AI2 Natural Language Notes

D. Sannella and H. Thompson
Department of Artificial Intelligence
University of Edinburgh

1986/87

AI-2 Natural Language
Lecture Notes

D. Sannella and H. Thompson
Department of Artificial Intelligence
University of Edinburgh

Abstract

These notes introduce computational mechanisms for understanding natural language and answering questions. Computation models of increasing power (finite state machines/regular expressions, recursive transition networks/context-free grammars, augmented transition networks) are examined with the help of extended examples. First-order predicate calculus is used for the representation of meaning and theorem proving is used to support question answering. An analysis of the problems inherent in natural language processing, a discussion of several existing systems, and a brief introduction to an alternative notation for representing meaning (Schank's conceptual dependency) are also included.

Table of contents

Lecture I

Natural language processing -- significance and difficulties

1.1 Natural language as a medium for communication

Natural language: as opposed to *invented* languages like computer languages (e.g. Prolog, Pascal) and *formal* languages as used in logic and mathematics (e.g. predicate calculus).

Use of language for communication is what separates humans from other animals. Clearly, the ability to communicate via language is an important aspect of intelligence. The word "dumb" (stupid) originates from "dumb" (unable to speak). Even so, language is universal; everybody learns to speak their native language fluently somehow by the time they are 6 or so. Only very severely handicapped people don't learn to speak.

A.I. interest in natural language stems from:

- the practical interest of systems which can communicate; and
- the psychological interest of understanding mechanisms involved in human language.

1.2 The Turing test

The Turing test was proposed in 1950 by Alan Turing as a definition of artificial intelligence. If a computer is able to pass the test, it deserves to be called intelligent (he claimed). The test is heavily based on the use of language; a computer is intelligent if it can carry on a conversation in a way indistinguishable from a human.

Test: Human A sits at a terminal connected to terminals B and C. On B or C is a human and on the other a computer (A knows this). A must determine which is which by carrying on a (typed) conversation, with no restrictions on what is typed. If A is unable to distinguish the computer from the human, the computer is intelligent.

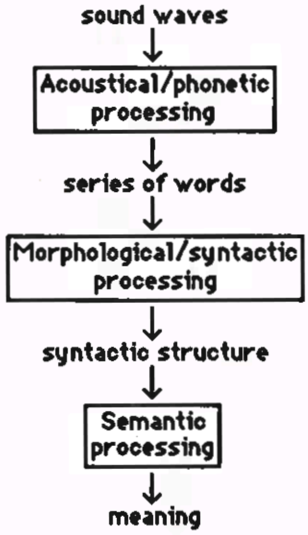
The dialogue could include (for example):

- general knowledge questions
  - Q: Where is the castle?
  - A: At the top of the Royal Mile.
  - Q: What is 243 times 429?
  - A: (pause) ... 104247, I think.
  - Q: What is the capital of Upper Volta?
  - A: How on earth shall I know?
- comprehension tests
  - Q: Once upon a time ...... Why did Bob poison Kevin?
  - A: He was jealous that Kevin was so good in AI2.
- emotional response tests
  - Q: What do you think of AI2?
  - A: It would be okay if only the lectures were a bit later in the day.
  - Q: You lazy twit! I guess you're just too stupid to understand all the profound thing we're learning.
  - A: Would you like me to tell you what you can do with your profound things?

Language ability alone isn't enough to pass the Turing test: the computer also needs a large amount of world knowledge, an ability to generate plausible emotional responses when appropriate, etc. But language ability is a necessary requirement.

1.3 What is so difficult about language understanding?

The understanding process can be split into three steps:



At each level there are big problems.

Common sense and introspection don't help much in figuring out how humans understand or produce language; psychological experiments show this. So for example although it seems like it is possible to start a sentence without knowing how it will end, which would indicate that at least production is not a three-step process, maybe you just don't know that you know.

Psychological experiments can test some theories of language understanding. For example, if a theory says that sentence A requires more work to understand than sentence B, it is possible to test whether it takes more time to answer questions about A or about B. But there is a limit to what can be tested in this way.

1.3.1 Acoustical/phonetic processing

A naive approach to the problem would probably be to separate continuous speak into words by looking for silences in between periods of sound. Long pauses are commas, very long pauses are ends of sentences. The sounds in a word can be analysed to decide what the word is: k a t = "cat".

But this approach unfortunately does not work. Most of what you think you hear people saying is not really there at all; the actual acoustic signal is noisy and ambiguous, and lots of things are missing. For example:

- *Word boundaries are not always silences.*
  Listen to speech and you'll realize how much words are slurred together.
- *Not all pauses are word boundaries.*
  Voiceless consonants (k, p, t) are just silences which can only be distinguished by their effects on nearby vowels.
- *Length of a pause doesn't fit well with sentence structure.*
- *Different accents have different sounds, e.g. for vowels.*
  You have to learn how to understand an accent you've never heard before by trying to figure out what the person might be saying (examples: deaf person's speech, Scottish).

There is no rigid boundary between this level and other levels; what you expect to hear influences what you hear.

1.3.2 Morphological/syntactic processing

Many words mean several things (*lexical* ambiguity):

He saw her duck.

Without contextual information we don't know if "duck" here is a noun or a verb, so we cannot parse this sentence.

Morphemes (like "s" at the end of a word to form the plural) are not dependable:
"bats" is the plural of "bat"
but
"bus" is *not* the plural of "bu".

Some dialects have words which are not in other dialects:
"grits", "bonnet", "messages"

How do we know who did what to whom in sentences like the following?
My aunt gave that teapot to me.
That teapot was given to me by my aunt.
My aunt gave me that teapot.
I was given that teapot by my aunt.

At the syntactic level, natural language is enormously complex compared with computer languages:
Here comes the dog that killed the cat that ate the rat that lives in the house that Jack built.
The rat that the cat that the dog killed ate was named Percy.

Structural ambiguity is a problem; consider the following (who has the stick?):
The boy hit the dog with the stick.
How does a program know which parse to take?

There is no strict boundary between the syntactic level and the semantic level:
I saw the Forth Road Bridge flying into Edinburgh.
This has two entirely different syntactic structures depending on who is flying, but common sense says which is the right one (based on meaning). How can a program be made to see this?

Two other examples:
I ate dinner with a friend.
I ate dinner with a fork.

Nobody has exactly the same dialect (ideolect).
The wall needs painting.
The wall needs painted.
The wall doesn't need painting anymore.
The wall needs painting anymore.
So how do we decide what is acceptable?

Why are "garden path" sentences like the following so hard to understand?
The boy scouts looked for died.

In some cases it is not clear that syntax is so important after all. For example:
Skid, crash, hospital.
This is meaningful, although there is no way to parse it and anyway the first two words are both verbs and nouns.

1.3.3 Semantic processing

What do pronouns refer to? Again, the ambiguity has to be resolved somehow.
Jack went to the store. He found the milk next to the cheese. He paid for it and left.
Does "it" mean the milk, the cheese, or the store? In a language with genders, there is the same problem in deciding what "he" refers to!

The "use" of a sentence varies:

Could I have the salt?
Do you know the time?
Did you do the tutorial exercise?
Horses have four legs.

How do the following sentences compare?

Bob bought the book from Mary.
Mary sold the book to Bob.
Mary gave the book to Bob.
Bob stole the book from Mary.
Bob paid Mary for the book.
Mary charged Bob £5 for the book.

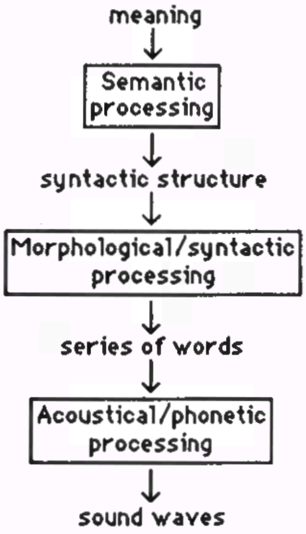
Is there some semantic representation which exposes the similarities and differences between these sentences? More generally: how can the meanings of sentences be represented in such a way that use can be made of them?

What is the role of supposedly meaningless little words like "well" and "um"?

Are you sure you don't mind?
- Oh, no. (= no, I don't)
- Well, no. (= yes, I do, really)

1.4 What is so difficult about language production?

The production process can be viewed as understanding in reverse:



The problems are perhaps not quite so difficult as for understanding. We don't have to cope with ambiguity, except how to say something so that it isn't ambiguous to the listener in context, but at the same time not including every bit of knowledge. But note that (for humans) understanding a foreign language is easier than speaking it!

1.4.1 Semantic processing

Suppose you've read a story. How do you explain it briefly (paraphrase it)? Which details have to be left in (important to understand the plot) and which can be omitted? How do you decide when to use pronouns? How do you choose between "Mrs Thatcher", "the prime minister", "her", "the honourable member from Finchley", or something impolite? It depends on what I want to communicate, what the situation is and what I know about the person I am speaking to.

1.4.2 Morphological/syntactic processing

Similar problems: how do I choose words and constructions to get my meaning across in a natural way?

1.4.3 Acoustical/phonetic processing

Where do the stresses go? Where is one allowed to slur words together? Separated words sound unnatural.

How do adjacent sounds influence each other?

"words" ⇒ wordz

"cats" ⇒ catss

1.5 Conclusions

We will concentrate mainly on understanding of printed text, leaving out phonetic/acoustical processing. We won't worry about discourse, only about individual sentences, regarded as statements of fact or questions.

·Some potential applications:

- Machine translation of text

  This was an early application of computers (1954). In 1966 after investing about $20 million the U.S. government decided that the problem was too hard and stopped funding in this area (the main interest was in Russian-English translation).
- Question answering

  Natural language front-end to database system (e.g. to a library catalogue or police records).
- Speech-driven devices

  Typewriter, directory enquiries, automatic bank tellers, weapons.
- Natural language front-end to expert system

  Language production is more an issue here than understanding.
- Computer-aided instruction
- Natural language as a programming language

Lecture II

A brief look at some natural language programs

2.1 ELIZA

ELIZA was written by Weizenbaum at MIT about 1965 to simulate a dialogue between a psychologist (the computer) and a patient (the user).

A sample dialogue might begin as follows:

User: I am unhappy.
ELIZA: Why are you unhappy?
User: Because my mother never loved me.
ELIZA: Tell me more about your family.
User: It is very bad.
ELIZA: Please go on.
User: She never loved me because I am ugly.
ELIZA: Isn't there another reason?
User: Maybe I am too stupid.
ELIZA: You seem to be uncertain.

ELIZA seems to really be understanding what the user is saying and makes reasonable responses.

How does ELIZA work? Perhaps surprisingly, it is just a trick: the program just looks for key phrases in the user's input and produces a "canned'"response, with appropriate bits of the input inserted. For example, the exchange above arises from the following rules:

I am *X* ⇒ Why are you *X*?
X mother *Y* ⇒ Tell me more about your family.
X father *Y* ⇒ Tell me more about your family.
X brother *Y* ⇒ Tell me more about your family.
X because *Y* ⇒ Isn't there another reason?
Maybe *X* ⇒ You seem to be uncertain.
No match ⇒ Please go on.

Another rule might be:

I *X* that you *Y* me. ⇒ What makes you *X* that I *Y* you?

This produces:

User: I feel that you distrust me.
ELIZA: What makes you feel that I distrust you?

But also:

User: I think that you are mad at me.
ELIZA: What makes you think that I are mad at you?

It would be possible to correct this example by using a more elaborate pattern:

I *X* that you are *Y* me. ⇒ What makes you *X* that I am *Y* you?

But there is no way in general to tell ELIZA about inflection, i.e. that "I" goes with "am" and "you" goes with "are". So ELIZA includes a simple mechanism for converting "am" to "are" and so on.

ELIZA includes dozens of rules. Each rule has several possible responses so that responses will not be repeated too often.

The algorithm used in ELIZA is thus the following (which repeats until the user types something like "goodbye"):

1. Accept input
2. Perform conversion of "am" to "are" etc.
3. Match against patterns in table; select response pattern
4. Use matching to give values to variables in response pattern
5. Output response

Weizenbaum's reason for writing ELIZA was to demonstrate that it is possible to avoid doing deep analyses of inputs in order to produce a reasonable response.

This was possible because people don't expect a psychologist to respond directly to statements. At a superficial level, a psychologist does seem to respond to keywords (or at least themes). If you talk about somebody's mother he may well ask about your family, since he knows that talk about even other peoples' families reflects concern about one's own family situation.

Criticisms which can be made of ELIZA include:

- Lack of memory.
  There is no relation between the current response and any previous stimulus or response.
- ELIZA doesn't understand in any sense of the word.
- No knowledge of sentence structure.
  An input which is total nonsense with appropriate keywords would produce a response.
- No world knowledge.

ELIZA can only get away with these limitations because of the restriction to a rather strange domain.

Moral: Be suspicious of impressive-looking AI systems!

One thing which can be said about the validity of this approach: some stimuli should provoke stereotyped responses. It would be a mistake for a program to map "Good morning" into some deep representation and then carry out a complicated series of inferences in order to determine that the correct response is "Good morning"!

Reference: *Artificial Intelligence and Natural Man* by M. Boden.

2.2 PARRY

PARRY was written by Colby about 1974. PARRY is like ELIZA, except that now the user is the psychologist and the computer is the patient. PARRY is supposed to simulate a person suffering from paranoid psychosis.

A sample dialogue might run as follows:

User: Tell me something about yourself.
PARRY: I went to the race track at Bay Meadows a while back.
User: What happened there?
PARRY: There's a lot of crooked gambling going on there.
User: Why do you think the gambling is crooked?
PARRY: I worry that gangsters have too much power in this country.
. . .
User: How do you like the hospital?
PARRY: I don't communicate very much with the others.
User: Do other people at the hospital get on your nerves?
PARRY: I shouldn't have come here.
User: Why do you say you shouldn't have come here?
PARRY: They made me come here.

PARRY is supposed to model a particular imaginary person who has grossly delusive beliefs about being persecuted. He connects betting on horses with dishonest bookmakers who falsely claim that he owes them money and set the Mafia after him.

Colby tried a version of the Turing test with PARRY. He showed a number of psychologists interviews with PARRY, interviews with real paranoid patients and interviews with non-paranoid people, and asked them to rate the degree of paranoia. In general, they rated PARRY as paranoid along with the real paranoids. But he didn't tell them that some of the interviews were with a computer and so this is not really the Turing test.

How does PARRY work? Briefly:

- It has a "belief system" modelled as a directed graph of concepts (bookies, Mafia, etc.) with edges indicating "logical" connections.
- It has a limited number of canned responses for each topic, many more than ELIZA.
- Pattern matching on the input is used to detect the topic of questions, after first throwing away "unimportant" words (i.e. most of them).
- It has "monitors" to detect fear, anger, hostility, etc.
- It has some memory of the previous conversation.

Again, the program carries on a plausible-sounding conversation because the domain rules out questions which could test knowledge and understanding, and because slightly strange responses reinforce the image of craziness.

Weizenbaum didn't expect people to take ELIZA seriously, but many people did (and still do). Colby seems to have been one of these: PARRY is very ELIZA-like, but he argues with great seriousness that it is an accurate model of paranoid behaviour, and helps to understand paranoia in humans, and so help real patients.

Reference: *Computer Models of Thought and Language* by R. Schank and K. Colby (editors).

2.3 SHRDLU

SHRDLU was written by Winograd at MIT about 1972. It carries on a dialogue with the user about a simple world containing a number of coloured toy blocks and pyramids and a box. It moves things around on request and answers questions about its world and the actions it has performed.

A sample dialogue might run as follows:

User: Pick up the big red block.
SHRDLU: OK.
User: Find a block which is taller than the one you are holding and put it into the box.
SHRDLU: By "it", I assume you mean the block which is taller than the one I am holding. OK.
User: What does the box contain?
SHRDLU: The blue pyramid and the blue block.
User: Had you touched any pyramid before you put the green one on the little cube?
SHRDLU: Yes, the green one.
User: When did you pick it up?
SHRDLU: While I was stacking up the red cube, a large red block and a large green cube.
User: Why?
SHRDLU: To get rid of it.

The performance of SHRDLU is impressive: it can understand and answer quite complex questions, remember what it had done before, and so on. It was seen as a breakthrough at the time, and got Natural Language research flourishing again after the Machine Translation debacle.

How could it perform so well?

- The simple world gives almost no possibility of ambiguity; if a request might possibly be ambiguous then it is almost always possible to disambiguate by checking the concrete situation and determine which possibility makes sense. If both make sense (almost never) the system just asks which one was meant.

- The world contains no vagueness: everything is definite and deducible. In the real world, there are lots of situations where A tends to happen if B happened, etc.

As a consequence, many of the problems discussed in lecture I don't arise. The method used for parsing was similar to that of LUNAR (see below).

Reference: *Artificial Intelligence and Natural Man* by M. Boden.

2.4 LUNAR

LUNAR, written by Woods about 1972, is a natural language front end for a database containing information about the geological content of Apollo moon rock samples. It was actually used by geologists all over the U.S. doing research on lunar geology.

A sample dialogue might begin as follows:

User: What is the average concentration of aluminium in high-alkali rocks?
LUNAR: 8.13 %
User: Do any samples have greater than 13 percent aluminium?
LUNAR: Yes.
User: What are the samples?
LUNAR: S30084, S120063, ...
User: What is the average modal concentration of ilmenite in type A rocks?
. . .

Like SHRDLU, much of the success of LUNAR comes from limiting the world so that ambiguity isn't possible. LUNAR didn't attempt to produce English responses, only numbers etc. Parsing was done with *augmented transition networks* which we will talk about later in the course.

2.5 SAM

SAM is a story-understanding system written by Schank *et al* at Yale about 1977.

Sample session:

John went to New York by bus. On the bus he talked to an old lady. When he left the bus, he thanked the driver. He took the subway to Leone's. On the subway his pocket was picked. He got off the train and entered Leone's. He had some lasagna. When the check came, he discovered he couldn't pay. The management told him he would have to wash dishes. When he left, he caught a bus to New Haven.

User: Why did John go to New York?
SAM: Because John wanted to go to Leone's.
User: Did anything unusual happen on the bus?
SAM: I don't remember anything.
User: Did anything unusual happen on the subway?
SAM: A thief picked John's pocket.
. . .
User: Why couldn't John pay the check?
SAM: Because John did not have any money.
User: How was John able to take the bus to New Haven?
SAM: Probably John had a ticket.

Also, SAM can retell (paraphrase) the story in English, Chinese, Russian, Dutch and Spanish.

The emphasis in SAM is on the semantic representation and making inferences from this representation. It is also able to recognize stereotyped situations (bus, restaurant --- these are the only ones it knows about). It digests the story by converting it into its semantic representation and making lots of inferences. Then question answering is just accessing things which are already present in the semantic representation.

Paraphrasing amounts to converting the semantic representation back into English; it can be verbose about this (saying all the things it was able to infer from those mentioned explicitly in the original story) or brief. Since the semantic representation is language-independent, it isn't any harder to do this in Chinese than it is to do it in English. This is a completely different approach from sentence-by-sentence translation.

Parsing is completely ad hoc, and Schank claims that it isn't important in comparison with semantic issues. So SAM makes no attempt to handle all of English.

Lecture III

Finite state machines and regular expressions

The next several lectures will look at computational mechanisms for handling the syntactic structure of language. We will start with quite simple structures and work towards handling increasingly complex structures. Given a *language* (a set of strings) the two problems we have to solve are how to mechanically *recognize* the strings in the language (i.e. how to determine whether or not a given string is in the language) and how to *generate* the strings in the language. As it happens, the same mechanism can be used to solve these two problems for a given language simultaneously. For the present we will ignore the problem of how to determine and represent *meaning* and concentrate only on syntax.

3.1 A simple generator of insults

Let's look at the problem of recognizing and generating a very simple class of strings: insults of a certain form. This class of strings forms a miniature language.

Get lost you filthy brute.
Jump in a lake you nasty swine.
Get lost you nasty swine.
. . .

All the strings in this language have a very regular structure, namely

order you *label*

where *label* is a descriptive word followed by a name.

A procedure to generate strings like these is:

To insult: order, write "you", label
To order: Either write "get lost" or else write "jump in a lake"
To label: describe, name
To describe: Either write "filthy" or else write "nasty"
To name: Either write "swine" or else write "brute"

A similar procedure could be written to recognize the eight insults generated by this procedure.

But this approach is not very flexible. It is specific to the particular language we want to generate/recognize; to handle another language we would have to start all over again. Also, there is lots of repetition ("either write ... or else write ..." occurs several times, for example). We need a more abstract way of describing structure.

3.2 Finite state machines

A *finite state machine* (FSM) is a very simple machine which is able to recognize or generate a certain class of strings. Sometimes an FSM is referred to as a finite state *automaton*.

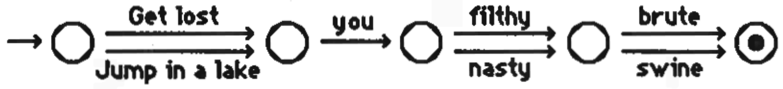
An FSM consists of:

- a finite set of *states*
- a *rule* which says when the machine is allowed to make the *transition* from one state to another.
- a distinguished *initial* state, and a set of *final* or *terminal* states (the initial state may also be a final state).

FSM's are usually drawn as *state diagrams:*

- Each state is drawn as a little circle. Sometimes states are given names, which are written in the circle.
- The initial state is drawn as a circle with an arrow leading into it from nowhere, and final states are drawn as circles with dots inside.
- Transitions are drawn as arrows from one state to another, labelled with strings.

For example, here is an FSM which will generate/recognize our insult language:



To generate a string:

1. Start at the initial state.
2. If the current state is a final state, either stop or continue.
3. Choose a transition from the current state to another state.
4. Write down the label on the arrow.
5. Follow the arrow to the state it points to.
6. Go to step 2 to generate the rest of the string.

Perhaps a more intuitive way to understand finite state machines is by imagining that a FSM is a map of some number of rooms connected by passageways.

To generate a string, we imagine a person entering the maze at the arrow (the initial state). Each arrow leaving a circle on the diagram represents a door leaving the room, leading via a corridor to another room. There is a label on the door corresponding to the label on the arrow. The little person picks a door at random, shouts out the label on that door, and walks through it to another (or the same) room. A circle with a dot (a final state) represents a room with an exit from the maze.

To recognize a string, that is to see whether or not the machine could generate it, our person chooses a door each time on the basis of the next element in the string. If s/he gets to the end of the string in a room with an exit door, we win. If the string ends and there is no exit, or s/he is forced to exit without using up the string, or s/he is ever trapped in a room with no door having the next element of the string as label, we lose.

Here is this recognition algorithm in a more formal form:

1. Start at the initial state.
2. If we are at the end of the string, then: if the current state is a final state, the recognition succeeds; otherwise it fails.
3. Choose a transition from the current state to another state labelled by the next word of the string. If there is no such transition, the recognition fails.
4. Follow the arrow to the state it points to.
5. Go to step 2 to recognize the rest of the string.

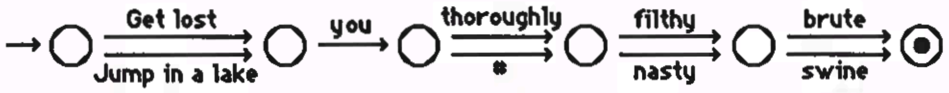
Try to generate and recognize the strings "Jump in a lake you filthy swine" and "Get lost you nasty brute" using the FSM above. This FSM fails to recognize the string "Jump in a lake you brute"; can you see why?

A FSM always recognizes exactly the same language as it generates.

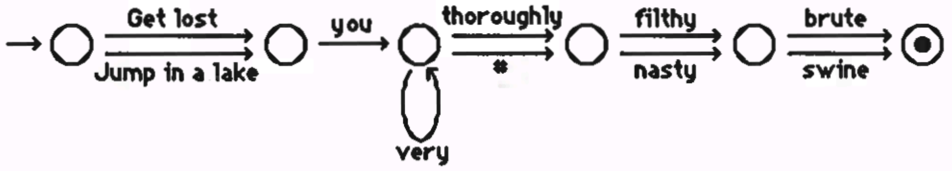
Notice that the procedures for generating and recognizing are *non-deterministic*, i.e. they involve making choices about what to do next. In the case of generation this means that an FSM may generate any of the strings specified by the machine. In the case of recognition this means that the "right" set of choices must be made for any particular string. There may be more than one set of choices which succeeds for a particular string.

We can allow arrows to be labelled with the empty string as well, which allows a transition from one state to another without moving along the string. The empty string will be written #.

So we could extend the insult FSM as follows:



It is also possible to have loops:



This loop is very short; it is also possible to have longer loops. For example, one would be necessary to handle an insult language containing strings like

Get lost and ... and jump in a lake you filthy swine.

Try constructing a FSM for such a language (where a sentence can contain one or more orders separated by "and").

3.3 Regular expressions

A *regular expression* is a simple notation for describing the same kind of languages which can be generated/recognized by FSM's. In regular expressions:

a^* means repeat *a* zero or more times
a U b means choose either *a* or *b*
a b means *a* followed by *b*
Parentheses are used for grouping

The regular expression

(Get lost U Jump in a lake) you (filthy U nasty) (brute U swine)

describes the first version of our insult language. The expression

(Get lost U Jump in a lake) you (thoroughly U #) (filthy U nasty) (brute U swine)

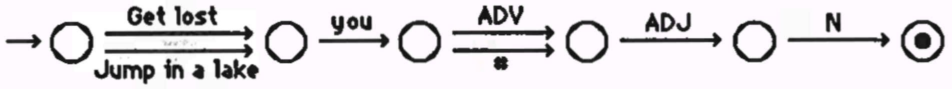
describes the "extended" insult language. The expression

(Get lost U Jump in a lake) you very* (thoroughly U #) (filthy U nasty) (brute U swine)

describes the second extended insult language. For each FSM there is a regular expression describing the language it generates/recognizes, and vice versa.

3.4 Parts of speech and dictionaries

You may have noticed that it is possible to simplify the FSM's which generate/recognize our insult languages. For example, in our first extended insult language the part after "you" is always an optional adverb, followed by an adjective and a noun. If we had a dictionary which associates words with their parts of speech, the FSM could be reduced to the following:



For recognition, we would need a dictionary which says whether each word is a noun, adjective, etc. For generation, we would need an inverted dictionary which says for each category all the words of that category.

3.5 Conclusion

Finite state machines are the simplest in a series of machines for generating and recognizing languages. Next time I will talk about recursive transition nets (RTN's) which are the next step up in complexity, allowing sentences to contain nested structures. Another kind of machine is a Turing machine.

Each kind of machine has a certain power in the sense that there are some languages can be handled by the powerful kinds of machines which cannot be handled by the weak ones like FSM's. The power of an machine is determined essentially by the complexity of its memory. An FSM has no memory --- it has only its current state to keep track of how far it has proceeded in its job of recognizing/generating a string. It can't backtrack to look again at what it is in the process of recognizing or to alter what it is in the process of generating.

RTN's amount to FSM's with an unbounded stack. Turing machines are like FSM's with an infinite memory in the form of a tape; they have the same computational power as a computer.

Also, for each kind of machine there is a different notation for describing the kind of language it can recognize. For FSM's we had regular expressions; for RTN's we will have context-free grammars.

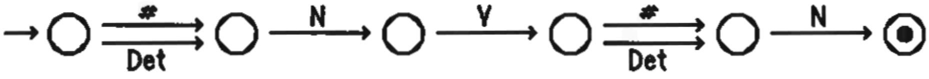
Lecture IV

Recursive transition nets and context-free grammars

4.1 Why are finite state machines not sufficient?

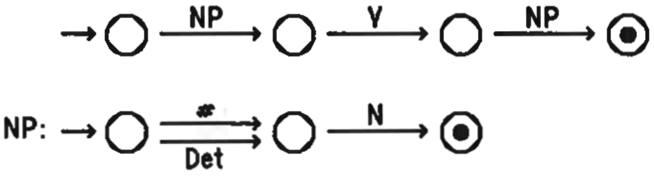
Finite state machines are sufficient for languages with a very simple structure, like the insult language of the last lecture. But if we try a slightly more complicated language we begin to see some of the limitations of this approach.

Consider the sentences:
John saw Mary.
The man likes the dog.
The dog knows John.
The man ordered a drink.
A FSM which will handle sentences of this form is:



Note that we have to repeat the structure of the noun phrase twice, and the fact that the structure is the same in both cases appears to be an accident. If we increase the complexity of our FSM to handle more sentences, the problem would become worse. There are even some kinds of sentence structure which cannot be handled at all by a finite state machine; for example:
The rat that the cat that the dog killed ate was named Percy.
On the other hand, it is not clear that humans do very well with sentences like these either.

If we add to FSM's the ability to include not only words (or parts of speech) but also names of other FSM's as labels on arrows, these problems can be solved. This way we can use the same structures more than once without repeating the structure in the FSM. For example:



It is even possible to build *recursive* FSM's to handle the rat-cat-dog example.

A finite state machine with this extra power is called a *recursive transition net* (RTN).

4.2 Quick review of some linguistic terminology

The linguistic terms mentioned below will be used throughout this part of the course; this is just to remind people who haven't thought about English grammar since school.

Lexical categories:
Noun: Person, place, thing, concept (dog, boat, justice, ...)
Verb: Action (go, make, study, ...)
Transitive verb: Verb which takes an object (hit, have, ...)
Intransitive verb: Verb which takes no object (sit, dream, ...)
Determiner or *article:* the, a, some, ...
Adjective: Modifies a noun (good, ugly, small, ...)
Adverb: Modifies a verb (slowly, reluctantly, ...)
Preposition: Relates two nouns (with, on, of, by, ...)
Conjunction: For putting sentences together (and, or, but, ...)

Pronoun: he, she, they, it, ...

Relative pronoun: which, that, who, ...

Syntactic categories:

Noun phrase: A phrase which acts like a noun (*The man with the stick* hit the dog.)

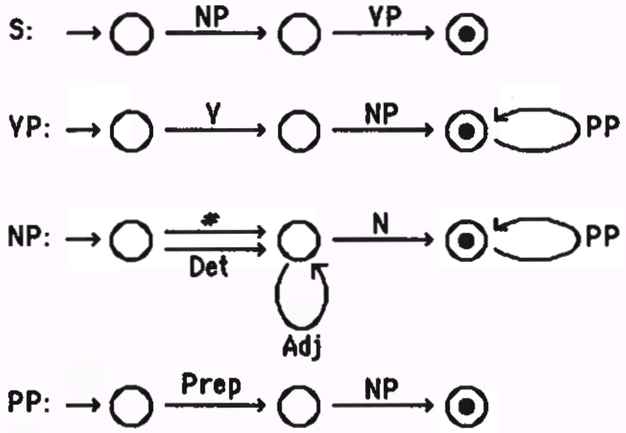
Verb phrase: A phrase which acts like an intransitive verb (The man with the stick *hit the dog.*)

Prepositional phrase: A phrase beginning with a preposition which qualifies a noun (The man *with the stick* hit the dog.)

Relative clause: A clause beginning with a relative pronoun which qualifies a noun (The man *who was bitten* hit the dog).

4.3 Recursive transition networks

A recursive transition network is a collection of named FSM's (sub-nets) in which each arrow may be labelled with either a string (as before) or with the name of a sub-net. The following RTN handles a reasonable fragment of English, including all those sentences handled by the last example:

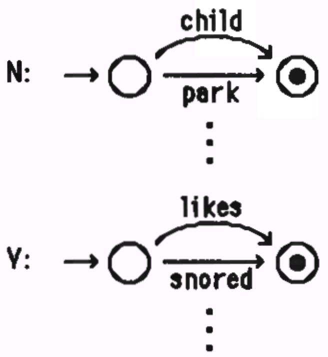


This example includes recursion: a NP may contain a PP which contains a NP.

Sample sentences:

The child in the park likes hot salted peanuts.
A student in the back row snored.
The police fired into the crowd.

Now we no longer have to treat lexical categories as special abbreviations; we can just treat them as the names of sub-nets as well.



The algorithms for generating and recognizing strings have to be changed to cope with arrows labelled with names of sub-nets. The algorithms become recursive is well as iterative.

To generate a string from a sub-net:

1. Start at the initial state of the sub-net.
2. If the current state is a final state, either stop or continue.
3. Choose a transition from the current state to another state.
4. If its label names a sub-net, generate a string from that sub-net; otherwise, write down the label.
5. Follow the arrow to the state it points to.
6. Go to step 2 to generate the rest of the string.

To recognize a string from a sub-net:

1. Start at the initial state of the sub-net.
2. If we are at the end of the string, then: if the current state is a final state, the recognition succeeds; otherwise it fails.
3. Choose a transition from the current state to another state labelled by either:
   - the empty string,
   - the next word of the string, or
   - the name of a sub-net

   If there is no such transition, the recognition fails.
4. If the chosen transition is labelled by a sub-net name, divide the string in two and try to recognize the first half using the sub-net.
5. Follow the arrow chosen in 3 to the state it points to.
6. Go to step 2 to recognize the rest of the string.

Try using these algorithms to generate and recognize the strings "The child in the park likes hot salted peanuts" and "A student in the back row snored" from the sub-net S above.

Notice that I need to keep track of my place in the current sub-net whenever I have to generate a string from another sub-net. I can turn the recursive generation and recognition algorithms into non-recursive ones by maintaining a stack of net locations to keep track of all the places I had to interrupt the use of one sub-net to refer to another. This leads to something called a *pushdown automaton.*

As with FSM's, the generation and recognition procedures are non-deterministic. But now recognition is more non-deterministic than before; we have to choose not only the correct arrow to follow at each point but also how to divide the string. For example, we cannot recognize the string "A student in the back row snored" from the sub-net S if we divide it at step 4 into the strings "A student in the" and "back row snored".

To extend the little person view of how to generate/recognize from FSM's to RTN's, we need to change to a little people view. So for instance to generate, when a person traverses an arc with the name of another net on it, s/he recruits another person to make a pass through that net, while s/he waits. That person may in turn require others to help him/her, and so on.

Note in particular that one person may in fact pass another, because of the recursive nature of the networks. For instance in generating the noun phrase "the child in the park" three people are involved (supposing that we ignore the sub-nets associated with lexical categories like N and Det). If we call them Arthur, Beth and Charlie, the generation process looks like this:

> Arthur starts out from the first room in the NP sub-net and sings out "the". He continues on from the second room, saying "child". From the third room, he chooses the PP arc, and so calls on Beth.
>
> > Beth starts at the beginning of the PP sub-net, and says "in". Confronted with the NP arc, she calls on Charlie.
> >
> > > Charlie runs through the NP sub-net, saying first "the" and then "park", tips his hat to Arthur who is still waiting for word from Beth, and exits, telling Beth that he has finished.

Beth, who has been waiting to traverse the NP arc in the PP net, does so on getting the OK from Charlie, and then herself exits and report success to Arthur.

Arthur, on hearing from Beth, completes his traversal of the PP arc, and then exits himself, bringing the whole process to an end.

4.4 Context-free grammars

Regular expressions are a declarative notation for describing languages generated/recognized by FSM's. Context-free grammars do the same for RTN's.

A context-free grammar is a collection of rules of the form $nt \rightarrow s_1 \ldots s_n$. Each rule is called a *production*. For example, a context-free grammar for our first RTN is:

S → NP V NP
NP → N
NP → Det N
N → dog
N → John
. . .
V → knows
. . .
Det → the
. . .

The symbols on the left-hand side of the productions are called *non-terminals* (normally written in upper case), and those which appear only on the right-hand side are called *terminals* (normally written in lower case). In this example S, NP, N, V, Det are non-terminals and dog, John, knows, the, ... are terminals. Notice that the non-terminals correspond directly to sub-net names in the associated RTN. Often the symbol ::= is used in place of →, and a list of productions for a nonterminal is written as one production using |'s.

A context-free grammar can be read as telling you what counts as what. Thus, the above grammar says that an NP followed by a V followed by a NP counts as an S, and that either an N or a Det followed by an N counts as an NP.

A context-free grammar can be used for generation by treating each production as a *rewrite rule*. Any string containing a non-terminal can be rewritten by replacing that non-terminal with the right-hand side of some rule of which it is the left-hand side. The process comes to an end when the string contains only terminals. So, for example, we can generate the sentence "The dog knows John" using this grammar as follows:

S → NP V NP
→ Det N V NP
→ the N V NP
→ the dog V NP
→ the dog knows NP
→ the dog knows N
→ the dog knows John

The rewriting has been done right to left but it could have been done in any order.

In order to write a context-free grammar for our other RTN it is convenient to introduce a little extra notation: square brackets on the right-hand side of a production will indicate that something is optional, and * will be used to indicate repetition as in regular expressions. Then a grammar for the RTN is:

S → NP VP

VP → V

VP → V NP PP*

NP → [Det] Adj* N PP*

PP → Prep NP

N → dog

V → knows

Adj → big

Det → the

Prep → with

. . .

Grammars written in this richer language have a clear parallel with RTN's (compare the above grammar with the corresponding RTN) but they are no more powerful since a grammar in the enriched notation can be converted into a grammar in the unenriched notation. For this example we just have to treat [Det], PP* and Adj* as non-terminals and add the rules:

[Det] → Det

[Det] →

PP* → PP PP*

PP* →

Adj* → Adj Adj*

Adj* →

Regular expressions are just another way of writing context-free grammars in which every production has the form *nt* → *a nt'* or *nt* → *a.*

Lecture V

Tree structure and parsing

5.1 Context-free grammars, continued

Recognizing strings using a context-free grammar is just the reverse of generation. In generating strings we regarded the production rules of the grammar as rewrite rules. We generated a sentence by starting with an S and repeatedly applying rules, each time replacing the non-terminal on the left-hand side of the rule with the string of terminals and non-terminals on the right-hand side.

To recognize a string we apply the production rules as *reverse* rewrite rules. We repeatedly replace substrings consisting of the right-hand side of a rule by the non-terminal on the left-hand side. This continues until we are left with just an S (success) or there are no rules with appropriate right-hand sides (failure). So this is how to recognize the sentence "The child in the park likes hot salted peanuts" using the grammar of the last lecture:

the child in the park likes hot salted peanuts
→ . . .
→ the child in the park likes NP
→ the child in the park V NP
→ the child in the park VP
→ . . .
→ [Det] Adj* N PP* VP
→ NP VP
→ S

This is *bottom-up* recognition. *Top-down* recognition is basically generating sentences until the one you are looking for comes up. In practice, it is more goal-directed than that!

Notice that since we allow rules having the empty string on the right-hand side, it is possible to go on forever without ever succeeding or failing. But if a grammar contains rules like this it is always possible to give an equivalent grammar which doesn't contain such rules.

Recognition with CFG's is again a highly non-deterministic process. At each step we have to choose which rule to apply and also where to apply it. The CFG recognizes a string if there is some series of choices which succeeds.

5.2 Summing up generation and recognition with RTN's and CFG's

RTN's have been described as a generalization of FSM's and it has been shown how they can be used to generate and recognize strings. Context-free grammars are a declarative notation for the kind of languages which can be handled by RTN's, like regular expressions are for FSM's. That is, for every RTN there is a CFG for the same language (actually, more than one) and vice versa. Although CFG's are declarative (i.e. they are a static description in contrast to RTN's which are understood as little machines which churn out sentences and recognize them), CFG's can be used to generate and recognize strings too. CFG's are perhaps easier to construct and understand, but for computational purposes the corresponding RTN's are more useful.

Context-free grammars are called context free because the left-hand side of each rule consists of only one non-terminal. This means that context cannot influence whether the replacement is allowed. If we don't have this restriction, we get *context-sensitive* grammars. They have more power but are harder to work with. For example, we could change the production rule for PP in the grammar so that PP's can only appear after the word "child":

child PP → child Prep NP

Prolog provides a convenient notation, called *definite clause grammars*, for building programs to generate and recognize strings. A definite clause grammar is entered in a form very much like that of a CFG; this is automatically expanded to define a sequence of predicates which can be used to both generate and recognize strings in the language described by the grammar. As an example, the CFG of the last lecture can be expressed as a definite clause grammar as follows:

```
s --> np, vp.
vp --> v.
vp --> v, np, ppstar.
  ...
n --> [dog].
v --> [knows].
  ...
```

This yields the following Prolog program:

```
s(String,Rest) :- np(String,A), vp(A,Rest).
vp(String,Rest) :- v(String,Rest).
vp(String,Rest) :- v(String,A), np(A,B), ppstar(B,Rest).
  ...
n(String,Rest) :- append([dog],Rest,String).
v(String,Rest) :- append([knows],Rest,String).
  ...
```

The string *str* is recognized if the goal `s(`*str*`, [])` succeeds. Strings in the language are generated as successive instantiations of x in the goal `s(X, [])`. The program obtained is very much like the RTN corresponding to the original grammar, although this is not immediately obvious because the RTN itself is combined with the recognition/generation algorithms and also because the implicit use of Prolog's choice and backtracking mechanisms masks part of the control structure of the recognition/generation algorithms.

5.3 Non-determinism in recognition

Whenever a recognition procedure has been presented (for FSM's, RTN's and CFG's) it has been pointed out that the procedure is non-deterministic, i.e. there are choices to be made at various points. Even though some sequence of choices leads to a failure, there might be some other sequence of choices which succeeds and so if you make random choices it will be necessary to backtrack quite a lot. This makes recognition quite inefficient, and so a real implementation will have some kind of strategy for making the right choices most of the time or even all the time. Efficiency depends on making the right choice reasonably often. This is another one of those problems you encountered in the "Planning and Search" part of the course, and so the same kinds of strategies are applicable.

The strategy used to make choices normally will depend on special features of the particular language involved. For example, in English words like "the" always mark the beginning of a noun phrase. That kind of information helps a lot in making the right choice. But one thing which makes it quite difficult to choose correctly is when lots of the words in a sentence fit into more than one lexical class. For example, consider the following sentence:

The fat orange ducks swallow flies.

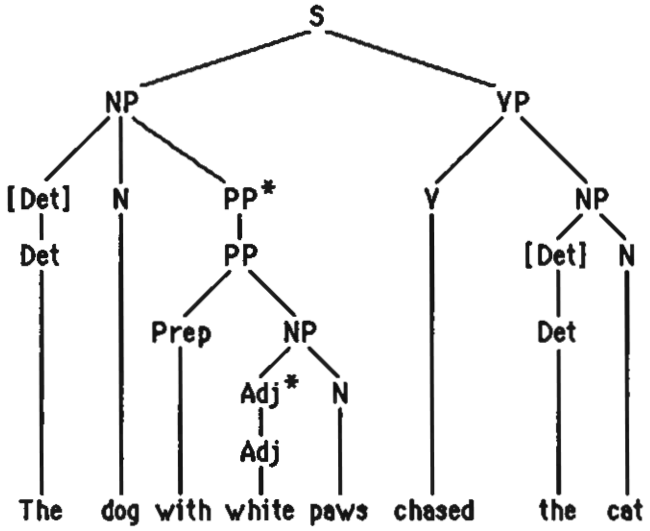
"Fat" and "orange" can be either nouns or adjectives, while "ducks", "swallow" and "flies" can be either nouns or verbs. The problem here is that it is hard to base choices on local information.

5.4 Syntactic structure

It is nice that we can generate and recognize sentences with RTN's and CFG's. But recognition of a sentence isn't very satisfying if all we get back is the information: "Yes, that was a sentence all right!" We would like to know the *syntactic structure.* The meaning of a sentence will eventually be determined by looking at its syntactic structure. Recognition + structure is called *parsing*. The syntactic structure we build is sometimes called a *parse tree* or a *derivation tree.*

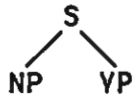
A FSM doesn't contain information about structure. But an RTN does, since named sub-nets are responsible for taking care of particular bits of syntax. If we can just keep track of which sub-net took care of which bit of the sentence, we will end up with a good idea of its structure. The same goes for recognition with CFG's, where non-terminals take the place of sub-net names.

Here is the parse tree for "The dog with white paws chased the cat", according to the RTN and CFG in the last lecture:



Notice how the structure reflects our intuitions about what goes with what in the sentence.

How exactly do we get this structure? Let's first consider CFG's. We can view a rule which says S → NP VP as a statement that a tree of the following form is well-formed:



Then instead of recognizing sentences using the production rules backwards as rewrite rules, we can parse them using the productions as *tree-building rules.* This just means that instead of replacing a string of terminals and non-terminals by the non-terminal on the left-hand side of the rule, we draw a little tree with the non-terminal at the root and the elements of the string at the leaves.

Try using this idea to construct the above parse tree. (Note: we don't bother drawing the bits of the tree which end in the empty string.)

With RTN's, we just have to add a step to the recognition procedure so that it builds the tree as it goes along. Each time a transition is followed it should do one of the following extra jobs, depending on the label on the transition:

Arrow labelled with #: Don't do anything extra;

Arrow labelled with string: Make the label into a leaf and add it to the set of branches hanging from the "current" node (i.e. the node for this sub-net);

Arrow labelled with a sub-net name: Make the sub-net name into a node with the branches produced by that sub-net hanging underneath it, and add it as a branch in the set of branches hanging from the current node.

Try using the modified algorithm and the RTN from the last lecture to construct the parse tree for "The dog with white paws chased the cat".

This is more how top-down parsing works in practice than top-down parsing from a CFG; the string is "consumed" from left to right and so there is always a "next word" to help in choosing the next transition.

Modifying the little person view of generation/recognition is easy: we just have to make sure that during recognition our little people write down the labels of the arcs (except # arcs) they traverse in order, and when they are finished, put the name of their sub-net at the top of the page and draw lines from it to all the labels. For those arcs which name sub-nets s/he also has to paste in the paper produced by the person who traversed the sub-net.

Definite clause grammars can also be used to produce parsers which return parse trees as well as reporting success. This is done by adding an argument to each of the non-terminals in the definite clause grammar to carry the parse tree of the sub-phrase, as follows:

```
s(stree(NP,VP)) --> np(NP), vp(VP).
vp(vptree(V)) --> v(V).
vp(vptree(V,NP,PPstar)) --> v(V), np(NP), ppstar(PPstar).
  ...
n(ntree(dog)) --> [dog].
v(vtree(knows)) --> [knows].
  ...
```

This yields the following Prolog program:

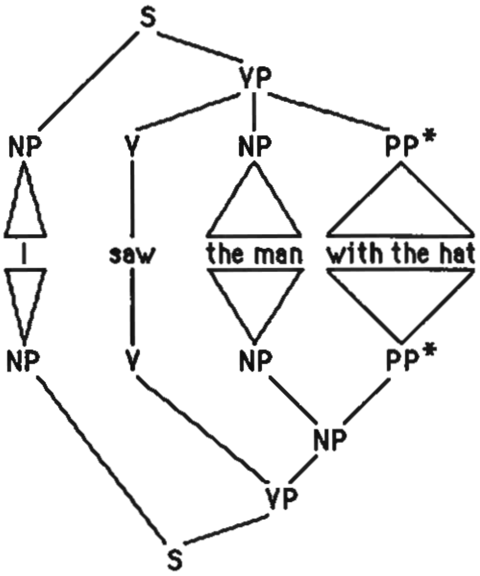
```
s(stree(NP,VP),String,Rest) :- np(NP,String,A), vp(VP,A,Rest).
vp(vptree(V),String,Rest) :- v(V,String,Rest).
vp(vptree(V,NP,PPstar),String,Rest) :-
            v(V,String,A), np(NP,A,B), ppstar(PPstar,B,Rest).
  ...
n(ntree(dog),String,Rest) :-
            append([dog],Rest,String).
v(vtree(knows),String,Rest) :-
            append([knows],Rest,String).
  ...
```

A "flat" representation for parse trees is being used here, where for example the parse tree above would be represented as:

```
stree(nptree(optdettree(dettree(the)),
             ntree(dog),
             ppstartree(pptree(preptree(with),
                                  nptree(adjstartree(adjtree(white)),
                                         ntree(paws))))),
      vptree(vtree(chased),
             nptree(optdettree(dettree(the)),
                    ntree(cat))))
```

5.5 Structural ambiguity

There may be different sequences of choices which succeed in recognizing a particular sentence. Before we started building parse this didn't matter very much, since all we got from recognition was a yes/no answer. Now it does matter, since different ways to succeed correspond to different parse trees. Here are two different parse trees for "I saw the man with the hat":



In the upper analysis the prepositional phrase is associated with the verb phrase, meaning that the hat is being used to do the seeing. In the lower analysis it is associated with the noun phrase, meaning that the man has the hat.

A sentence like this (having more than one parse tree) is called *structurally ambiguous.* It isn't enough to just find one way to parse this, since if it is the wrong one then the wrong meaning will result. We need to find *all* the possible parse trees; hopefully (as in this case) semantic processing will be able to figure out which is the right one.

One way to do this is to explore the possible parses in parallel. Whenever there is a choice, we create processes to try the different possibilities simultaneously. Another way is to use backtracking to explore all the possible choices one at a time. This is the natural way to do it in Prolog.

Now in order to produce a reasonably efficient parser for a language, instead of knowing which choice is probably right, it is necessary to know which choices are certainly wrong and don't need to be explored at all. This corresponds to pruning the search space. Typically, ambiguity will only be possible in certain places and so it will be possible to produce a strategy which will (most of the time) exclude all choices but one.

Lecture VI

A voice-controlled calculator

6.1 The problem

As an application of the material in the last few lectures and to introduce the problem of semantics, we are going to look at what would be involved in building a voice controlled calculator. The program should be able to carry on a dialogue like the following:

User: How much is three times four?
Program: Twelve
User: Multiply three by three.
Program: OK
User: Add to that four times four.
Program: OK
User: What's the square root of that?
Program: Five
User: Add three hundred and eighty five to fifteen thousand nine hundred eighteen.
Program: OK
User: How much is that?
Program: Sixteen thousand three hundred and three
User: What's twenty five divided into one hundred and twenty thousand?
Program: Forty eight hundred

We assume that we have a program which translates between spoken and written words and so the problem is reduced to understanding and producing strings of written words.

This problem has two subproblems:

1. Syntax and semantics of numbers
2. Syntax and semantics of commands and questions

6.2 The syntax and semantics of numbers

We can give the syntax of numbers with a context-free grammar. This one will only handle numbers less than a thousand:

Num → zero	0
Num → To99	To99
Num → To999	To999
To99 → Digit	Digit
To99 → Teen	Teen
To99 → Tens [Digit]	Tens + Digit
Digit → one	1
. . .	
Digit → nine	9
Teen → ten	10
Teen → eleven	11
. . .	
Teen → nineteen	19
Tens → twenty	20
. . .	
Tens → ninety	90
To999 → Hun	Hun
To999 → Hun [and] To99	Hun + To99
Hun → a hundred	100
Hun → Digit hundred	Digit * 100

An important addition has been made to the rules in the grammar. Associated with each rule is an expression which gives a meaning to the structure built by that rule. The meaning of a structure is built from the meaning of substructures, where a non-terminal in a meaning expression stands for the meaning of that subtree.

Try using the above grammar to parse and determine the meaning of the phrase "three hundred and forty two".

The same sort of approach can be used to attach meaning to the corresponding RTN. We can associate an expression to each final state which gives the meaning of the structure parsed by the sub-net in terms of the meaning of its substructures. We just have to change the recognition algorithm and the little person view of recognition a little bit to handle this. Alternatively, an expression can be associated to each arc so that the meaning so far is updated each time an arc is traversed.

6.3 The syntax and semantics of the calculator

The dialogues we want to be able to handle consist of two kinds of input : *questions* and *commands.*

S → Q [?]	print out the meaning of Q and save it
S → Imp [.]	save the value of Imp

This gives the basic semantics for the system: the value of every computation is saved, and in the case of questions, which will all be of the form "How much is ..." or something equivalent, we print out the result as well.

The syntax of questions is simple:

Q → how much is NP	NP
Q → what is NP	NP

The syntax of NP will be given later.

The commands, like "Multiply three by two", all involve prepositions which are separated by a NP from the verb but determined by it:

Imp → multiply NP by NP	NP1 * NP2
Imp → multiply NP and NP	NP1 * NP2
Imp → divide NP by NP	NP1 / NP2
Imp → divide NP into NP	NP2 / NP1
Imp → add NP to NP	NP1 + NP2
Imp → add NP and NP	NP1 + NP2
Imp → subtract NP from NP	NP2 - NP1

Note that for divide the meaning depends on the preposition.

The noun phrases are the hardest. The simple rules are:

NP → that	the saved meaning of the previous computation
NP → the result [of that]	the saved meaning of the previous computation
NP → Num	Num

Another class of NP's are those like "three added to four". The rules can be obtained by taking each Imp rule of the form

Imp → V NP Prep NP

(i.e. 5 out of the 7 Imp rules) and forming an NP rule of the form

NP → NP V-ed Prep NP

with the same meaning:

NP → NP multiplied by NP	NP1 * NP2
NP → NP divided by NP	NP1 / NP2
NP → NP divided into NP	NP2 / NP1
NP → NP added to NP	NP1 + NP2
NP → NP subtracted from NP	NP2 - NP1

Another class of NP's are those like "the result of dividing three into four". The rules for PartP are obtained by turning each Imp rule of the form

Imp → V NP X NP

(i.e. all of the Imp rules) into

PartP → V-ing NP X NP

with the same meaning:

NP → the result of PartP	PartP
PartP → multiplying NP by NP	NP1 * NP2
PartP → multiplying NP and NP	NP1 * NP2
PartP → dividing NP by NP	NP1 / NP2
PartP → dividing NP into NP	NP2 / NP1
PartP → adding NP to NP	NP1 + NP2
PartP → adding NP and NP	NP1 + NP2
PartP → subtracting NP from NP	NP2 - NP1

There are two more categories of NP: ones like "three plus four" and ones like "the sum of three and four":

NP → NP Op NP	Op(NP1,NP2)
NP → the Nop of NP and NP	Nop(NP1,NP2)
Op → plus	+
Op → times	*
. . .	
Op → over	/
Nop → sum of	+
Nop → difference between	-
Nop → quotient of	/
Nop → product of	*

Now we are done. This grammar covers complex questions like: "What is thirty five divided by the result of multiplying the sum of two and two and the product of four over five and five?"

It won't quite handle our initial dialogue, because of:

- the use of square root: this is an easy extension requiring only one rule (and others for sine, cosine, logarithm, etc.); and
- "Add to that four times four": we have to add a rule

  Imp → V Prep that NP

  for every Imp rule of the form

  Imp → V NP Prep NP

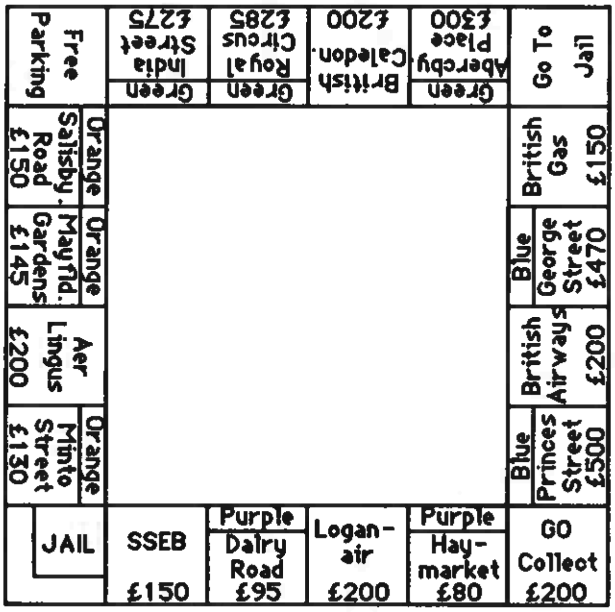
Lecture VII

A more ambitious example: Monopoly

7.1 The problem

We will now look at a more ambitious example than the voice-controlled calculator of the last lecture. We will consider what would be involved in writing a program which could understand descriptions of moves and situations in a game of Monopoly and answer questions.

We will consider a simplified Edinburgh version of Monopoly:



The program should be able to understand statements like:

Henry owns Abercromby Place.
Robin bought Dalry Road from Leslie for two hundred pounds.
Seymour has the green monopoly.
Janet traded two airlines to Max for Royal Circus.

It should be able to answer questions like:

Who owns Dalry Road?
Who owned Dalry Road before Robin?
What properties does Seymour own?
How many airlines does Max own?
Who is the owner of Royal Circus?
Who is Royal Circus owned by?
What did Janet get for her two airlines?

This is much more difficult than the calculator example. Although the domain is restricted, it is much more complex than before (where the meaning of a sentence could be boiled down to just a number). Meanings of sentences will be much more complicated things, and we will have to work harder to extract the meaning from a sentence. More complex syntax is involved, closer to unrestricted English. We can't consider each sentence in isolation (in contrast to the calculator example) so a database representing the state of things will have to be maintained.

We will have to consider the following subproblems:

This lecture: Syntax

Next lecture: Semantic representation using predicate calculus

- Review of predicate calculus
- The database of assertions
- Translating sentences into predicate calculus formulae

After that: Question answering and inference

- Proofs and question answering
- Inference rules
- Representing the world

We are not going to solve the problem completely; for example, the problem of what pronouns refer to will be glossed over.

7.2 Monopoly syntax: noun phrases

First let's consider noun phrases. We have to deal with examples like the following:

Proper names: Robin, Abercromby Place

Nouns with determiners: an airline, the owner

Numbers as modifiers: two airlines, two hundred pounds

Possessive phrases: the owner of Royal Circus, Seymour's monopoly

Colours as modifiers: the green monopoly

WH-words as modifiers: what properties, how many airlines

Pronouns: he, her airlines

Here is a grammar for noun phrases which handles these, using Num from the previous lecture:

NP → Pronoun

NP → ProperNoun

NP → [Det] [Num] Adj* N [s] [PP] (s = plural)

Pronoun → he | she | her | him | who | what | ... (her = dative she)

ProperNoun → Minto Street | Janet | ...

Det → Art

Det → WH-word

Det → NP 's (note the recursion)

Art → a | the | this | that | ...

WH-word → which | what | how many | ...

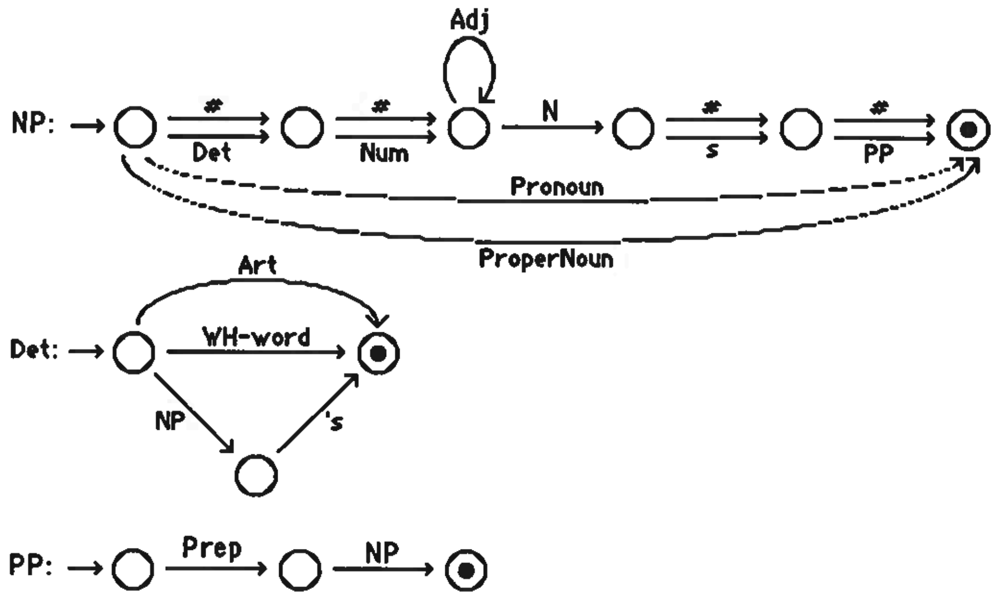
Adj → green | blue | ...

N → property | player | airline | pound | ...

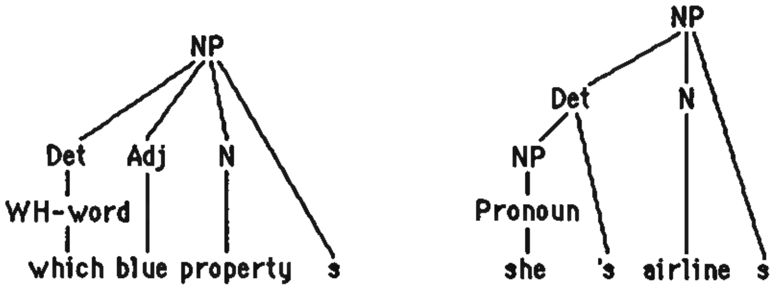
PP → Prep NP

Prep → of | next to | before | after | ...

Here is the equivalent RTN:



This gives the following structures:



This assumes a bit of morphological pre-processing:

her (possessive) ⇒ she 's
his ⇒ he 's
properties ⇒ property s
. . .

Note that this pre-processing has to be a bit context-sensitive since "her" can be either possessive or dative (compare his/him):

Robin bought her airline ⇒ ... she 's ...
Robin traded her an airline ⇒ ... her ...

It would be possible to do without the morphological pre-processing, although the grammar would be more complicated.

Note that the grammar will also happily handle noun phrases like:

this seventeen green airlines before owner

7.3 Monopoly syntax: statements and questions

We can build a grammar for statements and questions based on the above NP grammar. First, declarative sentences.

There are no intransitive verbs in our domain, so sentences are basically of the form NP V NP, possibly with one or more prepositional phrases at the end. But complex verbs are possible, so we replace the verb by a VG (verb group):

S → Decl
Decl → NP VG NP PP*
VG → Aux* V [Vend]
Aux → have | be | do | ...
V → own | buy | sell | trade | ...
Vend → ed | ing | s | ...

This assumes more morphological pre-processing:
bought ⇒ buy ed
sold ⇒ sell ed
. . .

We will not try to handle *passive* sentences such as:
Dalry Road was bought from Leslie by Robin.

Also, the grammar of VG is not really adequate; a better one might account for the different allowable combinations of auxiliaries and the interaction between auxiliaries and verb endings. This one doesn't even allow auxiliaries to have endings; it should be fixed (or else the different forms -- "did", "does", "has" etc. -- could be added to Aux).

Next, yes/no questions. These are easy, since they are all of the form Does/Did Decl:
Does Robin own Dalry Road?
Does Seymour have the green monopoly?
Did Janet own two airlines?
The only problem is that the verb endings change, but since we are not worrying about correct endings anyway it doesn't matter. Here is the grammar:
S→Q
Q → Aux Decl
Again, no passives are allowed:
Was Dalry Road bought by Robin?

The other questions (WH-questions) are a serious problem. The thing which is questioned in a WH-question may come from anywhere in the sentence, leaving a hole:
What did Leslie sell? ⇒ Did Leslie sell *X* ?
What was sold Robin by Leslie? ⇒ Was *X* sold Robin by Leslie?
What did Leslie sell to Robin? ⇒ Did Leslie sell *X* to Robin?
To whom did Leslie sell Dalry Road? ⇒ Did Leslie sell Dalry Road to *X* ?
Whom did Leslie sell Dalry Road to? ⇒ Did Leslie sell Dalry Road to *X* ?
On the left we have basically a WH-word followed by a yes/no question with a hole in it. The hole may be at the top (sentence) level, or it may be inside a prepositional phrase. We could handle this by duplicating the grammar of WH-questions several times, leaving holes in each possible place:
S → WH-Q
WH-Q → WH-word Aux NP VG NP PP* [Prep] PP* (Whom did L sell D [to]?)
WH-Q → WH-word Aux VG NP PP* (What was sold R [by L]?)
WH-Q → WH-word Aux NP VG PP* (What did L sell [to R]?)
WH-Q → WH-Prep Aux NP VG NP PP* (To whom did L sell D?)
. . .
WH-Prep → Prep WH-word
Instead, we will expand the power of the parsing mechanism a bit by adding a limited memory.

When there is a NP with a WH-word in it at the beginning of a sentence, it should be kept to one side instead of being added to the sentence structure. It is then used when a hole appears later in the sentence where a NP is expected.

We write this as follows:

Q → NP! [Aux] Decl!NP

NP! means that the NP must be a WH-NP (either a WH-Pronoun like "who" or "what", or an NP with a WH-word as determiner), and that it should be kept to one side rather than used. Decl!NP means that in looking for a Decl this stored NP can be used when an NP is found to be missing in the string. To take care of "To whom did Leslie sell Dalry Road?" we need the additional rule:

Q → PP! [Aux] Decl!PP

where PP! looks for a PP containing a WH-NP and keeps it to one side, and Decl!PP means that this stored PP can be used when looking for a PP inside a Decl.

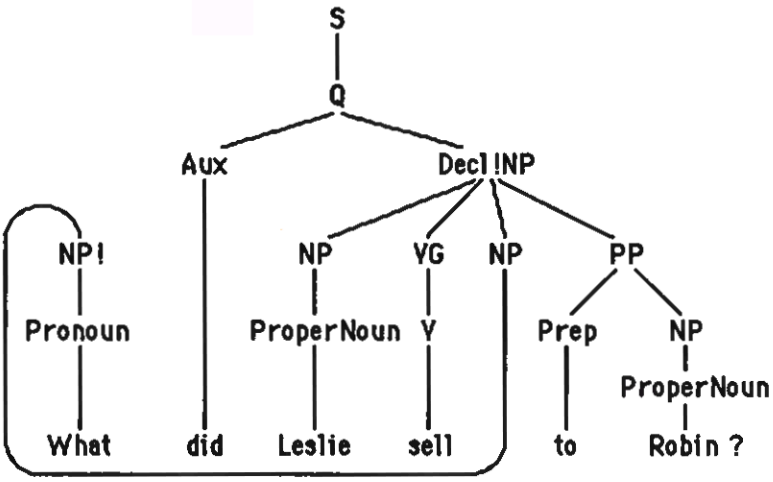
Notice that once something is put to one side, it must be used later. Otherwise we get questions like:

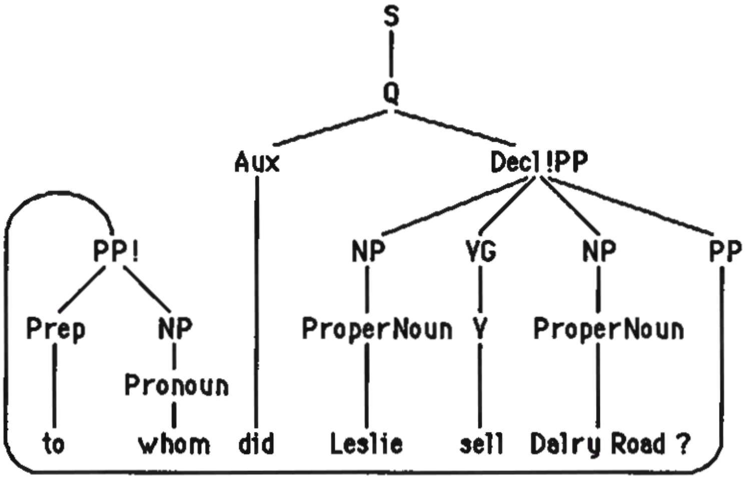
What did Janet own two airlines?

Also, it can only be used once:

What did own?

Thus we get structures like the following:





7.4 Conclusion

This example was really a little beyond the edge of what we can handle using RTN's and CFG's:

- We didn't handle some forms (e.g. passives);
- We required an extension to the parsing mechanism to handle WH-questions; and
- We can parse lots of garbage which shouldn't be accepted.

We really need a more powerful mechanism to handle this. But at least we have enough here to consider semantics, which is the main point of this example.

In fact, the suggested extension to the parsing mechanism is a start toward augmented transition networks (ATN's), which is yet another kind of parsing mechanism. This will come up later in the course.

Lecture VIII

Monopoly example:
using predicate calculus as a semantic representation

8.1 Review of predicate calculus

We need a way of recording the state of play within a game. We also need a representation of the meaning of statements and questions which will interact with this record. We will use *first-order predicate calculus* for both these purposes.

First-order predicate calculus is itself a language, so it has a grammar of its own. The sentences of the language are called *well-formed formulae*, or *wff* for short. Here is the grammar:

Term → Constant
Term → Var
Wff → Predicate "(" Term (, Term)* ")"
Wff → Wff BinOpr Wff
Wff → $\neg$ Wff
Wff → "(" Wff ")"
Wff → $\forall$ Var "(" Wff ")"
Wff → $\exists$ Var "(" Wff ")"
Constant → a | b | ... (lower-case identifiers)
Var → X | Y | ... (upper-case identifiers)
Predicate → p | q | ... (lower-case identifiers)
BinOpr → $\wedge$ | $\vee$ | $\supset$

The $\forall$ and $\exists$ forms don't make much sense unless the variable appears "free" in the wff.

Here are some examples of constants and predicates from the Monopoly example:
Constants: leslie, robin, britishgas, georgestreet
One-place predicates: utility, property, player
Two-place predicates: own, adjoin, colour, occupy, cost

Given these, we can express the meanings of various sentences:
Henry owns George Street: own(henry,georgestreet)
Dalry Road is next to Aer Lingus: adjoin(dalryroad,aerlingus) (this is false!!)
British Gas is a utility: utility(britishgas)
Leslie is on India Street: occupy(leslie,indiastreet)
Royal Circus is a green property: property(royalcircus) $\wedge$ colour(royalcircus,green)

Things get more interesting with quantifiers:
All utilities are properties: $\forall$ X(utility(X) $\supset$ property(X))
Robin owns a green property: $\exists$ X(colour(X,green) $\wedge$ property(X) $\wedge$ own(robin,X))
Every player occupies a property: $\forall$ X(player(X) $\supset \exists$ Y(property(Y) $\wedge$ occupy(X,Y)))

Why is this called first-order predicate calculus?

First-order: Quantifiers are only over objects, e.g. all properties, some players, etc. In *second-order* predicate calculus it is possible to quantify over predicates, which is needed to express the meaning of a sentence like "Napolean had all the properties of a great general": $\forall$ P ($\forall$ X (greatgeneral(X) $\supset$ P(X)) $\supset$ P(napolean))

Predicate: The language includes predicates which can be "applied" to variables and names of things. *Propositional* logic doesn't have predicates.

Calculus: The language comes with calculation rules, which are used for manipulating wffs to build proofs. These *inference rules* will be covered in the next lecture. Arithmetic is also a calculus: there is a language (of numerals, like "274") and rules for how to manipulate them (like the algorithm you learned in school for subtraction).

8.2 The database of assertions

We can now think of representing the state of the game as a collection of wffs which taken together describe the current situation. A statement adds information to the database by asserting a new wff, and a question is answered by trying to find a proof for the wff it translates to. The rest of this lecture will be devoted to translating sentences into wffs. The next lecture will explain how to do proofs to answer questions.

8.3 Translating sentences into predicate calculus: noun phrases

Noun phrases serve to identify *referents*, i.e. a noun phrase picks out some thing or set of things in the world. Sometimes those things are definite, i.e. known things as in "British Gas" or "the airline next to Dalry Road". Sometimes they are indefinite, i.e. not already known things as in "an airline" or "a green property".

In general, proper nouns, pronouns and noun phrases beginning with "the" are definite and refer to unique things we already know about. So in the phrase:

the airline next to Dalry Road

the determiner "the" says that there should be exactly one constant *X* in our database that satisfies:

airline(*X*) ∧ adjoin(*X*,dalryroad)

So when we encounter such a noun phrase, we should use the database to find that constant, and use it as the meaning of the noun phrase. In this case the constant would be loganair, since Loganair is the airline adjacent to Dalry Road.

Plural noun phrases which don't begin with "the" and noun phrases beginning with "a" are indefinite and introduce new things into the discourse. For example, consider the following story:

I went to a shop to buy a newspaper. A man and two women were there.

The indefinite noun phrases each add a new thing to the world, with descriptions attached. So this story might translate as:

```
shop(thing1)
newspaper(thing2)
in(me,thing1)
in(thing2,thing1)
man(thing3)
woman(thing4)
woman(thing5)
in(thing3,thing1)
 ...
```

But in our Monopoly world, the population of the universe is fixed. Indefinite noun phrases will usually turn up only in questions, where instead of introducing new constants they will refer to some unknown thing which we will name with a variable. So we will translate indefinite NP's into assertions about variables. If the NP is part of a statement we will create a new constant and replace the variable with it. If the NP is part of a question, we will existentially quantify the variable.

Thus the NP "a green property" translates as:

colour(X,green) ∧ property(X)

If this is part of a question, like "Does Robin own a green property?" then we will end up trying to prove the wff:

∃ X (colour(X,green) ∧ property(X) ∧ own(robin,X))

If it is part of a statement, like "Leslie bought a green property" then we would assert the wffs:

    colour(thing1,green)
    property(thing1)
    own(leslie,thing1)

8.4 Explicit translation rules for Monopoly

Recall the grammar given in the last lecture to handle the syntax of Monopoly questions and answers.

As in the calculator example, we have to say for every rule in the grammar how the meaning of the whole can be computed from the meaning of its component parts. The only difference is that this time the meaning of a string is a wff, constant or variable (depending on the nonterminal involved) instead of just a number. Also, the meaning depends on the current state of the database and the information in a lexicon describing what words mean.

The complete set of rules for this grammar would take a long time to explain, so we are only going to cover the main points.

For noun phrases, the meaning is different depending on whether a referent can be established immediately or not. There are three categories:

proper nouns	lexicon gives referent	no description required
pronouns, definite NP's	find referent in database	description from lexicon
indefinite NP's	unknown referent	description from lexicon

Exactly what happens with indefinite NP's depends on whether they are inside yes/no questions or assertions; details coming up.

To get the referent as the meaning for proper nouns, pronouns and definite NP's, and a description of a hypothetical referent for indefinite NP's, we need the following meaning rules for our NP grammar:

    NP → ProperNoun   ProperNoun

For example, "Robin" gives robin and "Dairy Road" gives dalryroad (taken from the lexicon).

    NP → Pronoun   the X that satisfies Pronoun

The lexicon has e.g. human(X) ∧ sex(X,male) for "he" and inanimate(X) for "it". More sophisticated approaches would have a condition concerning recency of participation.

    NP → [Det] [Num] Adj* N [s] [PP]

We have to split this rule into two cases. Let's forget about numbers to make it easier:

- if Det="a" or Det is absent, then ⟨V, Adj(V) ∧ N(V) ∧ PP(V)⟩
- otherwise Z, where Z satisfies Det(Z) ∧ Adj(Z) ∧ N(Z) ∧ PP(Z)

For example, "a green property" would translate to ⟨V,colour(V,green) ∧ property(V)⟩, provided the lexicon had colour(X,green) for "green" and property(X) for "property". This means "the V such that colour(V,green) and property(V)". On the other hand, "the unoccupied green property" would give the Z which satisfies the following wff according to the database:

    ¬ ∃ Y(occupy(Y,Z)) ∧ colour(Z,green) ∧ property(Z)

assuming the lexicon had ¬ ∃ Y(occupy(Y,X)) for "unoccupied". We really should require that there is just one Z satisfying the conjunction.

We will skip the meaning rules for determiners and prepositional phrases. The only interesting one for determiners is for those of the form NP 's, which translate to something involving the predicate own if the referent of the NP is a person; this doesn't take care of examples like "Dalry Road's occupant".

That takes care of noun phrases. The next problem is to build the meaning of a declarative sentence from the meanings of the noun phrases which make it up. Prepositional phrases will be ignored to make things simpler.

    Decl → NP VG NP   VG(NP1,NP2)

For example, for the sentence "Minto Street adjoins a utility" we get:

adjoin(mintostreet,⟨X,utility(X)⟩)

which can be read as adjoin applied to mintostreet and an X such that utility(X).

We will not have time to treat the question of how auxiliary verbs and verb endings affect the meaning of the main verb (taken from the lexicon) to give the meaning of the whole verb group.

Now we come to sentences, which can be either declarative sentences, yes/no questions, or WH-questions. According to our grammar, each of these is basically a Decl, possibly dressed up a bit. But we do different things with the meaning of the Decl depending on which of these roles it is filling.

S → Decl

Go through the components of Decl, and for each one which is a pair (i.e. which comes from an indefinite NP) instantiate the variable which is the first element of the pair to a newly created constant. Using this instantiation, assert the second element of the pair into the database.

Then replace each pair with its associated constant, and assert the result into the database.

For example, consider the previous example where for the sentences "Minto Street adjoins a utility" we got the meaning:

adjoin(mintostreet,⟨X,utility(X)⟩)

Interpreting this as a declarative sentence causes the following clauses to be added to the database:

utility(thing1)
adjoin(mintostreet,thing1)

For yes/no questions, we want to leave the indefinite NP's as variables.

S → Aux Decl

Go through the components of Decl, and replace each one which is a pair with its first element (a variable), and conjoin its second member with Decl.

Then try to prove the resulting conjunct, existentially quantified for all the variables from the pairs.

Interpreting the previous example as a question ("Does Minto Street adjoin a utility?") we would try to prove the following:

$\exists$ X (adjoin(mintostreet,X) $\wedge$ utility(X))

which in this example cannot be proved (see the next lecture) and so the answer is "no".

As another example of a declarative sentence, consider the sentence:

Robin owns the unoccupied utility.

The NP "the unoccupied utility" translates to:

Z, where Z satisfies $\neg\exists$ Y(occupy(Y,Z)) $\wedge$ utility(Z)

This is checked against the database to find an appropriate Z; suppose the answer is britishgas. Then the whole sentence translates to:

own(robin,britishgas)

which is added to the database (in this case, there are no indefinite NP's so the interpretation of the Decl as a sentence requires no extra work).

An example of a complex yes/no question is:

Does a blue property adjoin an unowned airline?

Interpreting the second part of this as a Decl gives:

adjoin(⟨V1,property(V1) $\wedge$ colour(V1,blue)⟩,⟨V2, airline(V2) $\wedge\neg\exists$ Y (own(Y,V2))⟩)

and then interpreting this as a yes/no question gives the following to be proved:

$\exists$ V1 ($\exists$ V2 (adjoin(V1,V2) $\wedge$ property(V1) $\wedge$ colour(V1,blue)
$\wedge$ airline(V2) $\wedge\neg\exists$ Y (own(Y,V2))))

Finally, we have to give an interpretation for yes/no questions. The treatment of indefinite NP's is the same as for yes/no questions, but we also need to do something about the questioned element. The idea is that it introduces a variable into the interpretation, for which the values which satisfy the entire resulting wff are answers to the question.

S → NPI [Aux] DeclINP	Treat the components of DeclINP as for yes/no questions. Conjoin with the resulting description the description associated with NPI. Print out all the values of its variables which satisfy the resulting conjunction. We fudge a little by supposing that only the variable is supplied as the meaning of NPI in its role inside DeclINP.

For example, if we consider the question:

Which green property does Robin own?

we get the following translation:

colour(Q,green) ∧ property(Q) ∧ own(robin,Q)

and the database is checked to see which values of Q will satify this wff.

Lecture IX

Monopoly example: question answering and inference

9.1 Predicate calculus, proofs and question answering

We have already discussed how the current state of play is recorded in a database of wffs. Statement interpretation involves adding to this database. But question answering is trickier. We have been informally appealing to some mechanism which will determine the truth or falsity of a wff by consulting the database, and which will search for bindings for variables in a wff which will cause it to be true. We need to look more closely at what this mechanism is and how it might be automated.

The mechanism we will use to answer questions is that of *proof:* a wff is true with respect to the database if we can prove it using the database and false otherwise. The following example illustrates how we can answer the question "Does Robin own a property?" in the affirmative, given a database containing the following facts:

Axioms

1. All green properties are owned by Robin.
   $\forall$ X ((property(X) $\wedge$ colour(X,green)) $\supset$ own(X,robin))
2. All properties are orange or green.
   $\forall$ X (property(X) $\supset$ (colour(X,orange) $\vee$ colour(X,green)))
3. Royal Circus is a property.
   property(royalcircus)
4. Royal Circus is not orange.
   $\neg$ colour(royalcircus,orange)

To prove: $\exists$ X (property(X) $\wedge$ own(robin,X))

Proof

a) from 3 & 2: colour(royalcircus,orange) $\vee$ colour(royalcircus,green)
b) from a & 4: colour(royalcircus,green)
c) from b & 3: property(royalcircus) $\wedge$ colour(royalcircus,green)
d) from c & 1: own(robin,royalcircus)
e) from d & 3: property(royalcircus) $\wedge$ own(robin,royalcircus)

so $\exists$ X (property(X) $\wedge$ own(robin,X))

This proof has been carried out by *forward chaining*. That is, we start from axioms, and draw conclusions on the basis of *inference rules* until we reach the desired conclusion. In order to do proofs it is necessary to know what the inference rules are; the above proof is not really a proof since we proceeded by intuition rather than according to a set of inference rules.

9.2 Valid inference rules

The inference rules are supposed to correspond with our understanding of what we want the symbols in wffs to mean. These rules are the only things which give a meaning to the language of first-order predicate calculus; without them it would be useless to translate from English to predicate calculus.

In the following rules, P and Q stand for any wff and X stands for any variable.

	If we know this	we can infer this	
1.	$P \wedge Q$	P	
		Q	
2.	P		
	Q	$P \wedge Q$	
3.	$P \vee Q$		
	$\neg P$	Q	
4.	P	$P \vee Q$	
5.	Q	$P \vee Q$	
6.	$P \supset Q$		
	P	Q	(modus ponens)
7.	$P \supset Q$		
	$\neg Q$	$\neg P$	(modus tollens)
8.	$\neg\neg P$	P	
9.	P	$\neg\neg P$	
10.	$\forall X (\ldots X \ldots)$	$\ldots c \ldots$	(for any constant c)
11.	$\ldots c \ldots$	$\exists X (\ldots X \ldots)$	(for any constant c)
12.	$\forall X (\neg P)$	$\neg \exists X (P)$	
13.	$\exists X (\neg P)$	$\neg \forall X (P)$	

These rules work on pieces of wffs as well, so for example given $\exists X (P \wedge Q)$ we can use rule 1 to infer $\exists X (Q)$.

Given these inference rules, we can prove that rule 12 is true in reverse as well:

$\neg \exists X (P)$	assume this
$\neg \exists X (\neg\neg P)$	by rule 9
$\neg\neg \forall X (\neg P)$	by rule 13
$\forall X (\neg P)$	by rule 8

9.3 Representing the Monopoly world

Let's look at some facts about the Monopoly world. First we need some facts about the layout of the board. These are always true and would be in the database to start with:

```
follow(britishairways,georgestreet)
follow(princesstreet,britishairways)
property(georgestreet)
property(princesstreet)
colour(georgestreet,blue)
airline(britishairways)
   . . .
```

At a certain stage of a game the database might include the following additional facts:

```
own(henry,georgestreet)
own(robin,princesstreet)
```

We also need some *meaning postulates* which relate the meanings of the different predicates. Here are the ones which relate the meaning of follow (used to describe the layout of the board) and the meaning of adjoin (used for the translation of "next to"):

$\forall X (\forall Y (follow(X,Y) \supset adjoin(X,Y)))$

$\forall X (\forall Y (follow(X,Y) \supset adjoin(Y,X)))$

Now we can answer the question "Does Henry own a property next to an airline?" using a *backward chaining* or *goal-directed* proof. The translation of this question is:

$\exists X (\exists Y (own(henry,X) \wedge property(X) \wedge airline(Y) \wedge adjoin(X,Y)))$

If we can prove this the answer to the question is "yes", otherwise "no". (If we have a WH-question, we require an instantiation for the variables which makes the formula provable.) The goal-directed proof goes like this:

$\exists X (\exists Y (own(henry,X) \wedge property(X) \wedge airline(Y) \wedge adjoin(X,Y)))$

$\Leftarrow$ (by rule 11)

$\exists Y (own(henry,georgestreet) \wedge property(georgestreet)$
$\wedge airline(Y) \wedge adjoin(georgestreet,Y))$

$\Leftarrow$ (by database and rule 2)

$\exists Y (property(georgestreet) \wedge airline(Y) \wedge adjoin(georgestreet,Y))$

$\Leftarrow$ (by database and rule 2)

$\exists Y (airline(Y) \wedge adjoin(georgestreet,Y))$

$\Leftarrow$ (by rule 11)

$airline(britishairways) \wedge adjoin(georgestreet,britishairways)$

$\Leftarrow$ (by database and rule 2)

$adjoin(georgestreet,britishairways)$ (*)

$\Leftarrow$ (by meaning postulate and rules 6 and 10)

$follow(georgestreet,britishairways)$

FAILSI

but (*) $\Leftarrow$ (by meaning postulate and rules 6 and 10)

$follow(britishairways,georgestreet)$

which is true according to the database, so the answer is "yes".

This works by starting with the thing we want to prove as a goal and working backwards using inference rules to things which are known to be true. At each step, we look around to see what rules can be applied to prove the currect goal. Once the goal has been reduced to facts which are known to be true, reading the successful path of the search backwards gives a proof.

9.4 Conclusion of Monopoly example

We considered three aspects of this problem:

1. Parsing the input into syntax trees
2. Translating the syntax trees into first-order predicate calculus
3. Using theorem proving to answer questions

The syntax was slightly beyond the edge of what can be handled adequately using RTN's and CFG's, as detailed already at the end of lecture 7.

In translating we found that in general it seemed to be possible to build an adequate representation of the meaning of a sentence using predicate calculus. We had to ignore certain complications and handled some aspects in a superficial way (e.g. pronouns) but we managed to build translations of sentences from the translations of their parts.

Once statements and questions were translated into predicate calculus, we encountered no problems in using proofs to handle question answering.

Some things we didn't cover:

- Choosing an efficient order of conjuncts in translating questions into predicate calculus; for example, "Is a utility unoccupied?" could be translated into either of the following:

  $\exists X (\neg \exists Y (occupy(Y,X)) \wedge utility(X))$

  $\exists X (utility(X) \wedge \neg \exists Y (occupy(Y,X)))$

The second will take less time to answer since it will involve looking at the utilities (there are two) and checking if one is unoccupied. The first will search through all the spaces to find the unoccupied ones (there are perhaps 20) and check if one is a utility.

- A statement like "Robin bought India Street from Henry" involves removing an assertion from the database (the one which says that Henry owns India Street) as well as adding one.
- More generally, we didn't take proper account of time; the database contains only a snapshot of what is true at a particular time, with no information about the history of events. This means we can't handle questions like "Who owned Dalry Road before Robin?" or "What did Janet get for her two airlines?"

Lecture X

Introduction to augmented transition networks

10.1 What is wrong with RTN's?

We have tried using RTN's to produce parsers for two examples: a voice-controlled calculator and a Monopoly question-answering system. The language involved in the calculator example was simple enough that it was not too difficult to produce an RTN which would handle it successfully. But although we managed to produce an RTN which would handle the language involved in the Monopoly example with reasonable success, we ran into some problems. The main ones were:

- We had to extend the RTN parsing mechanism to handle WH-questions like "What properties does Seymour own?"; we added a limited memory to allow something ("what properties") to be put to one side and then used later (as the object of "own").
- Our RTN would parse lots of sentences which really shouldn't be accepted, for example "Does Robin buy Dalry Road to blue airline?"

Looking at ordinary English, it is hard to produce an RTN which will handle the following things:

Number agreement: "Those student" and "a books" are wrong, while "these books" is okay.

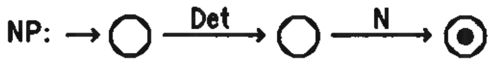
Subject-verb agreement: "He walk" and "they walks" are wrong, while "he walks" is okay.

Same structure for different sentences: The following sentences should ideally be given the same structure:

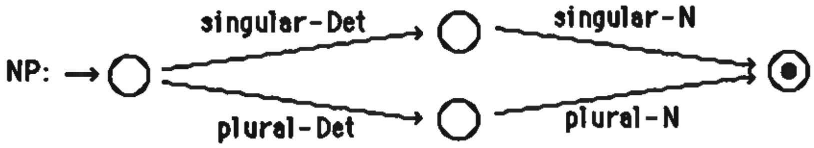
Henry gave the book to Sally.
Henry gave Sally the book.
The book was given to Sally by Henry.
Sally was given the book by Henry.

Why is this so hard? For number agreement, the problem is that the lexical categories Noun and Det group together different kinds of nouns/determiners (some are singular, some plural). There is no way of distinguishing between the singular and plural nouns/determiners and so both must be treated the same. This is also the reason why handling subject-verb agreement with an RTN is difficult. Producing the same structure for the sentences "Henry gave the book to Sally" and "The book was given to Sally by Henry" is impossible without adding memory to the RTN as in the Monopoly example since the RTN parsing algorithm retains the order of the words in a sentence and the three NP's "Henry", "Sally" and "the book" come in a different order in the four sentences.

It is possible to make things work to some extent by brute force. For number agreement, we could divide the lexical category Noun into Singular-Noun and Plural-Noun and divide the category Det into Singular-Det and Plural-Det. Note that some words would be in more than one category, for example "sheep" and "the". Then the following simple RTN for NP's:



would be replaced by the following more complicated RTN:



A similar idea would work for subject-verb agreement (the categories would be things like 1st-Person-Singular-Verb). In order to get the same structure for different forms of the same sentence, we would need to apply some kind of post-processing to convert different equivalent forms into the same form.

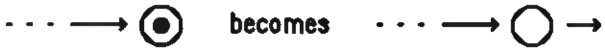
All this is feasible, but the collection of lexical categories and the resulting RTN get very complicated.

10.2 Augmenting the network

To handle these things in a nicer way, we can expand the power of RTN's by augmenting them to give ATN's, i.e. *augmented (recursive) transition nets.* We augment an RTN by adding *conditions* and *actions* to the arcs of the network.

- the conditions restrict the circumstances under which an arc may be taken;
- the actions perform structure-building and take notes for use by later actions/conditions.

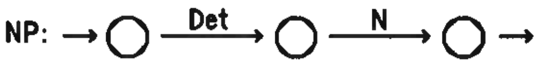
Also, instead of final nodes we use *exit arcs* which look like this:



This is to allow conditions and actions to be associated with exit from the network just like with any other arc.

10.3 Example: Number agreement in noun phrases

A simple RTN for NP is the following:



To avoid "those student" and "a books", we add the following conditions and actions:

Det action: set *number* to Number feature of Det
N condition: *number* must be the same as Number feature of N
Exit action: set Number feature of NP to *number*

Note that this gives two parses to the NP "the fish".

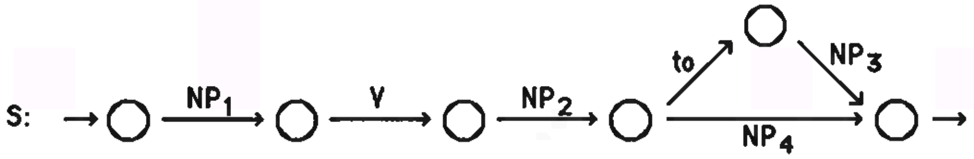
In building this ATN, two assumptions have been made:

- Each word is associated with various *features* in the dictionary, such as Number and Tense (for verbs). Syntactic structures can be given features too; we set the Number feature in the NP produced by the above ATN in order to allow an ATN for S which uses this one for NP to handle subject-verb agreement.
- There are *registers* available for temporary storage.

10.4 Building parse trees

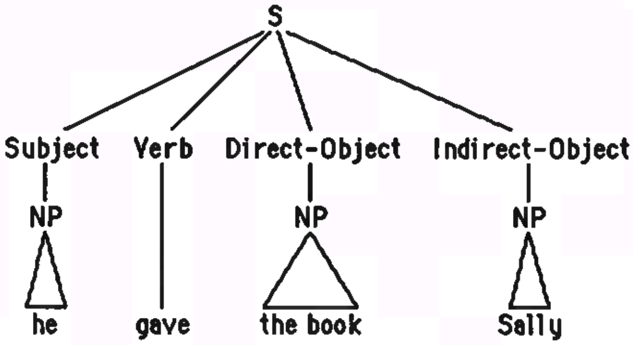
Actions will be used to build parse trees rather than leaving this to the parsing algorithm as with RTN's. For example, consider sentences like "He gave the book to Sally" and "He gave Sally the book"; as mentioned before, we would like these to get the same syntactic structure.

Here is an ATN which will parse sentences of this form and which handles subject-verb agreement:



NP_1 action: set Subject to NP_1; set *number* to Number feature of NP_1
V condition: *number* must be the same as Number feature of V
V action: set Verb to V
NP_2 action: set *object* to NP_2
NP_3 action: set Indirect-Object to NP_3; set Direct-Object to *object*
NP_4 action: set Indirect-Object to *object*; set Direct-Object to NP_4

So both "He gave the book to Sally" and "He gave Sally the book" get the following structure:



The parsing algorithm will only form the nodes of the syntax tree; the actions on the arcs are responsible for hanging things from the nodes. The things which "hang off" (Direct-Object, Verb, etc.) are treated more or less like *features* (like Number in the NP example above).

10.5 How does parsing work now?

The parsing algorithm is comparatively simple now, since much of the work is done by the actions on the arcs.

To parse a string from a sub-net:

1. Start at the initial state of the sub-net. Form a node labelled by the name of the sub-net.
2. Choose an exit arc from the current state or else an arc from the current state to another state labelled by:
   - the empty string,
   - the next word of the string, or
   - the name of a sub-net

   and whose condition (if any) is satisfied. If there is no such arc, the parse fails.

3a. If the chosen arc is an exit arc, then if we are at the end of the string perform the action on the arc (if any) and the parse succeeds; otherwise it fails.

3b. If the chosen arc is labelled by the empty string or by a word, perform the action on the arc (if any).

3c. If the chosen arc is labelled by a sub-net name, divide the string in two and try to parse the first half using the sub-net. Then perform the action on the arc (if any).

4. Follow the arc chosen in step 2 to the state it points to.
5. Go to step 2 to parse the rest of the string.

Try parsing the sentence "He gave the book to her" using the ATN above.

10.6 Power of ATN's

ATN's are a lot more powerful than RTN's. In fact, if we don't put strong restrictions on the actions and conditions which can be associated with arcs, ATN's have the same power as general-purpose computers! To see why, try constructing an ATN which adds two numbers (i.e. which will "parse" any string consisting of two sequences of binary digits separated by a comma, producing a "parse tree" containing their sum as its result).

Unfortunately, in passing from RTN's to ATN's we have lost the ability to do both parsing and generation from the same net. Generation is not possible now because the actions are for parsing only and cannot in general be reversed as would be necessary for generation. Dually, *transformational grammar* (ask a Linguistics student to explain what it is if you are interested) can be used to generate but is not very useful for parsing for the same reason. Transformational grammar is (sort of) related to ATN's in the same way as context-free grammar is related to RTN's.

Lecture XI

Monopoly example: using augmented transition networks

11.1 Introduction

In the last lecture it was argued that RTN's aren't quite powerful enough to handle certain phenomena in English such as subject-verb agreement. Recall the problems we encountered in the Monopoly example when we tried to use an RTN for parsing: we had to extend the RTN parsing mechanism with a limited memory to handle WH-questions, and the RTN we produced would parse lots of ungrammatical sentences as well as grammatical ones.

In this lecture we will construct an ATN to replace the RTN of the Monopoly example, which will handle WH-questions properly and which will reject some ungrammatical sentences which the RTN accepted. It would be possible to reject all ungrammatical sentences, but we are only going to concentrate on a few constructions to show how it can be done.

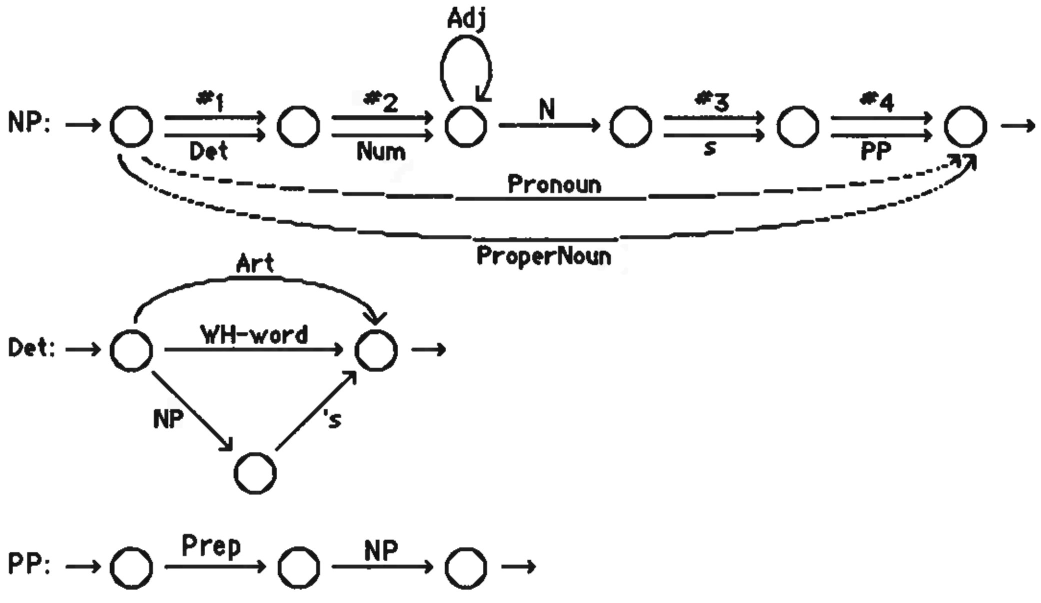
We will only look at the problem of parsing; since the parse trees our ATN will produce will be the same as those produced by the RTN before, the other components (translation into predicate calculus and theorem proving) are the same as before.

11.2 Noun phrases revisited

Recall the kind of noun phrases we have to handle:

Proper names: Robin, Abercromby Place
Nouns with determiners: an airline, the owner
Numbers as modifiers: two airlines, two hundred pounds
Possessive phrases: the owner of Royal Circus, Seymour's monopoly
Colours as modifiers: the green monopoly
WH-words as modifiers: what properties, how many airlines
Pronouns: he, her airlines

Here is the RTN which we produced to handle these:



This RTN will parse the following grammatical strings:
an airline, airlines, twenty-seven airlines, those twenty-seven airlines

as well as the similar ungrammatical strings:

an airlines, airline, a twenty-seven airlines, twenty-seven airline

In grammatical strings, the Det (if present) can be either Singular or Plural. If it is not present, the NP is Plural. If it is Singular, then no Num is permitted (actually "one" is permitted, but we will pretend it isn't). On the other hand, the WH-word (which is one kind of Det) "how many" is plural but does not take a Num; note that "how many airlines" is grammatical while "how many airline" and "how many twenty-seven airlines" are not grammatical.

To handle this, we can add conditions and actions as follows:

Det action: set *number* to Number feature of Det
$\#_1$ action: set number to Plural
Num condition: *number* must be Plural
s condition: *number* must be Plural or HowMany
$\#_3$ condition: *number* must be Singular
Pronoun action: set *number* to Number feature of Pronoun
ProperNoun action: set *number* to Number feature of ProperNoun
Exit action: set Number feature of NP to Plural if *number* is HowMany; otherwise set Number feature of NP to *number*

Plural nouns are treated here by assuming a separate morphological pre-processing phase which converts e.g. "airlines" to "airline s". Thus the condition is on the "s" rather than on the N. It would also be possible to make Number a feature of N and forget the pre-processing.

To get the above ATN for NP to work, appropriate actions have to be added to the Det sub-net to give each Det a Number feature. Some Det's are both Singular and Plural, such as "his". This is the case with possessive Det's like "the player's". We add conditions and actions as follows:

Art action: set Number feature of Det to Number feature of Art
WH-word action: set Number feature of Det to Number feature of WH-word
's action: set Number feature of Det to either Singular or Plural

Other things can go wrong in NP's. For example, only some things can be coloured: "a blue property" is okay but "a blue airline" is not. This could be caught either during parsing (relatively easy since the only adjectives we have in this domain are colours) or during translation. Also, certain PP's cannot appear after certain N's: "the owner of the property" and "the airline after Royal Circus" are okay but "the airline of the property" and "the airline after Henry" are not. Similarly, some possessives cannot be combined with certain N's: "the airline's owner" and "Henry's property" are okay but "the airline's property" and "Henry's owner" are not.

These could all be handled without much trouble, but we won't do them here. Also, our ATN needs actions to build the parse tree, but it isn't difficult to see what these actions would be and where they would go.

11.3 Statements and yes/no questions revisited

There are several things which can go wrong in a statement or yes/no question. For example, the grammar we produced originally would accept sentences with strange VG's such as: "Henry did did own Minto Street". This is just one example of a wrong way of combining verbs and auxiliaries; there are many others. Verbs may optionally take certain prepositions, so for example "Henry bought Minto street from Janet" is okay but "Henry bought Minto street to Janet" and "Henry owned Minto street from Janet" are not. However, subject-verb agreement is not a problem since all subjects are singular in this domain (only objects may be plural).

We won't treat any of these problems here; we also won't extend the grammar to handle passives such as:

Dalry Road was bought from Leslie by Robin.
Was Dalry Road bought by Robin?

All of these things can be handled with ATN's.

11.4 WH-questions revisited

Recall that handling WH-questions required the ability to keep sections of parse trees (NP's and PP's) to one side until a gap appeared later in the sentence where the saved bit could be inserted. This was because of the observation that WH-questions are just WH-words followed by a yes/no question with a hole:

What did Leslie sell? ⇒ Did Leslie sell *X* ?

What was sold Robin by Leslie? ⇒ Was *X* sold Robin by Leslie?

What did Leslie sell to Robin? ⇒ Did Leslie sell *X* to Robin?

To whom did Leslie sell Dalry Road? ⇒ Did Leslie sell Dalry Road to *X* ?

Whom did Leslie sell Dalry Road to? ⇒ Did Leslie sell Dalry Road to *X* ?

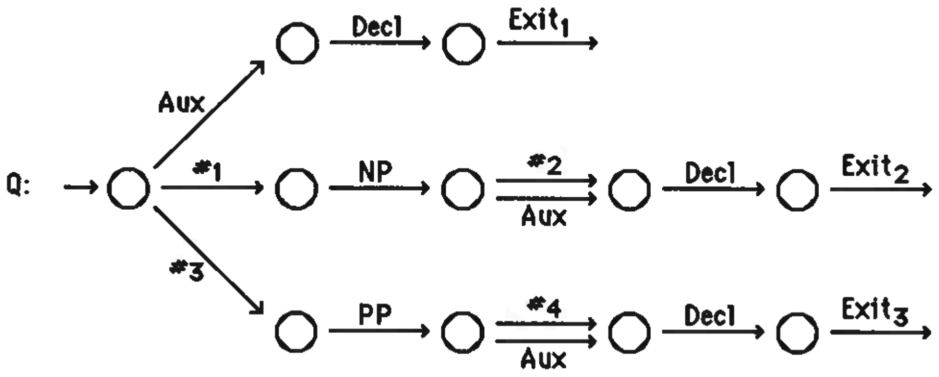
We came up with the following CFG rules to handle questions of this kind:

Q → NP! [Aux] Decl!NP

Q → PP! [Aux] Decl!PP

where NP! means that the NP must be a WH-NP (either a WH-Pronoun like "who" or "what", or an NP with a WH-word as determiner), and that it should be kept to one side rather than used. Decl!NP means that in looking for a Decl this stored NP can be used when an NP is found to be missing in the string. PP! and Decl!PP are the analogous things for PP's.

We can do the same kind of thing now with an ATN as follows:



The actions and conditions on the arcs are as follows:

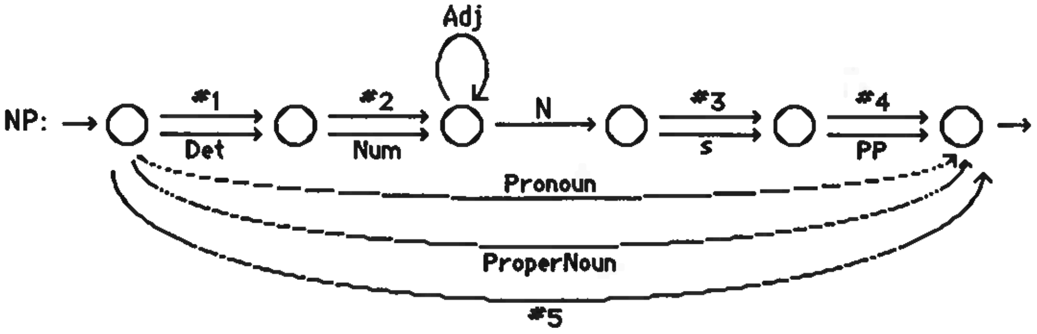
$\#_1$ action: set *WH-NP* to empty

NP condition: NP is a WH-pronoun or has a Det which is a WH-word
NP action: set *WH-NP* to NP
Exit$_2$ condition: *WH-NP* is empty

$\#_3$ action: set *WH-PP* to empty

PP condition: the NP of PP is a WH-pronoun or has a Det which is a WH-word
PP action: set *WH-PP* to PP
Exit$_3$ condition: *WH-PP* is empty

To make this work, the ATN's for NP and PP have to be changed. For example, here is the new ATN for NP:



The actions and conditions are as before, except for the new arc:

$\#_5$ condition: *WH-NP* is not empty

$\#_5$ condition: set NP to *WH-NP* and set *WH-NP* to empty

Note that *WH-NP* and *WH-PP* have to be registers which are *global* to the whole ATN since they are used by more than one sub-net.

Lecture XII

An alternative to the predicate calculus for semantic representation

12.1 Introduction

We have been using predicate calculus to represent the meaning of sentences. This works reasonably well for certain domains, as we have seen with the Monopoly example. There are other possibilities, though. This section will discuss an alternative representation called *conceptual dependency* notation, invented by Schank who works at Yale.

We have already seen an application of conceptual dependency in lecture 2, namely the SAM story-understanding system which was built in about 1977. The main idea behind SAM is actually not conceptual dependency notation but the idea of a *script*. Each script represents some situation from everyday life which everybody is familiar with and which often involves a stereotyped set of events. The story in lecture 2 involves three scripts: taking a bus/subway, going to a restaurant, and being robbed.

But conceptual dependency is used for semantic representation in SAM and this is important in that it is not clear that predicate calculus would be adequate for this purpose.

12.2 Problems with predicate calculus

Predicate calculus is quite good for expressing the meaning of some sentences. For example:

All green properties are owned by Robin.

$\forall$ X ((property(X) $\wedge$ colour(X,green)) $\supset$ own(X,robin))

All properties are orange or green.

$\forall$ X (property(X) $\supset$ (colour(X,orange) $\vee$ colour(X,green)))

Royal Circus is a property.

property(royalcircus)

Royal Circus is not orange.

$\neg$ colour(royalcircus,orange)

Robin owns a property.

$\exists$ X (property(X) $\wedge$ own(robin,X))

Another big advantage of predicate calculus is that answering questions reduces to the problem of proving theorems which is not always easy but at least is a familiar problem.

But consider a sentence like "John threatened Fred with a broken nose". How can that be represented in predicate calculus? We can't do much better than introduce a predicate called "threaten" where threaten(X,Y,Z) means that X (a person) threatens Y (a person) with Z (a consequence), in which case the sentence translates to:

threaten(John,Fred,brokennose)

This is not very useful for answering questions, particularly if the sentence occurs in the middle of a story like the following:

> Fred was fond of telling offensive jokes about midgets. John threatened Fred with a broken nose. Fred asked if he was able to reach so high. Suddenly Mary heard a loud cracking noise and a scream of pain.

How do we answer a question like "Who screamed?"

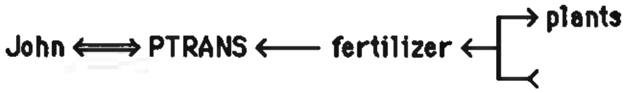
Actually, these problems were already present before in the sense that our predicate calculus representation for the Monopoly world didn't say anything very deep about the meaning of words like "own" and "sell". These words are simple ways to represent complicated processes. For example, in the real world if somebody says "Robin bought British Airways from Henry" it means much more than the fact that Henry used to own British Airways and now Robin does. One reason why predicate calculus was adequate for the Monopoly example is because in the Monopoly world it doesn't mean anything more than this.

12.3 Conceptual dependency notation

Note: It is not intended that you learn how to represent sentences in conceptual dependency notation (in contrast to predicate calculus!); this lecture is just supposed to give you a general idea of how it looks.

Schank claims that conceptual dependency notation is capable of describing the entire range of everyday human actions (although the examples Schank uses invariably involve either violence or food) using about a dozen *primitive acts*. These are:

PTRANS: A physical object is moved from one place to another. For example, "John put fertilizer on the plants" would be represented as follows:

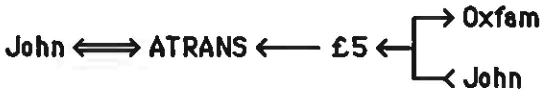


MOVE: A body part is moved to a place. This is different from PTRANS because it requires only an act of will. MOVE would be used in the representation of "John kicked Frank" since part of the meaning of "kick" is that John moved his foot.

PROPEL: An object is moved by direct application of force, as in throwing or shooting.

GRASP: An object is grasped by a person.

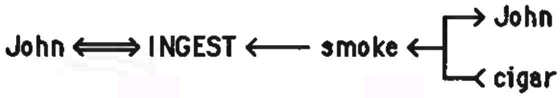
ATRANS: Control (possession) of something is transferred from one owner to another. For example, "John donated £5 to Oxfam" would be represented as follows:



Note that "John gave 50p to the beggar" would involve both an ATRANS and a PTRANS (of the coin itself).

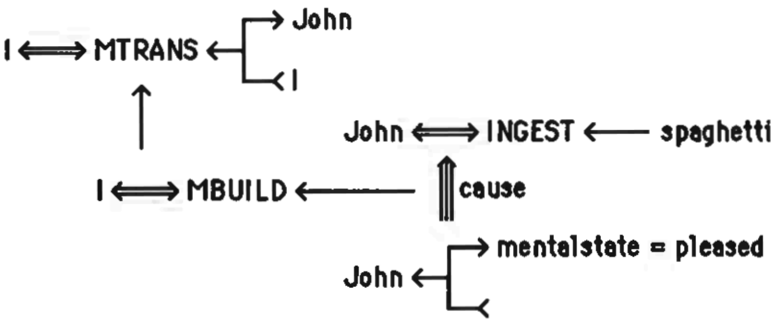
EXPEL: Sweating, exhaling, etc.

INGEST: Eating, drinking, inhaling etc. For example, "John smoked a cigar" would be represented as follows:



MTRANS: Information is transferred from one "place" to another, e.g. from one person's mind to another (as in the verb "tell") or from a person's long-term memory to his conscious mind (as in the verb "remember").

MBUILD: A person comes to believe something (a conceptualization) is true. For example, "I advised John to try the spaghetti" would involve an INGEST, an MTRANS (I advised John) and an MBUILD (I think John would like eating the spaghetti):

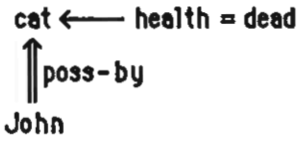


SPEAK: A person makes a sound.
ATTEND: A person directs a sense to an object or event (listens to, looks at, etc.).

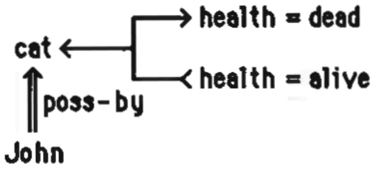
There are also various kinds of relations between events; we saw some already in "I advised John to try the spaghetti". For example, one event or conceptualization can *cause*, *prevent* or *enable* another event. One event can *precede* another event or be a *sub-event* of another.

Some acts (like PTRANS) take an object, a source and a destination. Some (like MOVE) involve an object and a destination but no source. Some (like MBUILD) take an event, and some (MTRANS) take an event, a source and a destination.

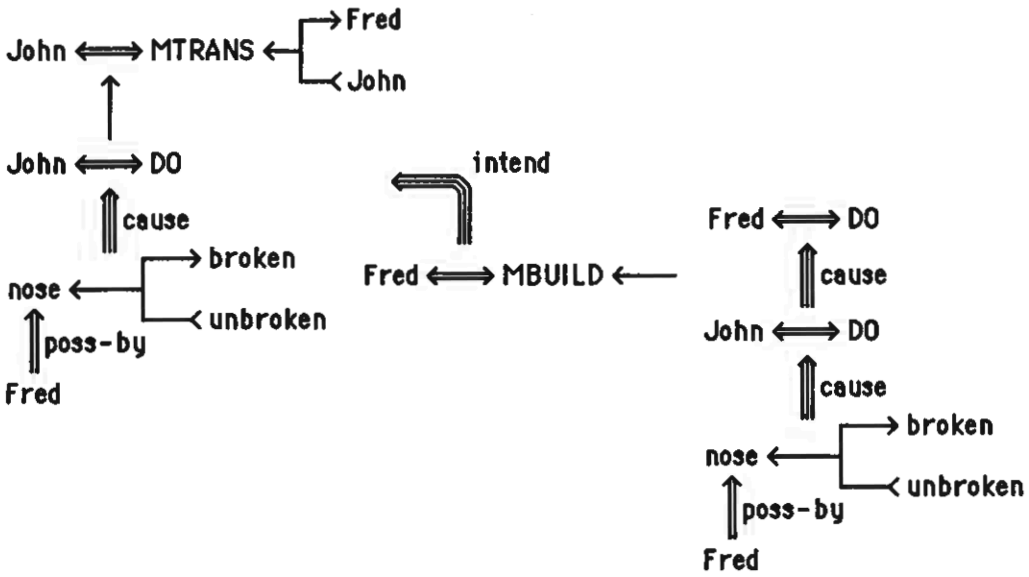
There are adjectives which are regarded as describing the *state* of an object, and possessive relations between people and things. For example, here is the representation of "John's cat was dead":



States of objects can change in the course of an event; compare the above diagram with the following representation of "John's cat died":



Finally, here is the conceptual dependency representation of "John threatened Fred with a broken nose":



In other words, John communicated to Fred the information that he will do something to break Fred's nose, which was intended to make Fred believe that if he does some particular thing it will cause John to break his nose. This uses the primitive acts MBUILD and MTRANS. It also uses DO, which is really something like a placeholder for an unspecified primitive act and not an act itself.

12.4 Advantages and disadvantages

Although we haven't gone into aspects of conceptual dependency like how to answer questions about a story, it seems clear that most of the answers to questions which might be asked are recorded directly in the diagrams. For example, the representation of the sentence "John's cat died" includes the previous state of the cat (alive) and the current state (dead); all we would need is some general rule about state changes to figure out the answer to the question "Is the cat dead?"

Although the diagrams contain words like "dead", these are really references to dictionary entries which contain information about the meanings of words. So presumably there would be enough information in the dictionary entry for "dead" to enable us to answer a question like "Will John's cat be available for chasing mice tomorrow?"

On the other hand, it is possible to argue that the notation is unwieldy. Also, although conceptual dependency notation is more successful than predicate calculus in capturing the meaning of sentences like "John threatened Fred with a broken nose", it is not so good for representing sentences like "Everything in Antarctica is cold" which can be expressed in a natural way using predicate calculus.

For more information about conceptual dependency notation, see *Introduction to Artificial Intelligence* by E. Charniak and D. McDermott, pp. 325-333 (but note that they treat conceptual dependency much more formally than we have done and they use a LISP-like notation instead of the diagrammatic notation Schank uses).