

So the mattress in the road to AI is lack of knowledge, and the anti-
mattress is knowledge. But how much does a program need to know
to begin with? The annoying, inelegant, but apparently true answer is:
a non-trivial fraction of consensus reality — the millions of things that
we all know and that we assume everyone else knows. If I listen to
the stock market on a roller-coaster, and you don't know what I mean, I
might like it to a seashore, or to a stormy sea. If you still don't know
what I mean, I probably won't want to deal with you any more.

The necessary knowledge includes not just static facts, but also heu-
ristics and other problem-solving methods. Moreover, selecting a base
of knowledge for AI purposes involves making some hard choices
about which categories, individuals, relations, representations, etc., to
include. The Cyc group at MCC is attempting to build a single intelli-
genent agent whose knowledge base contains these tens of millions of
entries. We believe such a system will be a useful, perhaps necessary,
platform on which to undertake the next generation of work in expert
systems, natural language understanding, and machine learning.

Earlier, we said that brittleness can be overcome largely by drawing
on specialized knowledge, by falling back on increasingly general
knowledge, or by analogizing to specific but superficially disparate
knowledge. The first case is pretty clear — it's the basis for expert sys-
tems, for example. Let's look at an example of each of the other two
cases — general knowledge and analogy — to see more precisely how
and why we think they should work in Cyc.

1.3. Falling Back on General Knowledge

Here's a typical expert system (ES) rule, from the task of deciding
whether a credit card company should authorize a purchase or not:

If the purchase-price is greater than the remaining-balance
and you query "Are there unusual circumstances this month?"
and the purchaser responds "Yes"
THEN authorize the purchase

Brittleness arises because the program doesn't really understand the
meaning of the various terms, such as unusual circumstances,
purchase-price, remaining-balance, authorize, purchase, or even Yes and
query.

So in an expert system, much of the meaning of the terms is in the
eye of the beholder. This is a sort of "free lunch" when it works; it
gives the illusion of depth of understanding. A human being, and like-
wise an expert system, can push those terms around without thinking
deply about them most of the time. But then one is confronted with
something a little bit nonstandard, and there the divergence between
the expert and the expert system becomes apparent. The human ex-
pert can easily think of many reasons why the above rule might have
led to an incorrect conclusion.

For instance, let's just focus on the term query in the above rule.
Figure 1-1 is a fragment of the tree of knowledge above

Asking&Answering (the name for the process of one person querying
another and getting a response back from them). Please don't study
the diagram in detail at this time, and please don't study the constraint
syntax or predicates; just notice that there are some links from
Asking&Answering up to more general concepts, and that there are
some constraints listed on those general concepts.

We can derive failure modes for Asking&Answering by negating the
various constraints that are listed there at that node, and at each
ancestor of Asking&Answering. Even in this trivial example, we find a
dozen plausible ways that the rule might give the wrong answer, such as:

The question was so wordy that the listener forgot the first part by
the time the last part was spoken.

The communications link went down during the querying or reply-
ing action.

The listener could not clearly make out the words they were being
asked.
Falling Back on General Knowledge

- The listener misunderstood some terms (e.g., unusual circumstances) in the question.
- The listener was not aware of being asked a question.
- The questioner was asking the wrong person.
- The questioner did not want to tell the questioner the truth.
- The listener was unable physically to utter a response.

Notice that the same bits of knowledge that can help find ways a credit card expert system rule might fail can also find ways in which the following medical diagnosis rule might fail:

IF the intake-interview is in progress, and the doctor asks the-patient "Do you suffer from x" and the-patient responds yes
THEN assert x as a patient-symptom

There the failure modes include the same sorts of things listed above for the credit card rule, plus the possibility that the questioner isn’t really a doctor after all; or even the possibility that there is no second person at all (maybe the first person’s problem is that he is hallucinating; or that the patient’s symptom changed (either went away or began manifesting itself) right after the now-incorrect answer was given; etc.

It is often difficult to make a convincing case for having a consensus reality knowledge base (KB), because whenever one cites a particular piece of common sense that would be needed in a situation, it’s easy to dismiss it and say “well, we would have put that into our expert system as just one more (premise on a) rule.” For instance, in diagnosing a sick twenty-year-old coal miner, the program is told that he has been working in coal mines for 22 years (the typical accidentally hit two 2s instead of just one). Common sense tells us to question the idea of someone working in a coal mine since age —2. Yes, if this sort of error had been foreseen, the expert system could of course question it also. The argument is, however, that we could keep coming up with instance after instance where some additional piece of common sense knowledge would be needed in order to avoid falling into an inhumanly silly mistake.

For example, here are a few more. How about a diagnosis program that is told about a pregnant woman whose age is 103 and weight is 40; obviously someone switched those two pieces of data. Or how about a grandmother whose age is entered as 9. Or twins whose ages are different. Those examples just touch on a few age-related mistakes that common sense could catch, and age in turn is just one of the thou-

Figure 1-1: A Fragment of the Tree of Knowledge above Asking&Answering
The Cyc Philosophy

sands of attributes that Cyc already knows about. (A rough estimate puts the number of such attributes at 15,000 by 1994.)

Most of the time, as we operate in the real world, we draw on compiled experiences — on very abstract, simplified “black-box” models. When we drive to work in the morning, we can’t and don’t rethink the route we take, the running condition of the car’s engine, etc. But if some unexpected problem develops — the route is suddenly bumper to bumper, or the car begins stalling out — we’re able to open up the black box and reason at the next more detailed level, and the next, and so on, until we get to the right level at which the problem can be handled. So we’re definitely not arguing for first-principle reasoning all the time; for cognitive economy, intelligent programs and people must “compile away” the general knowledge and assumptions to produce efficient, special-case rules and procedures. The danger comes in programs not having that basic knowledge available as a back-up. Such is the source of brittleness.

1.4. Analogizing to Far-flung Knowledge

The previous subsection illustrated how to cope with novelty by falling back on increasingly general knowledge. What of falling back on analogy? You may think that analogy is just an isolated curiosity, or a literary device that has dramatic power but no real heuristic power. See for yourself. Here is a partial analogy between treating a disease and waging a war:

*Treating a Bacterial Infection*  
*Fighting a War*

**enemyType**: Disease  
**enemyType**: MilitaryForce

**enemyLocal**: Bacteria  
**enemyLocal**: EnemyTroops

**protagonistType**: Physician  
**protagonistType**: Soldier

**enemyProcess**: Infecting  
**enemyProcess**: Invading

**protagProcess**: ClinTreating  
**protagProcess**: MilRepulsing

**usefulPreprocess**: Diagnosing  
**usefulPreprocess**: J2ing

**usefulTactics**: Vaccination  
**usefulTactics**: MilContainment

**locale**: BodyPart  
**locale**: GeographicRegion

**emotionalCharge**: Low  
**emotionalCharge**: High

...Often, analogy is used as little more than a dramatic device. For example, one news reporter might make use of the difference in emotionalCharge to dramatize the plight of a dying child, by describing “his brave battle, his war against the infection that has invaded his body.” Conversely, a government news agency might use that difference to downplay the horror of a firefight, by describing it in clinical terms like sterilization action.

Although this usage is common, it is (as regards Cyc) the least interesting and powerful use of analogy. Analogy has two other uses: to suggest brand new concepts and to help in fleshing out their details. Let’s look at how the above analogy may be used in these two ways.

Notice that some of the concepts on one side are analogues of those on the other side. But Vaccination and MilContainment aren’t analogues. In fact, they each have no analogue (yet) on the other side. Such asymmetry is bound to happen, and this is an opportunity; let the analogy guide the search for new, useful concepts on each side.

For instance, maybe we should define a medical analogue of military containment, and a military analogue of vaccination. The former might be medical containment — for example, the use of a tourniquet on a venomous snakebite, or the use of quarantine on a virulent plague. The military analogue of vaccination might be fortifying or propagandizing. To take the analogy even more seriously, the military vaccination might entail letting a small group of enemy soldiers overrun some territory before our friendly forces secure it, as a way of driving home to the local populace just how bad the enemy is.

The example above illustrated the utility of analogy as a guide for defining new concepts. Analogy can also help flesh out the new concepts. For instance, what precisely is medical containment containing? What does it locally contain? How should one do it? To answer those questions, go to the military containment concept, look up the answers there, and map them back using the existing analogy.

**MilContainment**

**usefulTacticIn**: Fighting-a-war  
**containedType**: MilitaryForce

**containedLocal**: EnemyTroops  
**attributeslimited**: Mobility

**howTo**: Bound&Isolate  
**counterTactic**: (Threaten containedArea)

**containedArea**: GeographicRegion

That suggests that medical containment, in the case of treating a bacterial infection, is containing a disease, and, more locally, bacteria. It might be done by surrounding and isolating the infected part of the body.

We could carry through a similar procedure to guess at the values of various slots on the military analogue of vaccination.

The example above, leading to military vaccination and to medical containment, illustrates that analogy can be useful in suggesting new concepts and in fleshing them out. That shows that analogy does have quite significant potential heuristic power. Moreover, as Lakoff and Johnson argue quite convincingly in *Metaphors We Live By*, analogy is
far more pervasive than one might at first imagine, appearing in almost every sentence we utter.

Our sentences and our thoughts are riddled with analogies (such as "riddled with"), which in turn are built on analogies, and so on, like the skins of an onion. At the core of the onion is a small tangle of very primitive somatic metaphors such as forward, backward, up, down, hungry, tired, pain, push, see, breathe, sleep, and eat.

It's easy to imagine how this massive dependence upon analogy and metaphor might have come about: We understand (and communicate) "which," not "what"; that is, we perceive (or transmit) something that's already well known plus a small set of changes. Learning and understanding and speaking occur at, and can modify, only the fringe of what we already know. So even if we began as neonates with largely non-analogical knowledge, as we live our lives new skins of the onion are continually added on.

1.4.1. Why Analogy Works Frequently. The previous subsection argued that analogy is frequently useful and discussed specifically how it might be useful. But why is it frequently useful? The answer to that lies in the nature of the world we happen to inhabit and the state of our understanding (and perhaps our capacity to understand). Three aspects of our world, and ourselves, make analogy frequently useful to us as human beings:

1. The moderate amount of novelty with which we're confronted
2. The moderate number of distinct causes in the world
3. The mediocre ontology and "knowledge metric" we have

NOVELTY. If the world were very volatile and chaotic and wildly unpredictable, analogizing would avail us little; it would almost always fail. If it were totally static and static and unchanging, we'd have little need for analogy; memory would suffice.

CAUSALITY. Analogies that have no common root cause are superficial and weak and, more often than not, no good for more than literary divestissement. If there were a zillion disparate causes in the world, analogy would usually be superficial and powerless in this fashion. On the other hand, if there were no variety of causal mechanisms in the world (for example, only one emotion, only one method of physical propulsion, only one kind of chemical reaction, etc.), there wouldn't be much power in classifying which of those causes were behind something. (In fact, there wouldn't be much point in having those terms, either.)

1.4.2. How To Analogize. Okay, so reasoning by analogy can be great, is frequently used, and is perhaps even indispensable. Now, how do we get a program to do it? Uemov [1970] dully notes:

When analogy is successful it is called 'deep,' 'strict,' 'scientific' and so on. When it fails, it is called 'superficial,' 'loose,' 'unsatisfactory' and so on. Naturally, there is a question as to how these two types of analogy can be distinguished before the practical realization takes place.

The simplest model for analogy is that of structurally mapping slots of one frame to slots of another frame. This must be generalized to include mapping between one network of frames and another network of frames; it must also include knowledge-guided reformulation to reveal commonalities between imperfectly matching entries, and likewise between imperfectly matching slots.

EXAMPLE 1. Consider the statement "Fred is urinous." Presumably that means to map TypicalBear (and the cluster of units associated with TypicalBear) to Fred (and the cluster of frames associated with Fred).

We might want to map the TypicalBear's qualitative size (Large, compared to the typical TypicalWoodlandCreature) to Fred's qualitative size (Large, compared to the TypicalHumanMale.) That was easy, because both the slot (qualitativeSize) and the value (Large) were the same. Frequently, though, either the two slots or the values are

KNOWLEDGE METRIC. If we had a terribly wrong view of the world, analogy would lead us even further into error (for example, thinking that the cosmic objects and meteorological phenomena are sentient, one might try to bribe them with offerings.) If we had a terrible grasp of the world, we'd always know precisely what knowledge was relevant to our present dilemma, and exactly how it should be applied. We wouldn't need analogical leaps into the possibly relevant. In many situations, we have some knowledge as to what aspects may be relevant to the decision-making problem, but we don't know how to "compute" the correct decision from those aspects. Analogy is useful because it allows us to find other situations to borrow from, situations in which we know some clearly (in)correct decision.

In other words, analogy is a useful heuristic method to employ because we and the world happen to fall near the midpoint of those three axes. (It is largely a matter of individual preference and belief whether one considers this coincidence to be an accident, or part of a conscious Design, or nothing more than a reflection of the current limits of human cognitive abilities.) A skewing along any one of these axes would reduce the power of analogizing.
different. For example, one might map the Bear's claw length and claw sharpness to various attributes of Fred's fingernails, which would be a cross-slot mapping. We might map some absolute numeric values (such as 800 pounds) to different numbers, and so on.

How can a program automatically find these, and other good mappings? That is, how does it notice to even try to do this, and how does it manage to carry it out?

To answer these important questions, let's ask: why might a person make that analogy and utter the statement "Fred is unlike Mowgli," if you think about it for a minute (as we did), the surprising answer is that this is not a powerful analogy after all, it's just a nice, powerful, and compact way of communicating (in parallel) a few facts that just happen to be true.

There is no causal connection here, just coincidence. All the speaker was doing was (a) compacting his message, and (b) injecting a little humor and hyperbole and variety into his speech. So it's not hard to imagine that a program could notice that a good match exists between Fred and TypicalBear, and that TypicalBear is well known to the typical listener, and, if the current problem is to describe Fred to that listener, then the program could decide to refer to Fred as unlike Mowgli in order to communicate a lot about Fred all at once.

A large fraction of human use of "analogy and metaphor" is not analogy or metaphor at all, then, but rather falls into this category of merely compacting a message by finding superficially similar "analogues." Let's turn to an example of a "real" analogy.

**Example 2:** "Mowgli is unlike Fred." This sure seems similar to example 1, at least on the surface. What's different about it? Here, in contrast to example 1, there is a causal connection underlying the analogy, because Mowgli (the character from Rudyard Kipling's *Jungle Book*) was raised by wolves. One's upbringing is a strong determinant of one's attitudes toward food, shelter, possessions, ethics, life, death, music, physical conditioning, and so on. If we ask you whether Fred (from example 1) likes to catch fish, or what method he uses to catch them, or whether he cooks them before he eats them, etc., it's unlikely that the answers from TypicalBear will prove to be reliable guesses. But if we ask you whether Mowgli likes to catch fish, or what method he uses to catch them, or whether he cooks them before he eats them, etc., then you would expect to be able to guess the answers by looking at TypicalWolf. To answer Lemov's skepticism, we remark that there's not too much bias in this example, because we (the authors) don't know much about Mowgli or about wolves. We have no idea whether wolves eat fish, or how they catch them, etc. But we'd be surprised to

learn that there's any episode in *The Jungle Book* that shows ways in which Mowgli and the wolves differ on diet, hunting methods, table manners, or other upbringing-dependent attributes.

There are several ways by which a program might first suspect this analogy:

1. By noticing that Mowgli was raised by Wolves, and that raised by strongly determines many other important properties. Following Russell [1988], we say that "Properties x and y determine z" to mean that most objects sharing the same value for their x and y properties will likely have the same z values as one another. For example, age and neighborhood determine language spoken. Two individuals of the same age, who reside in the same neighborhood, are likely to speak the same language.

2. By noticing that there are some unusual (uncommon) similarities between various attributes of Wolf-003 (some particular wolf) and Mowgli --- such as their table manners, howling frequency, goals, dreads, and gait; and then either
   a. Trying to "explain" these by finding some deeper, independent attribute(s) that determined them, and/or
   b. Trying to extend the analogy to other (not necessarily "deep") attributes to probe its boundary, simply because extending it is a cost-effective way of getting more information into the KB.

3. By some external clue, hint, or reference. "External" means "explicitly provided" by a human user, the surrounding text in the novel, or some very far-flung source.

4. By random search for analogies. This is a special case of all three of the preceding methods, and is probably too inefficient to make it worth trying to automate, at least on conventional serial computers. Perhaps the human brain does some of this random searching "in the background," unconsciously, and with a high factor of parallelism. If so, and if it is indeed a relatively rich source of useful analogies, then some specialized piece of parallel hardware might be warranted.

5. By noticing that "our problem, X, is very reminiscent of problem Y, which we've already worked on and know a lot about; and analogies of type Z helped us in a lot with Y, so maybe analogies similar to Z will help with X." This may be viewed as a variant of method 2b above, where X and Y are noticed as similar and then the useful Analogies slot of Y is mapped back to X.
In order to get these various schemes to work (excepting number 3, deus ex machina), the system must have access to a large knowledge base, a rich space in which to prospect for matches. Most of the five schemes also require deep understanding of the knowledge in the KB, to permit obscure or imperfect (but still promising) matches to be made, to sort out the superficial from the significant matches, to posit and judge the plausibility of a common causal explanation for the similarities, and to decide along which other axes (slots) to search for further similarities that would extend the analogy.

We’ve glossed over some hard issues — perhaps the hard issues — in successful analogical reasoning, namely how exactly to pick a promising analogy, how to develop it, to which as-yet unmatched slots the match should be (attempted to be) extended (for example, useful tactics), and to which slots it should not be extended (for example, inventories). Other critical issues are how to tell if the phenomenon going on is really “exploiting a hitherto unrecognized common generalization,” how to tell if the analogy is superficial and not worth extending anymore, when to rely on it, and so on.

Our basic approach to answering all these questions is “divide and conquer.” That is, we posit that analogy is a vague English word covering a multitude of types of inference. The types of analogical reasoning can be usefuly arranged in a space whose dimensions include things such as the nature of the boundary (where the analogy breaks down), the nature of the analogues (for example, the hardness of each analogue’s field), the nature of the attributes being matched, the purpose of the analogizer, and so on. Each cell in that n-dimensional matrix represents a type of analogical reasoning, and each of those types deserves to be teased out separately and studied. A bundle of heuristics can then be assembled for each cell; through course many of the heuristics would apply to whole sections of the matrix.

We hope to tame analogy through this two-pronged attack:

1. Breaking down the phenomenon into its various subtypes and then handling each one

2. Having a realistically large pool of (millions of) objects, substances, events, sets, ideas, relationships, etc., to which to analogize.

As we remarked a few paragraphs back, successful analogizing depends on the second prong above, so most of our efforts to date on Cyc have focused on building up that large KB, not on working out the details of the various kinds of analogical reasoning.

The Representation Trap

1.5. The Representation Trap

The careful reader may have noted a possible hole in our discussion of the brittleness of current expert systems: First, we criticized them for merely containing opaque tokens and pushing them around. Yet our example of “having more general, flexible knowledge” was nothing more than having more (and more general) tokens and pushing them around.

Yes, all we’re doing is pushing tokens around, but that’s all that cognition is. What makes our tokens “better” is that they aren’t tied to solving some particular problem. Naturally, all programs are built on some primitives (predicates, frames, slots, rules, functions, scripts). But if you choose task-specific primitives, you’ll win in the short run (building a program for that narrow domain) but lose in the long run (you’ll find yourself painted into a corner when you try to scale the program up.)

Given a specific task, the necessary knowledge can be represented in a number of ways; in particular, it is often possible to cheat by using long predicate names. For example, we might use the predicate WellTrained (or the attribute wellTrained) to indicate that some creature was well trained. We know that WellTrained horses have been well trained, and that WellTrained implies Trained. But both facts would have to be told explicitly to the system; it doesn’t speak English. So we could have named that predicate P00089, or Untrained, and its understanding would have been no different (for example, its behavior when facing a problem involving horse training would have been no different).

Here is a typical example of how to solve a problem dishonestly, in this case that of deciding whether something can live in a desert. Consider a program that contains the following four if/then rules:

(IF LivesNearWater x) THEN (NOT LivesInDesert x))
(IF LaysEggsInWater x) THEN (LivesNearWater x))
(IF Amphibian x) THEN (LaysEggsInWater x))
(IF Frog x) THEN (Amphibian x))

Now suppose we assert: (Frog Fred). We could use the above rules to conclude that Fred does not live in the desert. Note that the knowledge isn’t adequate to let the system conclude such things as:

Fred sometimes gets wet.
Fred lays eggs.
Fred is alive.
Fred doesn’t live in the Sahara.
The problem is that we are using very detailed, complex predicates (such as \textit{LaysEggInWater}) without defining them. (Incidentally, the conclusion being derived — that Fred does not live in the desert — is correct, but the chain of reasoning is invalid! Fred, a male frog, does not lay eggs.)

Why is it a bad idea to use complex predicates without defining them? Informally, it’s disturbing because for us to “understand” something means that we can answer questions about it, can relate it to most things to which it ought to be . . . or, related. If you hear that Fred \textit{LaysEggInWater} then you ought to be able to answer questions like “Does Fred lay eggs?” and “Is Fred sometimes in the water?”

Formally, it’s a bad idea because it’s explosive, as follows. (We’ll make a very simplistic argument.) Suppose we have a thousand primitive properties of objects that we can perceive; and let a complex property be composed of three primitives. Then we have a billion (1000!) complex properties. If we represent everything in terms of complex properties rather than primitives, then we may require up to a billion billion rules to represent their interrelationships — that is, 10^{12} rules. If instead we defined each complex property in terms of its primitive components, we would have a billion definitions and a million rules relating primitives. This is the same argument in favor of planning islands in a search, where again an exponential search has its exponent cut by a factor of 2. If we define a series of levels of decreasingly primitive relations, we can chip away more and more at this exponent.

Basically, what we’re saying is that we need to relate things by virtue of the relations between their constituents. This is what happens in the world too, it’s just that human beings are so good at abstracting to just the right level of abstraction that we aren’t conscious of the mental “work” we’re doing.

For instance, when you rent a car, you figure out how to drive it by dealing with the various parts of the car: the door lock, the door handle, the seat adjuster, the headlights, the wipers, the turn indicators, etc. You have and use a few rules for each of those types of car parts, rules that help you quickly locate them in a new car and operate them. You don’t have a thousand scripts like “how to drive a Prelude,” “how to drive a Prelude,” etc. (Of course, if we do happen to get into a type of car we are familiar with, we can draw on our already-cached script for that kind of car.)

With only one narrow class of problem to solve, such as getting ready to drive a Honda Prelude, and only a (relatively) small set of facts to represent, however, we can always get away with cheating. In the driving-a-Prelude case, we could program the specific arm motions that would turn on the car’s headlights.

In such cases the representations developed reflect more the structure of the problem than the structure of the world. The narrower (and smaller) the task, the worse this problem, this trap, becomes.

In knowledge representation, therefore, narrow domains are misleading. And small KBs likely won’t scale up easily into huge KBs. This, then, is the representation trap, the trap that has snared (or even characterized) expert systems to date: choosing a set of long, complex primitives (predicate names) that have a lot of knowledge compiled within them, and writing rules that are also tailored to the program’s domain (omitting premises that needn’t be worried about in that particular task). The bait in the trap is the fact that it works — at least within the narrow domain for which that particular program was designed. The catch is that the resultant system is isolated and brittle.

1.6. Ontological versus Knowledge Engineering or Why Building a Huge KB is Different from Building n Small KBs

Cyc needs a deep, well-organized ontology of real-world knowledge. This is in contrast to, say, the latest special-purpose Expert System.

1.6.1. Why Expert Systems Don’t Display Synergy

What’s special about having just a narrow range of problems to solve? It allows you to cut corners in two ways. The first way is pretty obvious — you need to encode only a sliver of the world’s knowledge.

The second way to cut corners is much less obvious: having a narrow task allows you to represent — and reason about — a simpler world, not just a smaller one. (See also “The Representation Trap,” Section 1.5.) The point is, even if your problem touches on some “tough” knowledge — belief, substances, events, time, space, structure and function, etc. — you can usually get by with capturing just a few selected aspects of them. Let’s give a few examples of what we mean:

For instance, when you rent a car, you figure out how to drive it by dealing with the various parts of the car: the door lock, the door handle, the seat adjuster, the headlights, the wipers, the turn indicators, etc. You have and use a few rules for each of those types of car parts, rules that help you quickly locate them in a new car and operate them. You don’t have a thousand scripts like “how to drive a Prelude,” “how to drive a Prelude,” etc. (Of course, if we do happen to get into a type of car we are familiar with, we can draw on our already-cached script for that kind of car.)

With only one narrow class of problem to solve, such as getting ready to drive a Honda Prelude, and only a (relatively) small set of facts to represent, however, we can always get away with cheating. In the driving-a-Prelude case, we could program the specific arm motions that would turn on the car’s headlights.

In such cases the representations developed reflect more the structure of the problem than the structure of the world. The narrower (and smaller) the task, the worse this problem, this trap, becomes.

In knowledge representation, therefore, narrow domains are misleading. And small KBs likely won’t scale up easily into huge KBs. This, then, is the representation trap, the trap that has snared (or even characterized) expert systems to date: choosing a set of long, complex primitives (predicate names) that have a lot of knowledge compiled within them, and writing rules that are also tailored to the program’s domain (omitting premises that needn’t be worried about in that particular task). The bait in the trap is the fact that it works — at least within the narrow domain for which that particular program was designed. The catch is that the resultant system is isolated and brittle.

1.6. Ontological versus Knowledge Engineering or Why Building a Huge KB is Different from Building n Small KBs

Cyc needs a deep, well-organized ontology of real-world knowledge. This is in contrast to, say, the latest special-purpose Expert System.

1.6.1. Why Expert Systems Don’t Display Synergy

What’s special about having just a narrow range of problems to solve? It allows you to cut corners in two ways. The first way is pretty obvious — you need to encode only a sliver of the world’s knowledge.

The second way to cut corners is much less obvious: having a narrow task allows you to represent — and reason about — a simpler world, not just a smaller one. (See also “The Representation Trap,” Section 1.5.) The point is, even if your problem touches on some “tough” knowledge — belief, substances, events, time, space, structure and function, etc. — you can usually get by with capturing just a few selected aspects of them. Let’s give a few examples of what we mean:

For instance, when you rent a car, you figure out how to drive it by dealing with the various parts of the car: the door lock, the door handle, the seat adjuster, the headlights, the wipers, the turn indicators, etc. You have and use a few rules for each of those types of car parts, rules that help you quickly locate them in a new car and operate them. You don’t have a thousand scripts like “how to drive a Prelude,” “how to drive a Prelude,” etc. (Of course, if we do happen to get into a type of car we are familiar with, we can draw on our already-cached script for that kind of car.)

With only one narrow class of problem to solve, such as getting ready to drive a Honda Prelude, and only a (relatively) small set of facts to represent, however, we can always get away with cheating. In the driving-a-Prelude case, we could program the specific arm motions that would turn on the car’s headlights.

In such cases the representations developed reflect more the structure of the problem than the structure of the world. The narrower (and smaller) the task, the worse this problem, this trap, becomes.

In knowledge representation, therefore, narrow domains are misleading. And small KBs likely won’t scale up easily into huge KBs. This, then, is the representation trap, the trap that has snared (or even characterized) expert systems to date: choosing a set of long, complex primitives (predicate names) that have a lot of knowledge compiled within them, and writing rules that are also tailored to the program’s domain (omitting premises that needn’t be worried about in that particular task). The bait in the trap is the fact that it works — at least within the narrow domain for which that particular program was designed. The catch is that the resultant system is isolated and brittle.
tied to whatever intervals during which it is relevant or true (for example, HearSayII, STRADS).

Decreasingly simple views of structure: (a) You might be able to ignore structure entirely, and just use a "feature-vector" representation. (For example, MYCIN can decide which kind of meningitis a person has, without knowing that people have two arms and two legs.) (b) MYCIN does presume that there is a single spinal cord and brain, but this is woven into the program at the level of global variables, not explicitly, declaratively "understood" by the program. (c) There may be a few specific joints, or compound assemblies, whose internal structure matters in doing your chosen task. If so, you can model each such structure in your program idiosyncratically, after learning what level of detail you'll need. (d) Your program might contain a whole vocabulary of task-related types of combinors or joints, plus a "grammar" of their legal uses, that allows more dynamic analysis of structures and synthesis of new ones.

Now we're getting to the source of incompatibility between different expert systems. Even if the two ES's were written in the same language (tool, shell), and even if their domains seem related (for example, medical diagnosis), still there's almost no chance of being able to usefully dump rules from one ES into the other. There are two types of almost-avoidable incompatibility:

**INCOMPATIBILITY 1.** Each rule, each piece of knowledge, has implicit in it the various assumptions and simplifications that were present implicitly in its ES. If you take a rule from a system that presumes "just one instant of time," it's going to have predicates that check whether the patient "has" symptoms x, y, and z. Even though we understand that this means "did the patient have them all at the same time," that's not stated in the rule — it's implicit in the way the ES was designed. Now dump that poor rule into an ES in which time is handled differently, and it might fire at inappropriate times.

**INCOMPATIBILITY 2.** What's even more likely to happen is that the rule will never fire, because the various ways that ES#1 carried up the world — the predicates used to state the rules — are not quite the same as the ways that ES#2 carried up the world. One system's rules might talk about the presence of the attribute Feverishness, the other might talk about the magnitude of the parameter Body Temperature. One might have Headache value categorized as Mild or Severe, the other might use three terms like Slight, Moderate, Extreme. The first ES would have other rules that would conclude "Severe" about some condition, and other rules that might be triggered by a "Severe" condi-

The Cyclic Philosophy

**Ontological versus Knowledge Engineering**

1.6.2. The Path to Fertile Co-mingling. So either the rules collide with each other, or they pass in the night without interacting at all. No wonder ES's aren't "inter-combinable"! What can we do to ensure a more fertile co-mingling?

To answer that critical question, let's think about how we are able to look at either of those ES's and understand what it's saying. How do we understand that Low/Medium/High corresponds to a dividing up of the scale of responses, as does Mild/Severe, or Slight/Moderate/Extreme? We have knowledge — the "right" knowledge — of how to interrelate them: namely, we can visualize the linear scale that is their common generalization. The ES's don't have that knowledge, but they don't individually "need" it, either, except when faced with some unusual, unexpected circumstances.

We now can see more clearly what each ES's task is. It isn't to do medical diagnosis, or design chips, or whatever, it's to help humans do those things, to input and output symbols that are largely meaningless to it but that will mean something to the people who are using the system.

At some level, people also choose to respond mechanically — we don't always drop down one level in detail, but the key difference is that, generally speaking, we could if the situation warranted it. When we are vague, when we omit details, it is generally a choice we've made; it is voluntary simplifying from a position of strength, a situation wherein we could be precise if we need to.

What would it mean for a program to operate from such a "position of strength" rather than to operate by the illusion of competence? It would mean a program that had an over-arching framework for the world's knowledge and that had its detailed knowledge — predicates, frames, slots, rules, or whatever — related to that global ontology. In addition, the program would need the general knowledge that connects together the specific areas' expertise and allows them to combine synergistically.

The "Mild" or "Severe" classes are a low-to-high partitioning of the continuum of Strengths. So is "Low, Medium, High," so is "Slight, Moderate, Extreme," and so on. An intelligent agent, be it human or machine, would know this common generalization, and would be familiar with these three special cases.

The "Feverishness" parameter would have links that relate it to — indeed, define it in terms of — an atypically high value for the
Body/Temperature parameter. That in turn would be related to Temperature in general, so that, for example, either Fahrenheit or Centigrade scales could be used, and (if any detailed piece of knowledge is needed) body temperature, skin temperature, and environmental temperature could be interrelated.

This makes it sound like we're advocating a set of translation tricks, a set of useful transformations. Well, in a way that's right. But it's not so much "a set of ..." as "an adequate, global set of ..." What do we mean by that? We mean that everything that humans can easily conceptualize ought to be easily, naturally tied in to this over-arching framework for knowledge. The sort of translations we listed above (converting among temperatures, or from fever to temperature, or from one way of breaking up a Low-to-High scale to another way) ought to be typical applications of the framework. How many such typical things would there have to be, then, in this "top layer"? The bad news is that we believe the answer is millions. But the good news is that it should be only a few million (at least if you count only "things" like frames that each have a lot of content to them).

Given that expert systems can't co-mingle today and will require something like Cyc to serve as the "semantic glue" among them before they can effectively cooperate, how are we to construct Cyc? Sections 1.7.3 and 1.7.4 present a similar chicken-and-egg situation with respect to machine learning (ML) and with respect to natural language understanding (NLU). Efforts in these three areas (ES, NLU, ML) have fissured the problems that we must solve in order to get Cyc built, so we can't just use any of them as a model and "scale them up."

1.7. Building Cyc

The previous subsections have argued for building a huge KB; that is the effort under way as Cyc. There are two parts to our task:

- Do the top layers of the global ontology correctly
- Relate all the rest of human knowledge to those top layers

This is where we diverge from philosophers and, frankly, previous AI work (yes, even our own). Instead of talking more about this, and making forays here and there as needed to gather a few examples, we set out to actually do it. Well, almost. We figured that if we could get pretty far on setting up the top layers, everyone would pitch in and help us relate the rest of human knowledge to them.

Is it possible? Who knows? But let's get started and see! That was our attitude in 1984; we gave Cyc a 10-20 percent chance of succeeding. Now it looks like our original guess about the size of the task was about right, and we'd give us a full 50-60 percent chance of succeeding.

The first task involves the order of ten million entries. (As Marvin Minsky observed, that's about the same order of magnitude as the number of things a human being acquires — burns into long-term memory — during ages 0 to 8, assuming one new entry every ten seconds of one's waking life.) The second task is unbounded, but probably another ten to fifty million entries would suffice for general intelligence (for example, the intelligence required for acquiring knowledge in school and in extra-curricular conversations). Quite a bit more may be needed for qualitatively super-human intelligence.

The most easily foreseen mode of failure was — and is — that the knowledge enterers might diverge, either by stepping on another's toes (misusing each other's terms) or by passing one another in the night (re-entering already-existing concepts, giving those units slightly different names).

If the latter duplication is ever noticed, then it may be fairly easy to fix (either by relating the units to one another, or, in extreme cases, just merging them), so the failure mode would lie in never realizing that this duplication occurred in the KB.

One interesting tool that helps in identifying such duplication is to have Cyc actively search for new analogies. Some of them are genuine, interesting analogies; some of them are mappings between non-analogous concepts A and B, which signifies that we haven't yet told Cyc enough about A and B to differentiate them properly; and a third class of apparent analogies are between concepts that are really just different formulations of the same knowledge — that is, passings in the night.

Since 1984, we've been building and organizing and reorganizing our growing consensus reality KB in Cyc. We now have about a million entries in it, and we expect it to increase by a factor of 4 by mid-1990. Thanks to an array of explicit and implicit methods for stating and enforcing semantics, they appear to be converging, not diverging. The following sections discuss what it means for semantics to converge; they also cover various tactics that might have been used for building the Cyc KB. The chapter ends with a brief sketch of the method we've settled on and the anatomy of the current Cyc system.

1.7.1. Convergence. Naturally, we must build up the Cyc KB from some sort of primitives. We have claimed that it must be built from deeply understood
knowledge rather than from complex “impenetrable” predicates (or slots or whatever).

At first, doing this just made life difficult; having a deep but small KB didn’t pay off. Fortunately, as we built Cyc ever larger and larger, we found that the set of primitives began to converge. That is, it requires less and less work to enter each new fact. This phenomenon is not surprising (it was, for example, predicted in Pat Hayes’ Naive Physics Manifesto); it is merely very important. And it was quite comforting to see it really happen.

Let’s illustrate how convergence can occur. Consider a legal reasoning system that must advise someone whether to sue someone else. Say your car has been scratched, and there are plenty of witnesses. With little common sense, the system might fire a rule like

\text{IF your property has been damaged by X, and there is little doubt as to the facts of the case, and monetary recompense is not forthcoming, THEN sue X.}

But if your car is scratched by a bag lady, litigation may be a bad idea. A clause like “... and the perpetrator is not a bag lady” might be added to solve this problem. For more generality, the new conjunct might be phrased “... and X is not destitute.”

The Cyc approach would be different. We would describe what the process of suing is (a way of getting money from a defendant) and what money is, and we would give very general rules about the process of transferring some X from Y to Z, including the precondition that some X must exist at Y in order to be transferred from there, the fact that there is some overhead cost to running a transfer process, and perhaps some special knowledge about who does what to whom and who pays what to whom during and after the suing activity that would take place in America today.

From this, Cyc could generate the appropriate behavior in this case. But more importantly, the system would now be able to exhibit robust behavior in an unimaginably large number of other cases also. For instance, it could reason that if someone drops a few coins into a beggar’s cup, then they had (at least) those coins on their person just prior to the charitable act, it could reason that the world’s consumption of resources will eventually have to cease, and it could reason that one usually doesn’t borrow money from beggars.

To solve the specific “car-scratching” problem, it’s tempting to put in special case knowledge. But as you widen the definition of the problem domain (for example, “acting intelligently in interpersonal situations involving money or property”), it becomes more economical to

1.7.2. Tactics for Ontological Engineering. So we must build a good global ontology of human knowledge (that is, one that spans current human consensus reality) if we are to avoid the representation trap. Choosing a set of representation primitives (predicates, objects, functions) has been called ontological engineering—that is, defining the categories and relationships of the domain. (This is empirical, experimental engineering, as contrasted with ontological theorizing, which philosophers have done for millennia.)

More than just having a good ontology, however, we must also build up a large knowledge base organized according to that ontology, a KB of millions of (frame-sized) pieces of consensus reality knowledge. How many millions? We hope and expect it’s about 5 million frames, each with several dozen “fact-sized” slot entries; but we’ll find out. This project is mankind’s first foray into large-scale ontological engineering.

Well, what about encyclopedias and thesauruses? Encyclopedia writers have been able to finesse 90 percent of the ontology issue because of three factors:

1. An encyclopedia is largely a linearly ordered sequence of articles, so the main decision to make is “grain size,” not organization.
2. People learn early in life what sorts of topics will and won’t have articles dedicated to them.
3. To the extent that 1 and 2 are insufficient, a peppering of cross references will help a person jump from an almost-correct place to the correct place.

Thesaurus writers have been able to finesse 99 percent of the ontology issue. Go take a look at the table of contents of a thesaurus, and you’ll see an interesting phenomenon: it’s terrible! For instance, the leading thesaurus was developed a couple centuries ago; one of the top-level divisions of knowledge is Theology, Physics is a sub-sub-part of Chemistry, and so on. And yet, the thesaurus works fine. Its job is not to be a good global ontology, but rather to be good locally, to clump together words with similar meanings. Thesaurus have to be only locally good.

So both Cyc and its ontology must be built almost from scratch. The question, then, is: How should such a gargantuan knowledge base be
constructed? What methodology, what tactics, will suffice? We’ll examine a few possible short-cut methods, and then describe the method we finally chose.

1.7.3 Free Lunch Try 1: Natural Language Understanding. It would be super to get a program to speak English, after which we could sit back and watch as it “went to school.” That works well for people, but there’s an awful lot that kids know long before they enter school, before they can even begin to speak coherently. It’s not clear how that knowledge gets into their heads and is organized in the 0-2 year time period; some of it probably has a head start (that is, is “wired in”) as a result of the way we’ve evolved over the eons. But the bottom line is this: You can’t seriously expect to do natural language understanding until you already know most of consensus reality! Recalling the following pair of sentences:

The ink is in the pen.
The pig is in the pen.

The first sentence’s “pen” is a writing implement, the second is a pigsty or corral. How do you know that? It’s not English that tells you, it’s your knowledge of how big things are, what it takes to serve as a container for liquid, what sorts of things ink goes in, what sorts of things pigs go in — and why. That is, you disambiguate the word “pen” by drawing on your knowledge of the real world. Here is another example:

The little girl saw the bicycle in the window. She wanted it.
The little girl saw the bicycle in the window. She pressed her nose up against it.

What does the word “it” refer to in each of those sentence pairs, the bicycle or the window? How do you know that? You know because of your knowledge of the real world, of human emotions and capabilities and anatomy, of physical and mental and physiological limitations, and so on — not because of English grammar and syntax. If the girl collected glass, the first sentence’s “it” might refer to the window; if she were fabulously wealthy, “it” might even refer to the whole store! But those would be one-in-a-million situations (or jokes); all the rest of the time, “it” would refer to the bicycle. In the second sentence, even more real-world knowledge is required to fully understand why she would press her nose against the store window; for example, to see more details (which in turn presupposes a good understanding of the relative locations of nose and eyes), or to get as physically close to the bike as possible (which in turn presupposes an understanding of the socioeconomic implications of actually entering a store as opposed to merely window-shopping).

Fred likes ice cream, candy, etc.

What does “etc.” mean in the sentence above? Looking at the common properties in the two given items to induce from, we infer the general category “sweets.” So ellipses (“etc.” “and so on,” “. . .”) can be understood only if the listener already knows enough of the properties of the items to be able to search for the commonality.

Dogs are on my front lawn.
Dogs are mammals.
Dogs are widely distributed on Earth.

Clearly, “Dogs are X” can mean many different things. The first sentence means “Each member of some specific set of dogs is on my front lawn.” The second sentence means “Each member of the set of all dogs is also a member of the set of all mammals.” The third sentence means “The area distributed over the set of all dogs is a large fraction of the land area of the Earth.”

The Columbia University School of Journalism collects humorous examples of ambiguous phrases from real newspaper headlines. While these are admittedly extreme, the average reader has no trouble understanding them on, say, the second or third reading:

British Left Waffles on Falklands
Sharon to Press His Suit in Israel
Mere Silversware Stolen — Police Seek Pattern

One day (hopefully in this century), natural language understanding will be the most important mode of knowledge acquisition into the growing Cyc KB. In that vision, Cyc reads texts, novels, journals, newspapers, and so on, and holds discussions with human beings to help clarify the difficult parts (or the unintelligible parts such as a transliterated accent or “dialect” woven into text) and to help check its assimilations.

One day, we hope Cyc will read those humorous headlines and understand them (and obviously understand their obvious misinterpretations). Before that day arrives, however, a lot of material must remain in the KB.

So, to summarize Free Lunch Try 1, we believe that Cyc and NLU will mature synergistically, each requiring the other. But NLU alone is not a short-cut to building Cyc, at least not in its first half-decade of existence.
1.7.4. Free Lunch Try 2: Machine Learning. It would be super to get a program to discover the needed knowledge on its own, just by its observing the world, noticing regularities, doing simple experiments to verify them, and so on. Lenat spent his youth (1973–1983) pursuing this dream. The AM and Eurisko programs were surprisingly successful, and are partially responsible for catalyzing the renaissance of learning as a subfield of AI. But... But they rely on a trick: having a representation so well chosen that syntax mirrors semantics. In such a situation, one can go far just by doing syntactic mutation and exploration. [See Lenat & Brown, 1984.]

The better understood the domain is, the more likely it is that such a representation can be found. Unfortunately, this is a lot like burning coal: after a while, the energy is released, the fire dies, and you have to go out and manually dig up some more fuel. We kept hoping that we'd ignite a positively reinforcing cycle, but we never even came close.

Machine-learning researchers are working on ways to actually generate a bootstrapping process, by using existing and newly acquired knowledge to guide and improve the learning process, but an effective system is a long way off.

In the final analysis, we succumbed to “the more you know, the more and faster you can learn.” More precisely, the inverse of this statement bit us:

If you don’t know very much to begin with,

Then you can’t learn much right away, and what you do learn

you probably won’t learn very quickly.

What was holding those programs back was the lack of a large knowledge base to use as a substrate for the learning process. And given current peripherals, it would be hard for Cyc to go out on its own and acquire a sufficient experiential base from which to induce the required common sense.

1.7.5. Try 3: Hard Work (No Free Lunch). The limited success we had with automatic program synthesis from examples and dialogues [Green et al., 1974] in the early seventies led us to the AM research (automated discovery of domain concepts). Its limited success led us to Eurisko, which tried to discover new heuristics as it went along. Its limited success led us to believe that there is no free lunch; that is, that we had to apply the tactics of last resort — hard work — and thus Cyc was born. We are building the needed KB manually, one piece at a time, at least up to the crossover point where natural language understanding begins to be a more effective way of further enlarging it.

This task looked — and still looks — just barely possible. We estimated that it would take about two person-centuries to build up that KB, assuming that we don’t get stuck too badly on representation thorns along the way. In real time, our schedule was — and still is — to complete the project (reach that crossover point) in the ten-year period that ends in late 1994.

Naturally, there were — and are — a lot of hard problems (or, depending on how you look at them, interesting issues). Some of them are: how we decide what knowledge to encode and how we encode it; how to handle the “thorns” (how to represent substances, parts, time, space, belief, and counterfactuals); how Cyc can access, compute, inherit, deduce, or guess answers; and how we’re going to squeeze two person-centuries into a decade of real time, without having the knowledge enters’ semantics fatally “diverge.”

Back in section 1.5, “The Representation Trap,” we spotlighted the need to choose a good set of primitives. That will be one of our “keys” to achieving convergence: defining knowledge in each area in terms of knowledge in other (often more general) areas. For example, when baseball is explained to Cyc, it is in terms of generic actions like running, hitting, catching, competing, cooperating, taking turns, and so on. This is also the source of power behind modularity in programs, primitives in programming languages, and grammatical structure rather than monolithic messages in natural languages.

The other “keys” to semantic convergence are (a) to have a sufficiently large KB that one can tell it something new by plucking a similar piece of knowledge and making some small editing changes to it, (b) to have a good enough global ontology to make that easy, and (c) to have Cyc function as an intelligent agent whose first important task is to help with its own continuing enlargement (including policing the KB, building and using user-models to accelerate their actions, making guesses that semi-automate the copy-and-edit process, and so on).

A few years ago, shortly after we began, we published our initial plans [Lenat, Shepherd, et al., 1986]. Our schedule was to have enough of the KB built to transition to natural language understanding as the dominant knowledge entry mode in 1994. By now, we’ve gotten pretty far along. Not surprisingly, there have been unexpected problems and unexpected discoveries. Perhaps the biggest surprise is that we’re still on schedule. The thorns we had to deal with — time, change, the overall ontology, and so on — have been faced up to and trimmed, rather than avoided. In more detail, …….
2. Overview of Cyc

Cyc comprises three “pieces”:

1. The knowledge base itself (Cyc KB)
2. The environment: the interface editing/browsing tools (UE and MUE), the multi-user knowledge server, the binary KB dumper, and so on
3. The representation language (Cycl)

The boundaries between these pieces are not as sharp as they are in most systems. For example, much of Cyc is represented in the KB, and some of it is only represented that way; so the border between the KB and Cyc is hazy. Similarly, much of the user interface is represented in (and only in) the KB, blurring the border between the KB and the environment.

The sections in this chapter answer a few basic questions about each of the three pieces of Cyc.

2.1. The KB Itself

The Cyc knowledge base is to be the repository of the bulk of the factual and heuristic knowledge, much of it usually left unstated, that comprises “consensus reality”: the things we assume everybody already knows.

What will Cyc’s KB Contain? As a broad “driving force” or “forcing function,” we chose to encode the knowledge required to understand a one-volume desk encyclopedia and a newspaper — including editorials, advertisements, advice columns, and so on. This does not mean the contents of such works (though, frankly, we often add some of that as well); rather, it means ferreting out and representing the underlying common sense knowledge that the writers of those articles assumed their readers already possessed.

Another good “forcing function” is to examine articles that we disbelieve and introspect on why we find them incredible. (For example, one article tells of an Indian guru who sat underwater for seven days and seven nights. One of its great internal details: “Skeptics claim he came up at night for air.”)

Introspection, although a good tool, is not the only one used. Knowledge editors pose questions about a piece of text just “explained,” questions that “anyone” should be able to answer having just read that text, and Cyc is further augmented until it, too, can answer those questions. The newly added knowledge is generalized, both to make it more broadly applicable and to help test — and push us into extending — Cyc’s ontology.

What is the Time Scale for the Construction of Cyc? During 1984–1989, a small team of AI researchers has been encoding mutually dissimilar articles and pieces of knowledge. Much of their time has been spent figuring out adequate work-arounds for various representation “thorns,” such as time, space, belief, causality, substances, intelligent action, mind versus body, and so on.

Beginning in 1989, a growing cadre of knowledge editors has begun to use machine-assisted copy/edit procedures to encode the final 99 percent of the Basic KB. At the same time, Elaine Rich, Jim Barnett, and other natural language understanding researchers have been building a system that interacts synergistically with Cyc, calling on Cyc to do semantic disambiguation, anaphoric reference resolution, ellipsis interpretation, etc.; and being used by Cyc as a front-end knowledge acquisition mechanism.

Anecdotal successes have been achieved in each “direction” during 1989, and the hope is that by late 1994, this combined system will make obsolete the current manual method of knowledge entry. Gradually, the role of the humans will shift to resemble that of teachers or tutors: recommending what to read next and explaining difficult passages. That phase (Cyc que voracious reader) will never quite end, but after some number of years — perhaps around the turn of the century — Cyc should be as well fleshed out as the “person on the street,” and as flexible. That is, Cyc will be increasingly “usable” during the 1990s.

How will Cyc’s KB Be Used? As performance programs get “stuck,” they can fall back on more and more general knowledge, and/or rely on analogy, to cope with their predicament. Those same techniques should also speed up knowledge entry.
As described in chapter 1, we believe that this KB will enable the next generation of AI work and achievements in expert systems, machine learning, and, of course, natural language understanding.

HOW IS Cyc’s KB ORGANIZED? This is the issue of the “global ontology of human knowledge.” We are not claiming that we have the correct organization, just a more or less adequate one — or maybe even less: one that is correctable and extendable into an adequate ontology. Subsequent sections will have a lot to say about the organizing principles that underly the knowledge base; what are the basic categories, why have categories at all, etc.

2.2. The Interface

WHAT TOOLS ENABLE A KNOWLEDGE EDITOR TO RAPIDLY BROWSE AND EDIT THE KB? The tools we’ve built so far include a “symbolic spreadsheet” frame editor (UE), whose commands are structural and textual, and a spatial “museum room” editor (MUE) that maps frames metaphorically into floorplans of rooms. See figures 2-1 and 2-2; also see chapter 8, which discusses these editors in more detail.

Beginning back in 1984, we also developed a node/link semantic net graphing and editing/browsing interface, but this soon became too cluttered and tangled to be useful. To avoid the “spaghetti” phenomenon, we tried having just local (instead of global) placement of nodes, but that lost the valuable kinesthetic memory, the “feel,” of the KB (for example, remembering that emotions are just over to the left of hobbies).

ARE THEY TOOLS FOR THE NOVICE OR THE EXPERT? Unlike most “human interface tools,” these tools have been built to help already-fluent knowledge editors — not naive users. They are powerful tools to navigate around knowledge space and to resculpt the knowledge. They are intelligence amplifiers for the most talented knowledge editors, not mental prostheses to hold the hands of neophyte editors.

Developing these tools has led us to consider and experiment with various exotic I/O technologies: speech synthesis, speech understanding, on-line visual dictionaries to point to, sound-space navigation, animation, color, pedals, 3D helmets, etc. Very little of that technology-dependent material will be covered in this book, because of its ephemeral and experimental nature and its general unavailability (or, at least, nonstandardization) outside our laboratory.
WHAT IS THE RELATIONSHIP BETWEEN THESE EDITING TOOLS AND THE OTHER PARTS OF Cyc? The UE and MUE tools are front ends, interfacing between the human user and the CycL Representation Language (described below). Each UE/MUE operation gets converted into a CycL command (or, occasionally, a sequence of CycL commands) that is then run and that changes the Cyc KB.

Although CycL is written in CommonLisp, UE and MUE cannot be because there are no CommonLisp standards for windows, debugging, network protocols, etc. The 1988 versions contained much code that was idiosyncratic to ZetaLisp, to SCL (Symbolics Common LISP), to the Symbolics Lisp machine, and even to the then-current Symbolics software release (Release 7.2). During 1989, we decided it was worth developing a set of equivalent customized interfaces for other machines (such as Suns, TI Explorers, DEC 31005, and Mac IIs).

HOW CAN MORE THAN ONE PERSON WORK ON THE SYSTEM AT A TIME? Several dozen users (knowledge editors) currently work simultaneously on the Cyc KB. Under the present scheme, each user has his own full copy of (each piece of) Cyc, and each is connected by a "thin wire" to a central machine called the Cyc Knowledge Server (KS). Each user runs UE and MUE, viewing and modifying his own local copy of the KB.

Each editing change performed by a knowledge editor is checked locally, to see if any constraints are violated (see section 2.3.2); if no errors are found, the change is broadcast to the KS. There it is checked again, and if, once more, it causes no errors, it is broadcast to the other users. Each of these high-level editing operations will generate a number of primitive changes on each machine (anywhere from one primitive operation to millions, but usually about a hundred).

If an error is detected, a dialogue is initiated to resolve it. In the case of a constraint violation, for example, the user might be asked if this is an exception, or if the constraint needs to be weakened or modified. In the case of one user directly undoing something that another has recently done, Cyc mediates a "discussion" by online menus to resolve the problem. (If one party has already logged off, a provisional decision is reached, but the discussion still occurs, just over a longer time period, by electronic mail messages.)

DOESN'T EACH USER HAVE LONG Waits, THEN, AS HIS (AND OTHER USERS') OPERATIONS TAKE PLACE? Originally the answer was Yes, so we had to take steps to correct the problem. The solution we chose is to have a separate foreground process (talking to the user) and background process (doing the ripping, talking to the Knowledge Server,
Overview of Cyc

and attending to other users' operations). That way, the user can continue, usually editing as quickly as desired, letting the background catch up asynchronously. This is not error-proof, of course, but in practice it almost never leads to collisions.

On a larger scale, any operation that might take a long time to propagate (for example, several minutes) is maintained as backward-only until an off-time (such as the middle of the night), at which point Cyc takes the time to forward-propagate it through the KB.

Several types of accretive operations are distributed among those machines that are currently idle (given a distribution of users and machines such as our present one, there are often as many idle machines as there are those in active use):

- Checking for policy conflicts and cases of two knowledge editors “stepping on each other’s toes.” These conflicts are typically more subtle than one KE’s directly undoing a recent action of another.
- Detecting inconsistencies or unlikelyhoods in the KB.
- Looking for analogies, which may turn out to be genuine, valuable new analogies; false analogies that point out omissions in the KB (X and Y are not analogous, Cyc just wasn’t told enough about them to know that); or trivial analogies that point out missing links (or, in extreme cases, full duplication of effort) in the KB.

2.3. The CycL Representation Language — Introduction

Before discussing the representation issues involved in building Cyc’s KB, let us take a brief look at some of the details of the language in which we will be representing the world.

Think of this section as a summary of what can be expressed in CycL and what types of inference are done for you by CycL. It should give you enough background to understand the later discussions and examples.

2.3.1. CycL Is Frame-Based. Superficially, CycL is a frame-based language: that is, it’s based around triples like “the capital of Texas is Austin.” All the assertions about Texas are gathered together into one data structure called the Texas frame or the Texas unit:

Texas
name: (Austin)
stateOf: (UnitedStatesOfAmerica)

The unit representing Texas is depicted here as having three slots, each of which has a corresponding value. The value of a slot of a unit is always a list of individual entries. Even a slot like capital, which should have only one entry, is assumed to have a singleton set as its value.

(An aside for purists: The semantics of the list (v1 v2 . . .) appearing as the value of the s slot of unit u is: {u,v1} ∪ {u,v2} ∪ . . . . Thus, the order of entries, and the number of times they occur, do not convey any meaning. So to be more precise about it, a slot’s value is just a mathematical set, not a bag, list, set, poset, or multiset.)
2.3.2. On Top of the Frames is a Predicate-calculus-like Constraint Language. Why did we say that Cyc is only superficially a frame language? Because the expressive power of frames by themselves is insufficient to represent concisely all that we would like to say. For instance:

"Bill is either a terrific fisherman or a terrific liar."
"Siblings almost never have the same first names."

In order to overcome this deficiency, another language, the Cyc constraint language, sits on top of the basic frame language and provides the requisite expressiveness.

The constraint language is essentially predicate calculus. We can have expressions like "for all slots s and s', if s is transitive and the inverse of s is s', then s' is transitive." Considering the slots as our "predicates," this means that our constraint language is at least second-order.

Pragmatically, expressions above first order are almost never actually used. Also, an explicit set over which any quantified variable could possibly range is almost always known. There are some optimizations that are performed in those cases; in case the statement is only first-order, in case it can be easily first-ordered, in case it has variables x that range over a given set X, in case it has some "constants" in it, and so on. For example, a few special constructs have been added, such as TheSetOf (e.g., (TheSetOf × PerfectNumber (isA × Odd-Number))). See the later subsection 2.3.6 for details.

Why bother having two languages — frames and predicate calculus constraints? Why not just use the more powerful one (predicate calculus)? Though the constraint language is more powerful, the frame language makes inference (deduction) much simpler and faster. It turns out that many of the things we need to say about the world can be said compactly in a simple, efficient frame language. So, although we could just adopt the constraint language, as it is more powerful, we would still want to add the cross-indexing that makes frames so efficient — and adequate — in most cases.

2.3.3. The Kinds of Frames That Exist in Cyc. Each Cyc unit represents something — a real-world object, a type of process, a particular event, an abstract idea. Often we'll just say Fred, in this document, to mean either the person Fred or the Cyc unit representing Fred. In potentially ambiguous cases, we'll prefix unit names by #%. So if we were being precise here, we'd say that #%Texas has three slots, but Texas has no slots (only frames can have slots — #%Texas is a frame in Cyc, but Texas is a state in the U.S.A.).

The CycL Representation Language — Introduction

Four basic kinds of frames exist in the system: "normal" ones, SlotUnits, SeeUnits, and SlotEntryDetails.

"NORMAL" UNITS

Ninety percent of all frames fall into this category; they are the frames that represent various real-world concepts and things such as Fred, the set of all Dogs, the process of Walking, the English word red, and so on.

SLOTUNITS

These are frames that represent types of slots. For example, the unit called #%residents is a full-fledged Cyc unit, and describes that type of slot. It contains information such as "only a geopolitical region can be said to have residents; each of those residents must be a person; if residents(x,y) then residentOf(y,x); all lifelong residents of a region are residents of that region." A SlotUnit represents the whole relationship — not just a particular instance of that relationship, like residents(Texas,Doug), or one small class of instances, like residents(Texas,(Douglas, Mary, Guha)).

Having a frame representing each type of slot is a big win; it lets us state facts about them, lets us interrelate them, and lets us define and constrain them. Consider:

residents

instanceOf: (Slot)
inverse: (residentOf)
makesSenseFor: (GeopoliticalRegion)
entryIn: (Person)
specSlots: (lifelongResidents illegalAliens
registeredVoters)
slotConstraints: ((coTemporal u v))

This says that #%residents is a kind of slot ("instanceOf" is like "member of" or "element of"); and x is on the #%residents slot of Y if and only if Y is on the #%residentOf slot of x; and only geopolitical regions should have a #%residents slot; all the entries on a #%residents slot had better represent individual people. The next slot — specSlots — indicates that if x is known to be a lifelong resident of Y (or an illegal alien living in Y, or registered to vote in Y), then x can be assumed to also be a resident of Y. The final slot — slotConstraints — says that if u is a resident of v, then they'd better both exist at the same time. That's how Cyc would know, for example, that Julius Caesar couldn't be a resident of New York.
These are units that serve as "footnotes," providing metalevel information about a particular slot of a particular unit. Consider the residents slot of the Texas unit above. We might want to say that this slot's value has about 10 million entries — though we certainly don't know exactly what they all are! Or we might want to say that the rate of change is low in the number of residents of Texas, even if we don't know how many residents it has. SeeUnits enable us to express these facts:

```
Texas
  capital: (Austin)
  residents: (Doug Guha Mary)
  stateOf: (UnitedStatesOfAmerica)

SeeUnitFor-residents-Texas
  instanceof: (SeeUnit)
  modifiesUnit: (Texas)
  modifiesSlot: (residents)
  rateOfChange: ()
  cardinality: (10000000)

SeeUnitFor-rateOfChange
  .SeeUnitFor-residents-Texas
  instanceof: (SeeUnit)
  modifiesUnit: (SeeUnitFor-residents-Texas)
  modifiesSlot: (rateOfChange)
  qualitativeValue: (Low)
```

The first "diamond" on the residents slot of the Texas unit points to the second unit. The second diamond, which is on the rateOfChange slot of that second unit, points to the third unit.

SeeUnits are full-fledged units in Cyc and hence can themselves have SeeUnits. In the above example, for instance, we don't know the true (absolute) rate of change of the residents slot of Texas, but we do know something about that value, namely that it is qualitatively Low.

These are similar to SeeUnits, but instead of "talking about" a whole slot of a unit, they modify a single entry on a slot of a unit. Suppose we want to say something about Guha being an entry on the residents slot of the Texas unit — for example, that this relationship became true during 1982 and has been a never-ending surprise both to him and to Lenat. To do that, we'd create a SlotEntryDetail unit:

```
SeeUnitFor-Guha-residents-Texas
  instanceof: (SlotEntryDetailTypeOfSeeUnit)
  modifiesUnit: (Texas)
  modifiesSlot: (residents)
  modifiesEntry: (Guha)
  becomeTrueIn: (1987)
  surprisingTo: (Guha Lenat)
  moreLikelyThan: (SeeUnitFor-PickupTrack ownsA Mary)
```

and that unit would be pointed to by the Guha entry in the residents slot of the Texas unit:

```
Texas
  capital: (Austin)
  residents: (Doug Guha Mary)
  stateOf: (UnitedStatesOfAmerica)
```

SeeUnits and SlotEntryDetails are similar to the "reified objects" suggested in [McCarthy 81].

### 2.3.4. The "Fields" Stored for Every Slot of Every Unit

(WARNING: You should probably just skim over this section on first reading, until you have looked over the rest of this chapter.)

In addition to just storing a value, CycL maintains half a dozen other fields as well, for every slot of every unit. These fields contain such information as:

- A truth value (TV) for each entry in the value
- A symbolic justification for why that entry is present in the value
- Bookkeeping information used by the truth maintenance facility (for example, other units/slot values that this value depends on and other units/slot values that depend on this value)
- Some of the properties that each entry v1 on the value inherits just by virtue of its being there
- Information pertaining to the attitudes of agents toward this proposition (beliefs, desires, etc.)

Notice that the TV field could in principle be handled by just having lots of SlotEntryDetails; and most of the other fields could in principle...
be handled by just having lots of SeeUnits. But by examining and codifying the most frequent — and highly optimizable — kinds of bookkeeping remarks we needed to make about a value or an entry, we were able to obviate the need for SeeUnits and SlotEntryDetails; at present, they are very rare in the KB.

2.3.5. Inference in CycL: What Does Cyc "Do"? We've already discussed frame/slot/entry triples, SeeUnits, and SlotEntryDetails.

Coupled with the details of our scheme for handling non-numeric certainty factors, alternate worlds, etc., these elements make up the "Statics" of Cyc. In this section, we'll discuss the "Kinematics" of Cyc: What sorts of inference does it actually do, as it runs?

Rather than have a single general-purpose inference scheme (such as, say, Resolution), Cyc has a number of special-purpose inference schemes (currently 24 of them), each of which is optimized for dealing with "inference rules" of a particular kind.

Some of these inference mechanisms include:

- Inheritance (simple propagation along any type of arc, not just A-KIND-OF)
- Automatic classification (recognizing something that satisfies a certain definition)
- Maintenance of "inverse links" (for example, keeping likes in sync with likedBy)
- Maintenance of "definitions" (for example, grandparents = parents = parents)
- Maintenance of "dependencies"
- Maintenance of "genlSlots," "refinements," and "inContextVersions" of a slot
- Firing of demons (for example, afterAdding methods on slots)
- Checking constraints (constraints can be absolute or just usually "true by default")
- Using general wffs (well-formed formulae)
- Agenda-based best-first search (metalevel guidance)
- Finding and using determinations and structural analogies
- Gathering and combining multiple constraints

- Guessing, based on making a nearly-closed-world assumption
- Guessing, based on "illegal" but metaphorically sensible leaps

A metalevel agenda is used to decide which inference scheme is to be used when. Each task on the agenda is a full-fledged Cyc unit that can be reasoned about in much the same way as can "object level" units.

The inference schemes in the language are divided into several levels, with each level depending on (and freely calling) the previous level.

Get0 Simply access the data structure; akin to GETPROP
Get4 Try some of the simpler, faster Cyc inference mechanisms: toCompute, genlSlots, demons, and inheritance that is temporarily backward (and eventually will be forward)
Get6 Try the above, plus the more costly Cyc inference mechanisms: slotValueSubsumes, classification, slotConstraints, structures, slotValueEquals, and full backward inheritance
Get8 Try some plausible guessing mechanisms, such as determinations, constraint-resolving, closed world assuming, analogy, metaphor, and metonymy

(CycL inference features such as inverses are never "called" by the Get... functions because there is a guarantee that these are always forward propagated. Hence, even Get6 should find the entries (if any) added by these inference mechanisms. The same goes for forward inheritance — its contributions are always cached, so even Get0 will find them.)

The metalevel information at levels 0-4 is almost completely calculated at the time of assertion of facts, and simply cached. That makes retrieval at those levels fast.

Almost all the reasoning done by the system can be considered a kind of defeasible reasoning. As we shall see later on, almost any "assertion" can be overridden by, or combined with, any more certain information.

CycL has a truth maintenance system (TMS); there are five types of truth values attached to — and symbolic justifications for and against — each entry on each slot of each unit. This system is not a general TMS: each optimized inference scheme is largely responsible for ensuring that the inferences drawn by it are still valid. So, for example, when a conflict arises, very customized control can be exerted to resolve it.
2.3.6. **Salient Aspects of the Constraint Language.** The constraint language supplements the basic workhorse — the language of frames and slots. It is a real boon, enabling us to easily state:

- Disjunctions (Jim’s father is either Sam or Bill or a Chinese fisherman)
- Quantified statements (Some of Fred’s older siblings got higher grades)
- Relationships among slot values (People are younger than their parents)
- Negations (Fred isn’t a Dane; Brothers rarely have the same first names)

We had the choice of not allowing such statements to be made, forcing them to be done by making a “statement” unit for each of these, or just biting the bullet and adopting an additional language in which it’s easy to make such statements. Given the need for expressiveness, the choice was quite obvious.

This is not to say that the frame language is completely useless. As it turns out, a vast majority of the statements we would like to make about the world can be stated in the form of binary predicates with both the arguments being constants; hence, these statements can be directly represented as simple units/slot/entry triples. As stated earlier, this is a feature of the real world (that is, of the two aspects of the world we’re usually interested in: tangibles and intelligences). So we may as well make use of this in order to make our language more efficient. Hence, the constraint language is simply an addition to the basic frame language.

The constraint language has a syntax similar to that of predicate calculus. Just as each kind of slot has a unit representing it, so each kind of predicate has a unit representing it. Any slot may be viewed as — and, in our constraint language, actually used as — a predicate. For example, we could say (residents Texas Doug).

There are modified versions of the universal and existential quantifiers; in our system, they’re predicates called ForAll and ThereExists. The modification is that one must specify the set over which quantification is done (either explicitly or by a functional form). This is at least as good as standard V and E, because this set can be specified to be the universal set, and in most real-world cases, we will know a limiting set for the variable to possibly range over, so it is computationally much better to be able to supply that set.

For example, consider the following expression:

```cyk
(#ThereExists x (#AllInstances (#First x))
(#ThereExists y (#AllInstances (#PresidentOf x)))))
```

This might be a worse search, if it weren’t clear that we were looking for a world leader. In other words, presidentOf and firstName are both legal slots for all people, so Cyc might (in the worst case) otherwise have had to look through all the units representing people in order to find out whether such a person existed. (Because presidentOf can be filled with any organization, not just a country, a person can be president of some group and not be a world leader.) Instead, Cyc can now look through the relatively tiny set of known world leaders. See [Green et al., 1974]. So the condition of an extra argument to the two quantifiers is only to improve the efficiency and has no effect on the semantics of the quantifiers.

Propositions in the constraint language can have a whole unit representing them. Associated with these propositions are a number of Lisp functions (generated automatically by the system), each of which is optimized for a particular task (such as checking the truth of the proposition at some point or translating it into the frame language for some set of bindings). So most of the time Cyc can ignore the proposition and use only these functions. This is largely for reasons of efficiency; it recoups some of the efficiency that would otherwise be lost by having such an expressive representation.

Expressions in the constraint language are used for different purposes in Cyc. For instance:

- Stating constraints on slot values (to signal errors and violations)
- Specifying which units can legally possess a certain type of slot
- Stating the definition of a collection (for automatic classification)
- Stating the premises and conclusions for different kinds of “rules”

Constraints on slot values may be “soft” — i.e., only defaults. That’s useful in the real world, where many things, though possible, should cause Cyc to raise its metaphorical eyebrows: a person over 110 years old, an armadillo with a job, or twin sisters who have the same first name.

The constraint mechanism is tightly coupled with the inheritance mechanism: constraints get inherited along various arcs (allInstances, parts, allSpecs) and end up in the inheritedSlotConstraints slot of many “lower” units, where they actually apply and are maintained.
Thus, the Inheritance part of CycL is being exploited by the Constraints part of CycL. Basically, inheritance provides a means of efficient indexing, so that it is always easy to figure out which rules and constraints affect a given unit.

We will discuss the Constraint Language in much more detail in chapter 3.