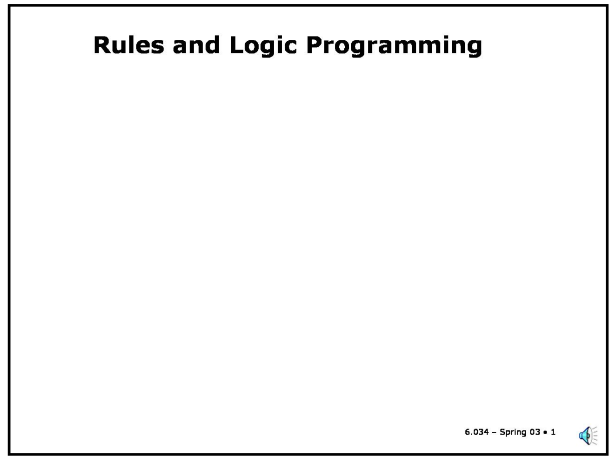
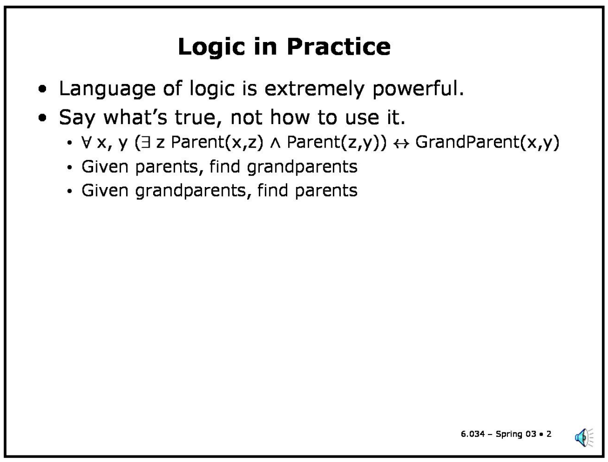
6.034 Notes: Section 9.1

Slide 9.1.1
We've now spent a fair bit of time learning about the language of first-order logic and the mechanisms of automatic inference. And, we've also found that (a) it is quite difficult to write first-order logic and (b) quite expensive to do inference. Both of these conclusions are well justified. Therefore, you may be wondering why we spent the time on logic.

We can motivate our study of logic in a variety of ways. For one, it is the intellectual foundation for all other ways of representing knowledge about the world. As we have already seen, the Web Consortium has adopted a logical language for its Semantic Web project. We also saw that airlines use a language not unlike FOL to describe fare restrictions. We will see later when we talk about natural language understanding that logic also plays a key role.

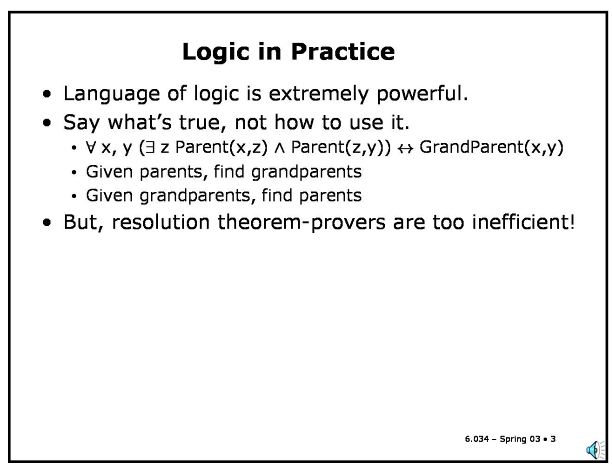
There is another practical application of logic that is reasonably widespread namely **logic programming**. In this section, we will look briefly at logic programming. Later, when we study natural language understanding, we will build on these ideas.

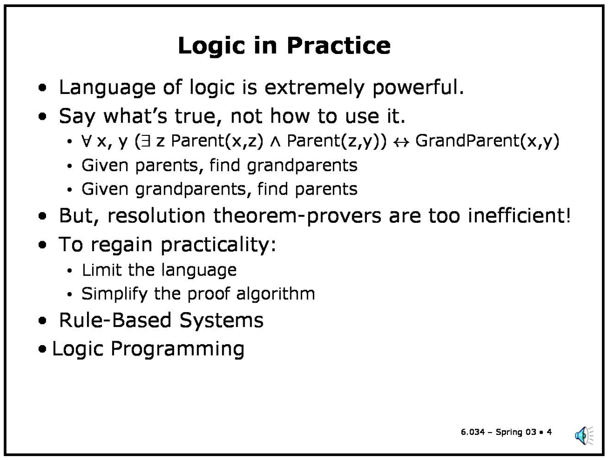




Slide 9.1.2
We have seen that the language of logic is extremely general, with much of the power of natural language. One of the key characteristics of logic, as opposed to programming languages but like natural languages, is that in logic you write down what's true about the world, without saying how to use it. So, for example, one can characterize the relationship between parents and grandparents in this sentence without giving an algorithm for finding the grandparents from the grandchildren or a different algorithm for finding the grandchildren given the grandparents.

Slide 9.1.3
However, this very power and lack of specificity about algorithms means that the general methods for performing computations on logical representations (for example, resolution refutation) are hopelessly inefficient for most practical problems.



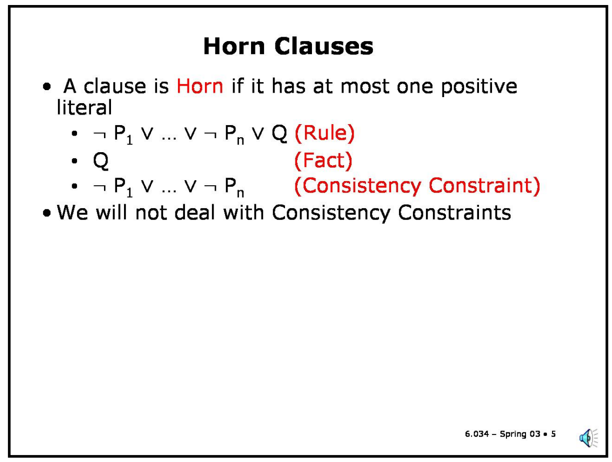


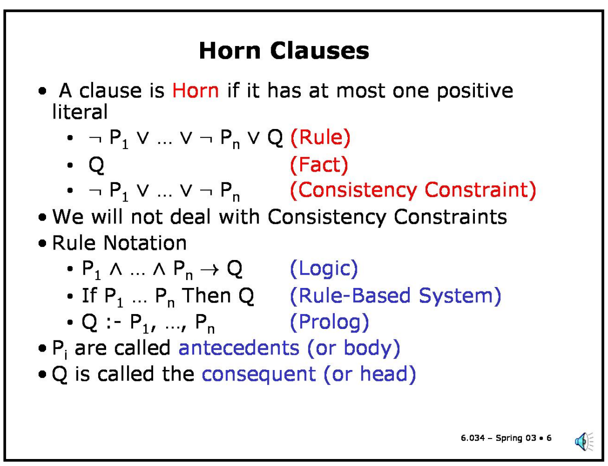
Slide 9.1.4
There are, however, approaches to regaining some of the efficiency while keeping much of the power of the representation. These approaches involve both limiting the language as well as simplifying the inference algorithms to make them more predictable. Similar ideas underlie both **logic programming** and **rule-based systems**. We will bias our presentation towards logic programming.

Slide 9.1.5
In logic programming we will also use the clausal representation that we derived for resolution refutation. However, we will limit the type of clauses that we will consider to the class called **Horn clauses**. A clause is Horn if it has at most one positive literal. In the examples below, we show literals without variables, but the discussion applies both to propositional and first order logic.

There are three cases of Horn clauses:

- A **rule** is a clause with one or more negative literals and exactly one positive literal. You can see that this is the clause form of an implication of the form `P1 ^ ... ^ Pn -> Q`, that is, the conjuction of the P's implies Q.
- A **fact** is a clause with exactly one positive literal and no negative literals. We generally will distinguish the case of a **ground fact**, that is, a literal with no variables, from the general case of a literal with variables, which is more like an unconditional rule than what one would think of as a "fact".
- In general, there is another case, known as a **consistency constraint** when the clause has no positive literals. We will not deal with these further, except for the special case of a **conjunctive goal clause** which will take this form (the negation of a conjuction of literals is a Horn clause with no positive literal). However, goal clauses are not rules.





Slide 9.1.6
There are many notations that are in common use for Horn clauses. We could write them in standard logical notation, either as clauses, or as implications. In rule-based systems, one usually has some form of equivalent "If-Then" syntax for the rules. In Prolog, which is the most popular logic programming language, the clauses are written as a sort of reverse implication with the ":-" instead of "<-".

We will call the Q (positive) literal the **consequent** of a rule and call the P_i (negative) literals the **antecedents**. This is terminology for implications borrowed from logic. In Prolog it is more common to call Q the **head** of the clause and to call the P literals the **body** of the clause.

Slide 9.1.7

Note that not every logical statement can be written in Horn clause form, especially if we disallow clauses with zero positive literals (consistency constraints). Importantly, one cannot have a negation on the right hand side of an implication. That is, we cannot have rules that conclude that something is not true! This is a reasonably profound limitation in general but we can work around it in many useful situations, which we will discuss later. Note that because we are not dealing with consistency constraints (all negative literals) we will not be able to deal with negative facts either.

Limitations

- Cannot conclude negation
  - $P \rightarrow \neg Q$
  - $\neg P \vee \neg Q$: Consistency constraint
  - $\neg P$: Consistency constraint

6.034 – Spring 03 • 7

Limitations

- Cannot conclude negation
  - $P \rightarrow \neg Q$
  - $\neg P \vee \neg Q$: Consistency constraint
  - $\neg P$: Consistency constraint
- Cannot conclude (or assert) disjunction
  - $P_1 \wedge P_2 \rightarrow Q_1 \vee Q_2$
  - $Q_1 \vee Q_2$
  - These are not Horn

6.034 – Spring 03 • 8

Slide 9.1.8

Similarly, if we have a disjuction on the right hand side of an implication, the resulting clause is not Horn. In fact, we cannot assert a disjunction with more than one positive literal or a disjuction of all negative literals. The former is not Horn while the latter is a consistency constraint.

Slide 9.1.9

It turns out that given our simplified language, we can use a simplified procedure for inference, called **backchaining**, which is basically a generalized form of Modus Ponens (one of the "natural deduction" rules we saw earlier).

Backchaining is relatively simple to understand given that you've seen how resolution works. We start with a literal to "prove", which we call C. We will also use Green's trick (as in Chapter 6.3) to keep track of any variable bindings in C during the proof.

We will keep a stack (first in, last out) of goals to be proved. We initialize the stack to have C (first) followed by the Answer literal (which we write as Ans).

Inference: Backchaining

- To "prove" a literal C
  - Push C and an Ans literal on a stack

6.034 – Spring 03 • 9

Inference: Backchaining

- To "prove" a literal C
  - Push C and an Ans literal on a stack
  - Repeat until stack only has Ans literal or no actions available.
    - Pop literal L off of stack

6.034 – Spring 03 • 10

Slide 9.1.10

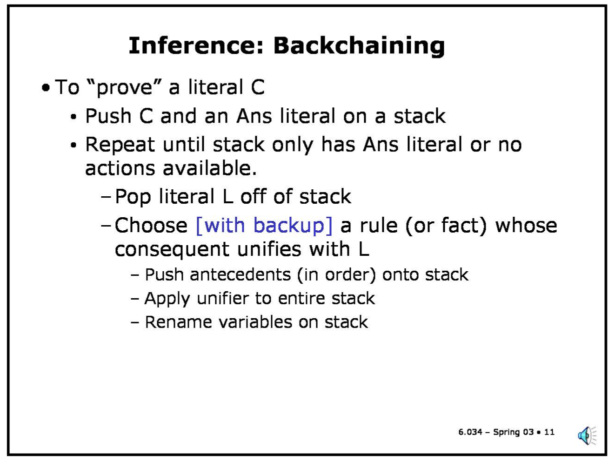
The basic loop is to pop a literal (L) off the stack until either (a) only the Ans literal remains or (b) there are no further actions possible. The first case corresponds to a successful proof; the second case represents a failed proof.

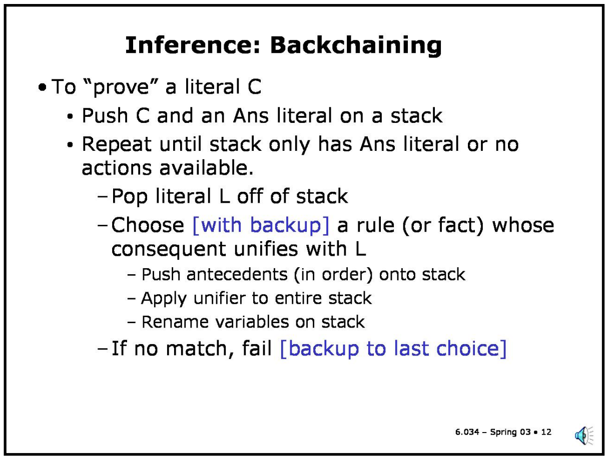
A word of warning. This loop does not necessarily terminate. We will see examples later where simple sets of rules lead to infinite loops.

Slide 9.1.11

Given a literal L, we look for a fact that unifies with L or a rule whose consequent (head) unifies with L. If we find a match, we push the antecedent literals (if any) onto the stack, apply the unifier to the entire stack and then rename all the variables to make sure that there are no variable conflicts in the future. There are other ways of dealing with the renaming but this one will work.

In general, there will be more than one fact or rule that could match L; we will pick one now but be prepared to come back to try another one if the proof doesn't work out. More on this later.





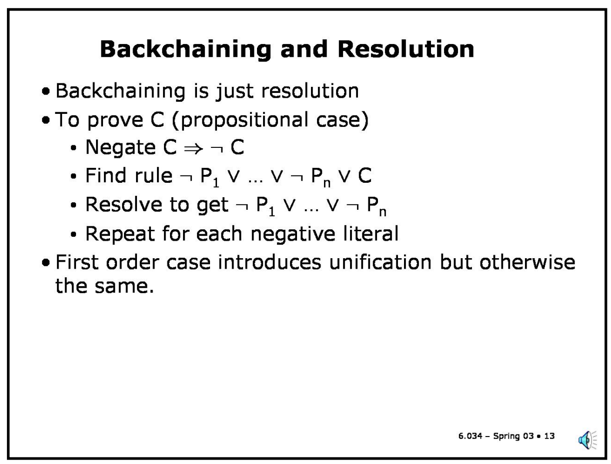
Slide 9.1.12

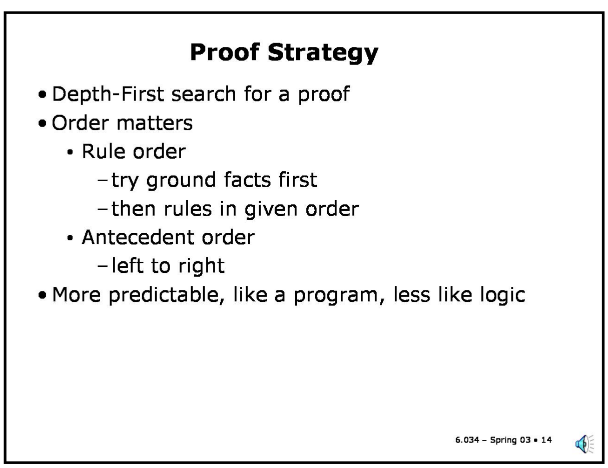
If no match can be found for L, we fail and backup to try the last choice that has other pending matches.

Slide 9.1.13

If you think about it, you'll notice that backchaining is just our familiar friend, resolution. The stack of goals can be seen as negative literals, starting with the negated goal. We don't actually show literals on the stack with explicit negation but they are implicitly negated.

At every point, we pair up a negative literal from the stack with a positive literal (the consequent) from a fact or rule and add the remaining negative literals (the antecedents) to the stack.



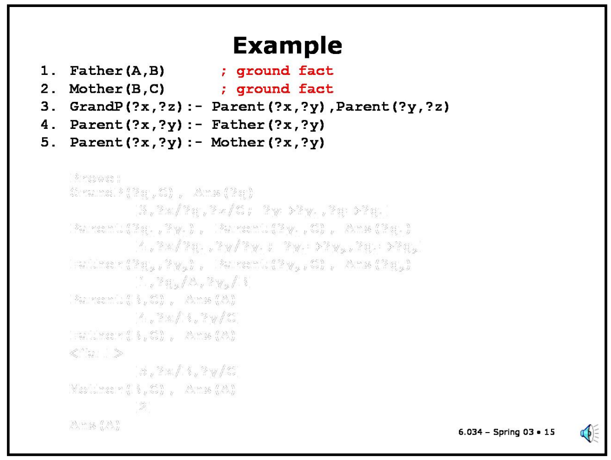


Slide 9.1.14

When we specified backchaining we did it with a particular search algorithm (using the stack), which is basically depth-first search. Furthermore, we will assume that the facts and rules are examined in the order in which they occur in the program. Also that literals from the body of a rule are pushed onto the stack in reverse order, os that the one that occurs first in the body will be the first popped off the stack.

Given these ordering restrictions, it is much easier to understand what a logic program will do. On the other hand, one must understand that what it will do is not what a general theorem prover would do with the same rules and facts.

Slide 9.1.15



Time for an example. Let's look at the following database of facts and rules. The first two entries are ground facts, that A is Father of B and B is Mother of C. The third entry defines a grandparent rule that we would write in FOL as:

```
@x . @y. @z. P(x,y) ^ P(y,z) -> GrandP(x,z)
```

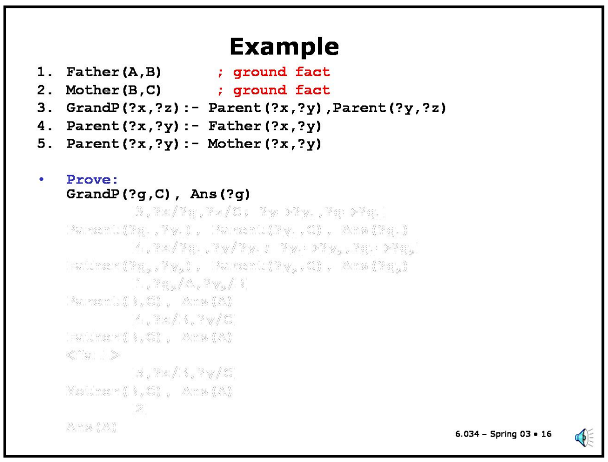
Our rule is simply this rule written with the implication pointing "backwards". Also, our rule language does not have quantifiers; all the variables are implicitly universally quantified.

In our rule language, we will modify our notational conventions for FOL. Instead of identifying constants by prefixing them with $, we will indicate variables by prefixing them with ?. The rationale for this is that in our logic examples we had lots more variables than constants, but that will be different in many of our logic-programming examples.

The next two rules specify that a Father is a Parent and a Mother is a parent. In usual FOL notation, these would be:

```
@x . @y. F(x,y) -> P(x,y)
@x . @y. M(x,y) -> P(x,y)
```

Slide 9.1.16



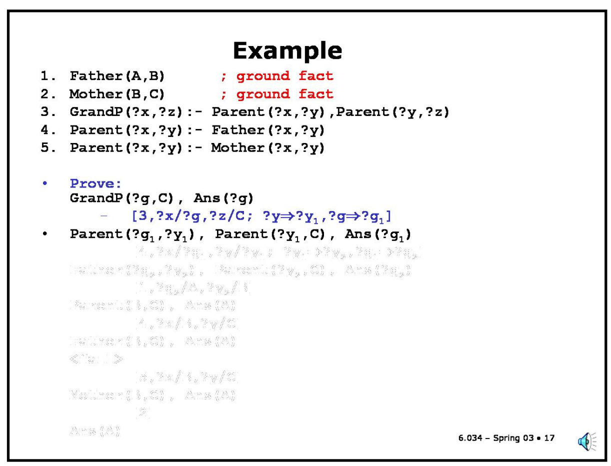
Now, we set out to find the Grandparent of C. With resolution refutation, we would set out to derive a contradiction from the negation of the goal:

```
~ ]g . GrandP(g,C)
```

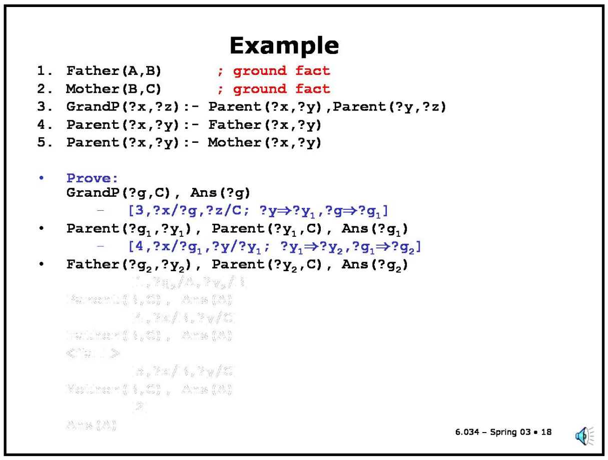
whose clause form is `~GrandP(g,C)`. The list of literals in our goal stack are implicitly negated, so we start with `GrandP(g,C)` on the stack. We have also added the Ans literal with the variable we are interested in, ?g, hopefully the name of the grandparent.

Now, we set out to find a fact or rule consequent literal in the database that matches our goal literal.

Slide 9.1.17



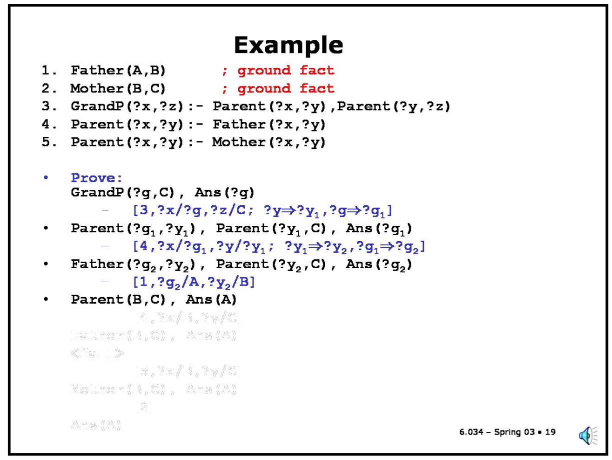
You can see that the grandparent goal literal unifies with the consequent of rule 3 using the unifer `{ ?x/?g, ?z/C }`. So, we push the antecedents of rule 3 onto the stack, apply the unifier and then rename all the remaining variables, as indicated. The resulting goal stack now has two Parent literals and the Ans literal. We proceed as before by popping the stack and trying to unify with the first Parent literal.

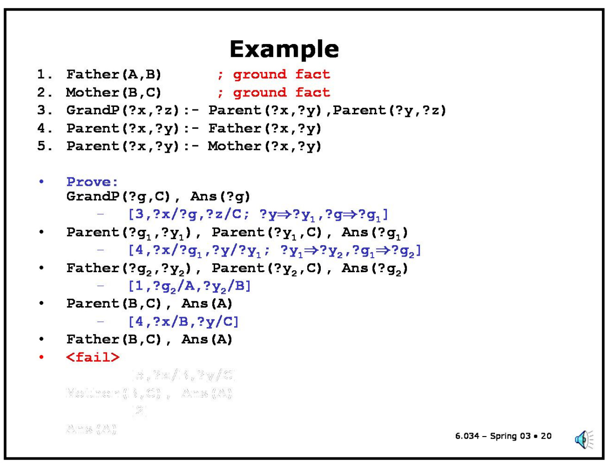


Slide 9.1.18
The first Parent goal literal unifies with the consequent of rule 4 with the unifier shown. The antecedent (the `Father` literal) is pushed on the stack, the unifier is applied and the variables are renamed.

Note that there are two Parent rules, we use the first one but we have the other one available should we fail with this one.

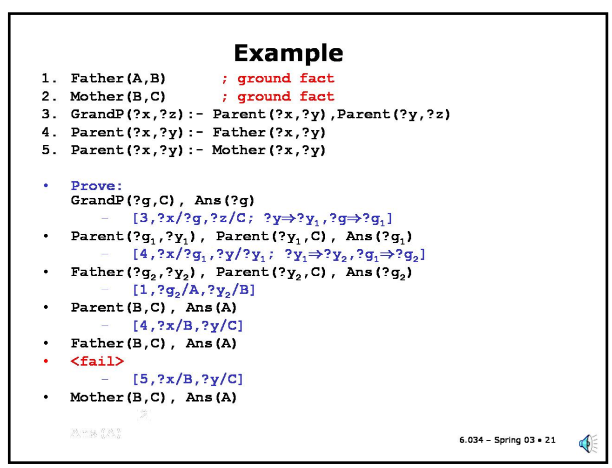
Slide 9.1.19
The `Father` goal literal matches the first fact, which now unifies the ?g variable to A and the ?y variable to B. Note that since we matched a fact, there are no antecedents to push on the stack (as in resolution with a unit-length clause). We apply the unifier, rename and proceed.

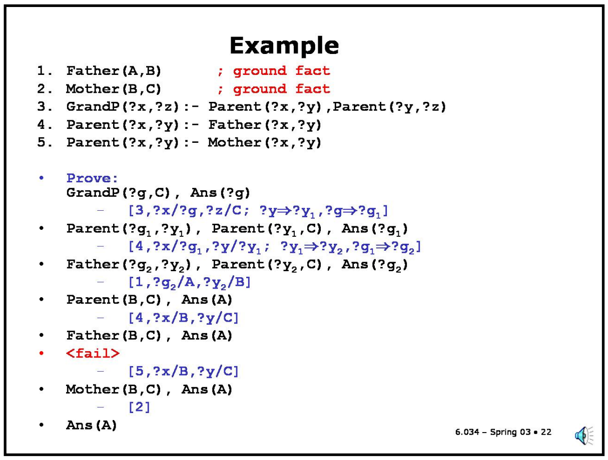




Slide 9.1.20
Now, we can match the Parent(B,C) goal literal to the consequent of rule 4 and get a new goal (after applying the substitution to the antecedent), `Father(B,C)`. However we can see that this will not match anything in the database and we get a failure.

Slide 9.1.21
The last choice we made that has a pending alternative is when we matched Parent(B,C) to the consequent of rule 4. If we instead match the consequent of rule 5, we get an alternative literal to try, namely `Mother(B,C)`.





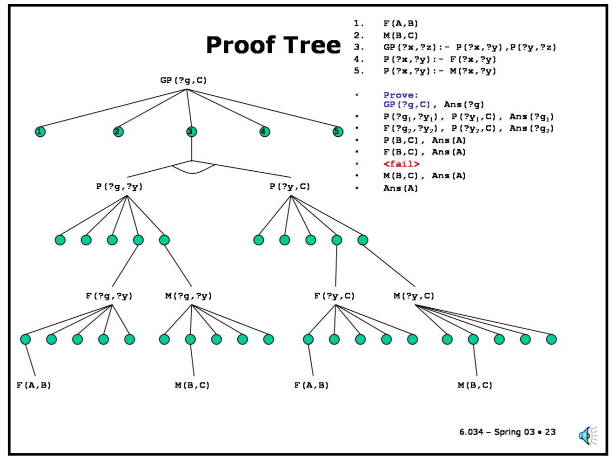
Slide 9.1.22
This matches fact 2. At this point there are no antecedents to add to the stack and the Ans literal is on the top of the stack. Note that the binding of the variable ?g to A is in fact the correct answer to our original question.

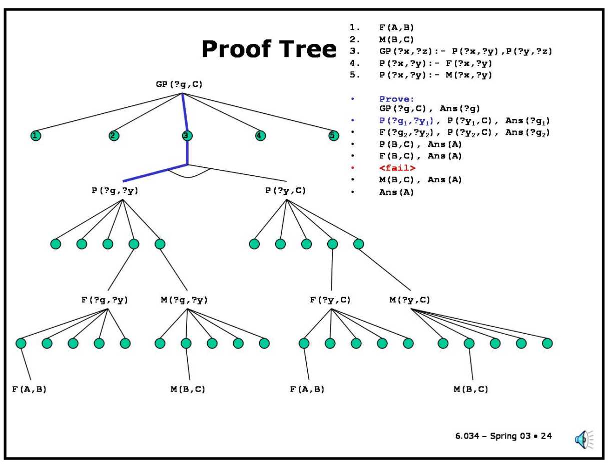
Slide 9.1.23
Another way to look at the process we have just gone through is as a form of tree search. In this search space, the states are the entries in the stack, that is, the literals that appear on our stack. The edges (shown with a green dot in the middle of each edge) are the rules or facts. However, there is one complication: a rule with multiple antecedents generates multiple children, each of which must be solved. This is indicated by the arc connecting the two descendants of rule 3 near the top of the tree.

This type of tree is called an AND-OR tree. The OR nodes come from the choice of a rule or fact to match to a goal. The AND nodes come from the multiple antecedents of a rule (all of which must be proved).

You should remember that such a tree is **implicit** in the rules and facts in our database, once we have been given a goal to prove. The tree is not constructed explicitly; it is just a way of visualizing the search process.

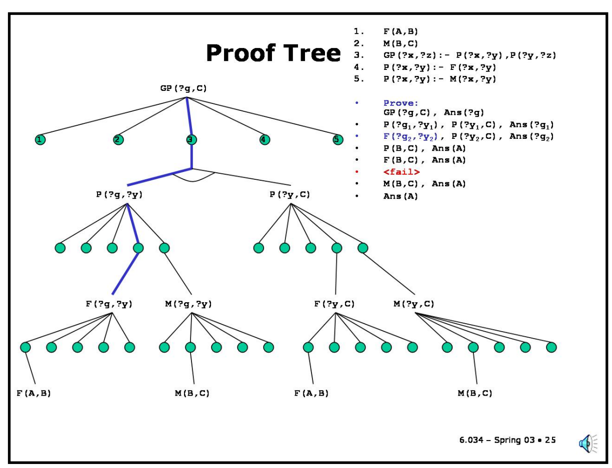
Let's go through our previous proof in this representation, which makes the choices we've made more explicit. We start with the GrandP goal at the top of the tree.

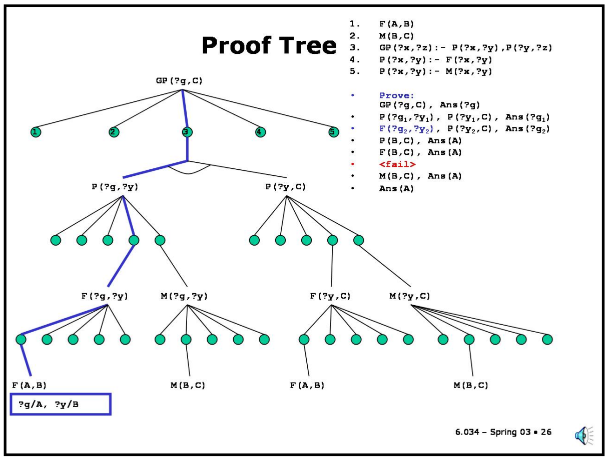




Slide 9.1.24
We match that goal to the consequent of rule 3 and we create two subgoals for each of the antecedents (after carrying out the substitutions from the unification). We will look at the first one (the one on the left) next.

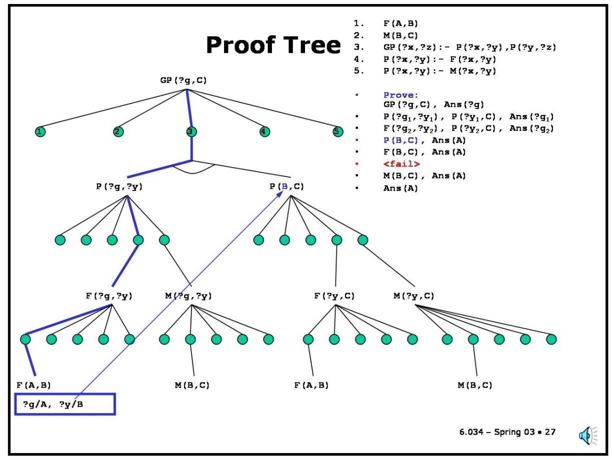
Slide 9.1.25
We match the Parent subgoal to the rule 4 and generate a `Father` subgoal.

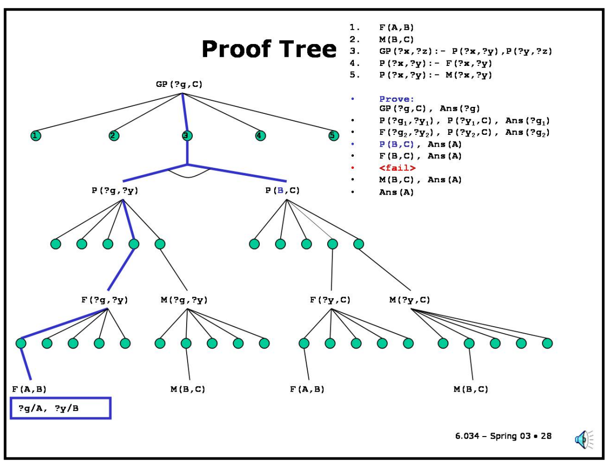




Slide 9.1.26
Which we match to fact 1 and create bindings for the variables in the goal. In all our previous steps we also created variable bindings but they were variable to variable bindings. Here, we finally match some variables to constants.

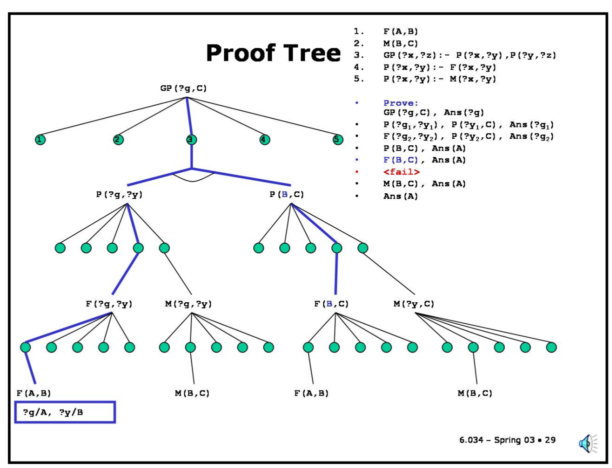
Slide 9.1.27
We have to apply this unifier to all the pending goals, including the pending Parent subgoal from rule 3. This is the part that's easy to forget when using this tree representation.

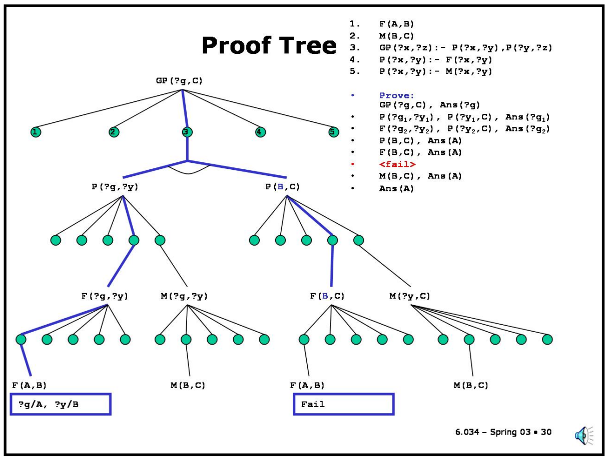




Slide 9.1.28
Now, we tackle the second Parent subgoal ...

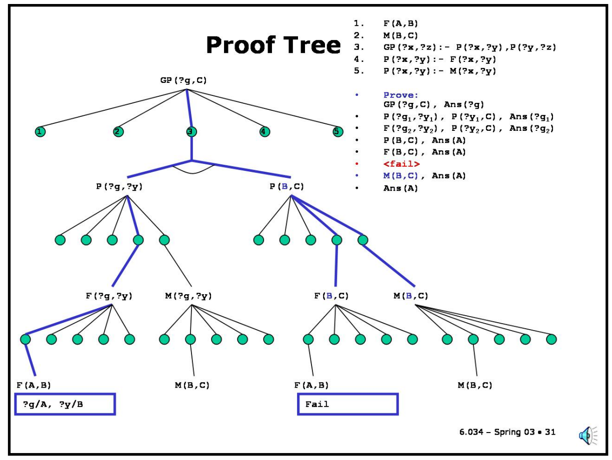
Slide 9.1.29
... which proceeds as before to match rule 4 and generate a `Father` subgoal, `Father(B,C)` in this case.

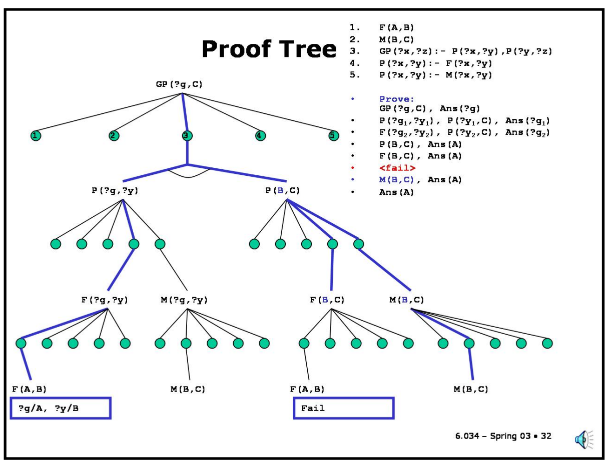




Slide 9.1.30
But, as we saw before that leads to a failure when we try to match the database.

Slide 9.1.31
So, instead, we look at the other alternative, matching the second Parent subgoal to rule 5, and generate a `Mother(B,C)` subgoal.



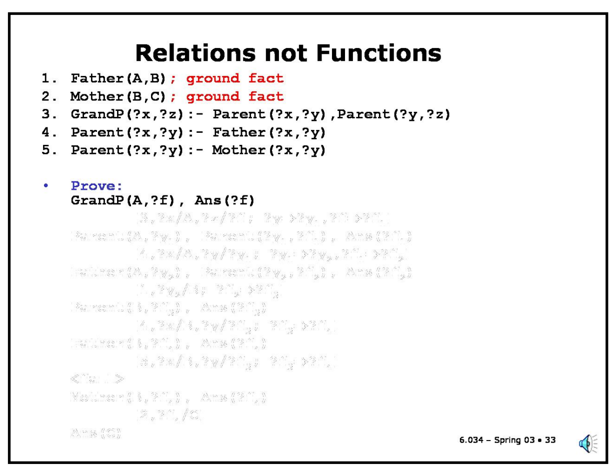


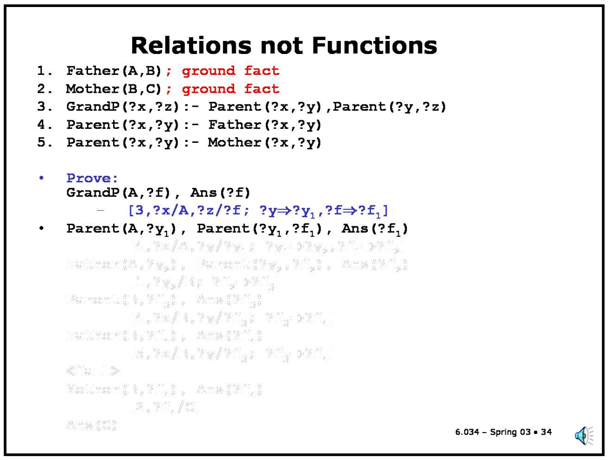
Slide 9.1.32
This matches the second fact in the database and we succeed with our proof since we have no pending subgoals to prove.

This view of the proof process highlights the search connection and is a useful mental model, although it is too awkward for any big problem.

Slide 9.1.33
At the beginning of this section, we indicated as one of the advantages of a logical representation that we could define the relationship between parents and grandparents without having to give an algorithm that might be specific to finding grandparents of grandchildren or vice versa. This is still (partly) true for logic programming. We have just seen how we could use the facts and rules shown here to find a grandparent of someone. Can we go the other way? The answer is yes.

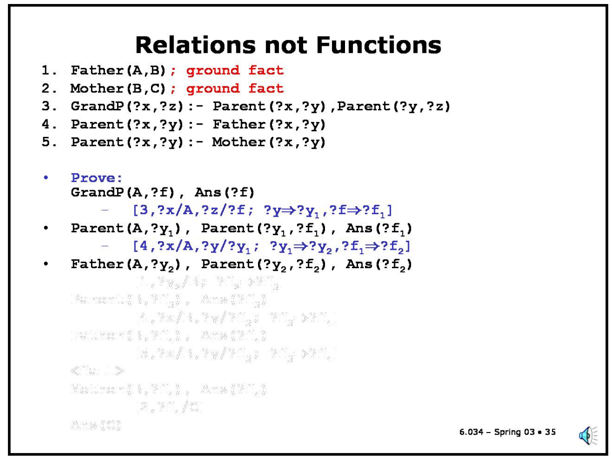
The initial goal we have shown here asks for the grandchild of A, which we know is C. Let's see how we find this answer.

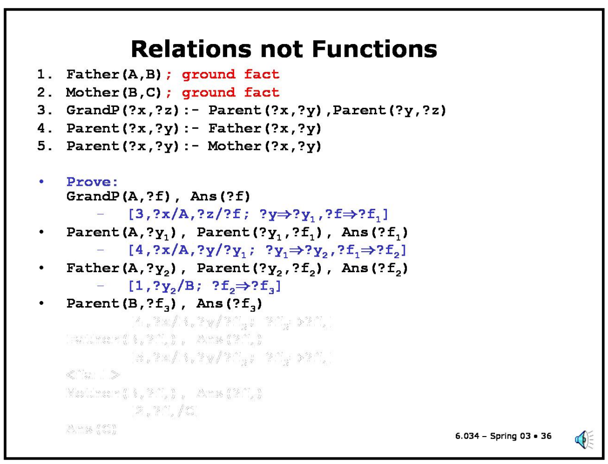




Slide 9.1.34
Once again, we match the GrandP goal to rule 3, but now the variable bindings are different. We have a constant binding in the first Parent subgoal rather than in the second.

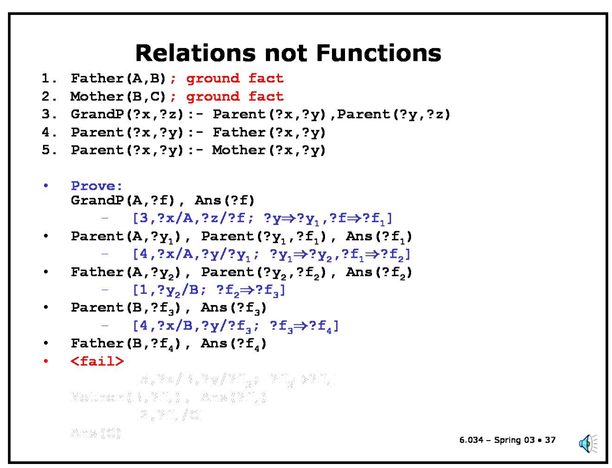
Slide 9.1.35
Once again, we match the Parent subgoal to rule 4 and get a new `Father` subgoal, this time involving A. We are basically looking for a child of A.

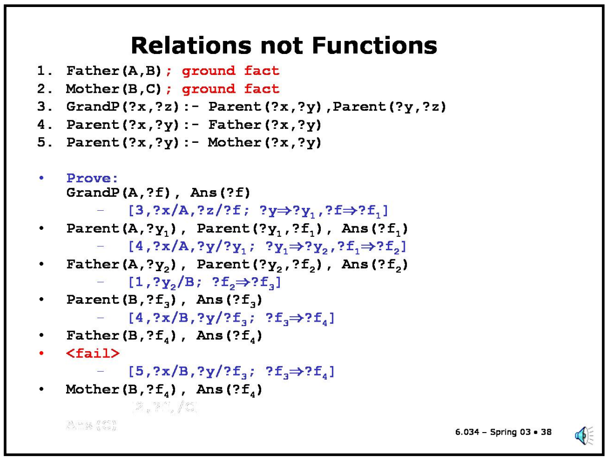




Slide 9.1.36
Then, we match the first fact, namely `Father(A,B)`, which causes us to bind the ?x variable in the second Parent subgoal to B. So, now, we look for a child of B.

Slide 9.1.37
We match the Parent subgoal to rule 4 and generate another `Father` subgoal, which fails. So, we backup to find an alternative.





Slide 9.1.38

We now match the second Parent subgoal to rule 5 and generate a `Mother(B,?f)` subgoal.

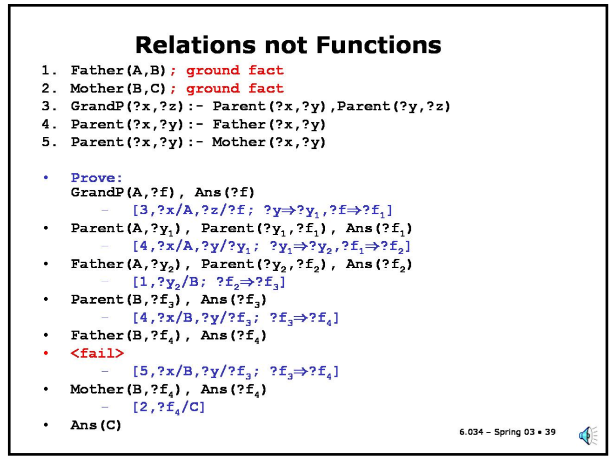
Slide 9.1.39

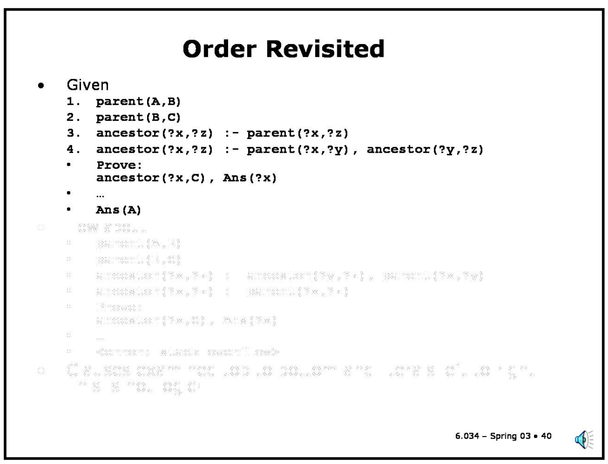
...which succeeds and binds ?f (our query variable) to C, as expected.

Note that if we had multiple grandchildren of A in the database, we could generate them all by continuing the search at any pending subgoals that had multiple potential matches.

The bottom line is that we are representing **relations** among the elements of our domain (recall that's what a logical predicate denotes) rather than computing functions that specify a single output for a given set of inputs.

Another way of looking at it is that we do not have a pre-conceived notion of which variables represent "input variables" and which are "output variables".





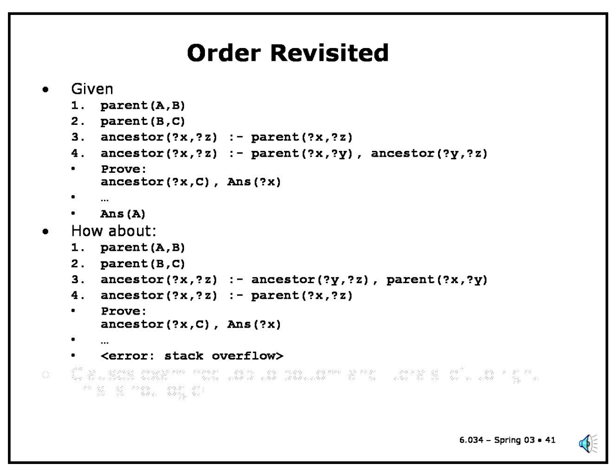
Slide 9.1.40

We have seen in our examples thus far that we explore the underlying search space in order. This approach has consequences. For example, consider the following simple rules for defining an ancestor relation. It says that a parent is an ancestor (this is the base case) and that the ancestor of a parent is an ancestor (the recursive case). You could use this definition to list a person's ancestors or, as we did for grandparent, to list a person's descendants.

But what would happen if we changed the order a little bit?

Slide 9.1.41

Here we've switched the order of rules 3 and 4 and furthermore switched the order of the literals in the recursive ancestor rule. The effect of these changes, which have no logical import, is disastrous: basically it generates an infinite loop.



Order Revisited

- Given
  1. parent(A,B)
  2. parent(B,C)
  3. ancestor(?x,?z) :- parent(?x,?z)
  4. ancestor(?x,?z) :- parent(?x,?y), ancestor(?y,?z)
  - Prove:
    ancestor(?x,C), Ans(?x)
  - …
  - Ans(A)
- How about:
  1. parent(A,B)
  2. parent(B,C)
  3. ancestor(?x,?z) :- ancestor(?y,?z), parent(?x,?y)
  4. ancestor(?x,?z) :- parent(?x,?z)
  - Prove:
    ancestor(?x,C), Ans(?x)
  - …
  - <error: stack overflow>
- Clauses examined top to bottom and literals left to right. This is not logic!

6.034 – Spring 03 • 42

Slide 9.1.42
This type of behavior is what you would expect from a recursive program if you put the recursive case before the base case. The key point is that logic programming is half way between traditional programming and logic and exactly like neither one.

Slide 9.1.43
It is often the case that we want to have a condition on a rule that says that something is not true. However, that has two problems, one is that the resulting rule would not be Horn. Furthermore, as we saw earlier, we have no way of concluding a negative literal. In logic programming one typically makes a **closed world** assumption, sometimes jokingly referred to as the "closed mind" assumption, which says that we know everything to be known about our domain. And, if we don't know it (or can't prove it), then it must be false. We all know people like this...

Negation

- We cannot have a rule such as
  - $P_1 \wedge \neg P_2 \rightarrow Q$
  - $\neg P_1 \vee P_2 \vee Q$ - not Horn (two pos literals)
  - Cannot have rule that concludes a negation
- In logic programming, we assume we have complete information about the world (closed-world assumption)

6.034 – Spring 03 • 43

Negation

- We cannot have a rule such as
  - $P_1 \wedge \neg P_2 \rightarrow Q$
  - $\neg P_1 \vee P_2 \vee Q$ - not Horn (two pos literals)
  - Cannot have rule that concludes a negation
- In logic programming, we assume we have complete information about the world (closed-world assumption)
- We use "failure to prove" as negation – a dangerous assumption.
  - Prove: ; in empty KB
    not P(?x), Ans(?x)
  - Ans(?x) ; success

6.034 – Spring 03 • 44

Slide 9.1.44
Given we assume we know everything relevant, we can simulate negation by failure to prove. This is very dangerous in general situations where you may not know everything (for example, it's not a good thing to assume in exams)...

Slide 9.1.45
... but very useful in practice. For example, we can write rules of the form "if there are no other acceptable flights, accept a long layover" and we establish this by looking over all the known flights.

Negation

- But often very useful in finite domains, e.g. flights database, products of a company, etc.
- For example:

```
Layover_not_too_long(?f1, ?f2) :-
  Arrival_time(?f1, ?t1),
  Departure_time(?f2, ?t2),
  not Alternative_connection(?f1, ?t1, ?f2, ?t2)
```

- Will succeed if the Alternative_connection literal fails.

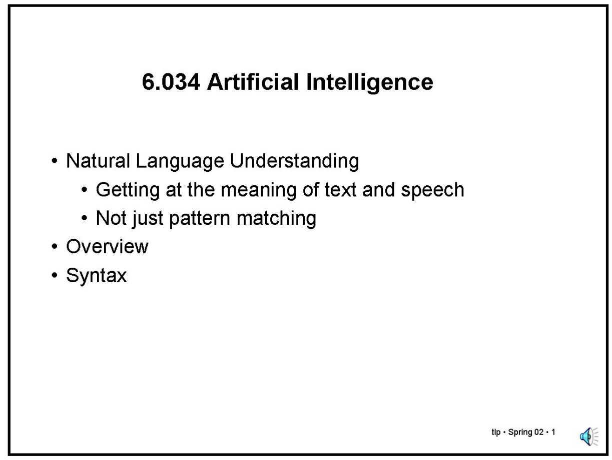
6.034 – Spring 03 • 45

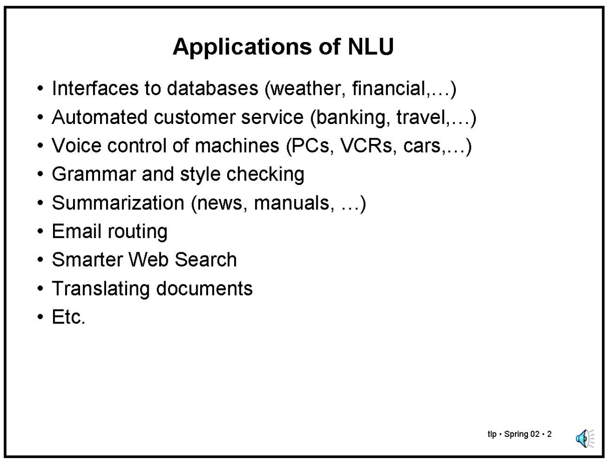
6.034 Notes: Section 9.2

Slide 9.2.1

In this chapter, we take a quick survey of some aspects of natural language understanding. Our goal will be to capture the **meaning** of sentences in some detail. This will involve finding representations for the sentences that can be connected to more general knowledge about the world. This is in contrast to approaches to dealing with language that simply try to match textual patterns, for example, web search engines.

We will briefly provide an overview of the various levels and stages of natural language processing and then begin a more in-depth exploration of language syntax.





Slide 9.2.2

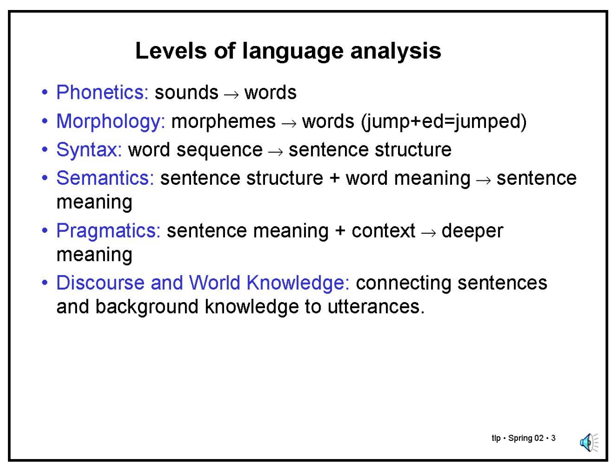
The motivation for the study of natural language understanding is twofold. One is, of course, that language understanding is one of the quintessentially human abilities and an understanding of human language is one of key steps in the understanding of human intelligence.

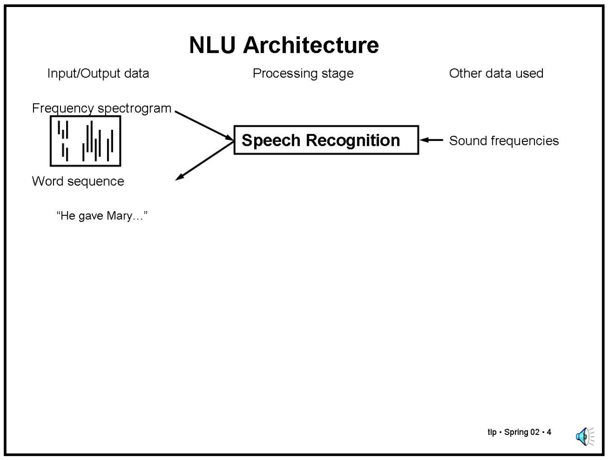
In addition to this fundamental long-term scientific goal, there is a pragmatic shorter-term engineering goal. The potential applications of in-depth natural language understanding by computers are endless. Many of the applications listed here are already available in some limited forms and there is a great deal of research aimed at extending these capabilities.

Slide 9.2.3

Language is an enormously complex process, which has been studied in great detail for a long time. The study of language is usually partitioned into a set of separate sub-disciplines, each with a different focus. For example, phonetics concerns the rules by which sounds (phonemes) combine to produce words. Morphology studies the structure of words: how tense, number, etc is captured in the form of the word. Syntax studies how words are combined to produce sentences. Semantics studies how the meaning of words are combined with the structure of a sentence to produce a meaning for the sentence, usually a meaning independent of context. Pragmatics concerns how context factors into the meaning (e.g. "it's cold in here") and finally there's the study of how background knowledge is used to actually understand the meaning the utterances.

We will consider the process of understanding language as one of progressing through various "stages" or processing that break up along the lines of these various subfields. In practice, the processing may not be separated as cleanly as that, but the division into stages allows us to focus on one type of problem at a time.





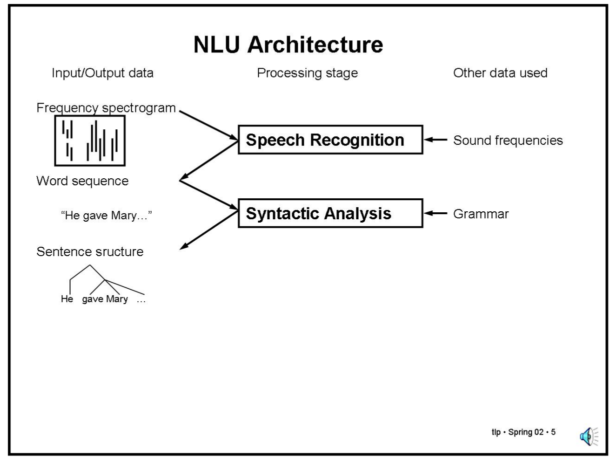
Slide 9.2.4

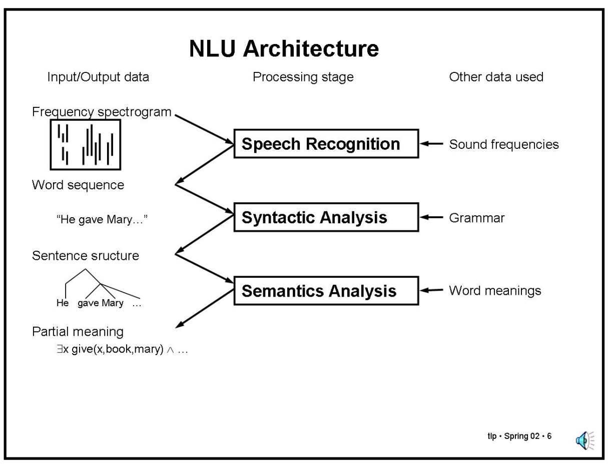
If one considers the problem of understanding speech, the first stage of processing is, conceptually, that of converting the spoken utterance into a string of words. This process is extremely complex and quite error prone and, today, cannot be solved without a great deal of knowledge about what the words are likely to be. But, in limited domains, fairly reliable transcription is possible. Even more reliability can be achieved if we think of this stage as producing a few alternative interpretations of the speech signal, one of which is very likely to be the correct interpretation.

Slide 9.2.5

The next step is **syntax**, that is, computing the structure of the sentence, usually in terms of phrases, such as noun phrases, verb phrases and prepositional phrases. These nested phrases will be the basis of all subsequent processing. Syntactic analysis is probably the best developed area in computational linguistics but, nevertheless, there is no universally reliable "grammar of English" that one can use to parse sentences as well as trained people can. There are, however, a number of wide-coverage grammars available.

We will see later that, in general, there will not be a unique syntactic structure that can be derived from a sequence of words.



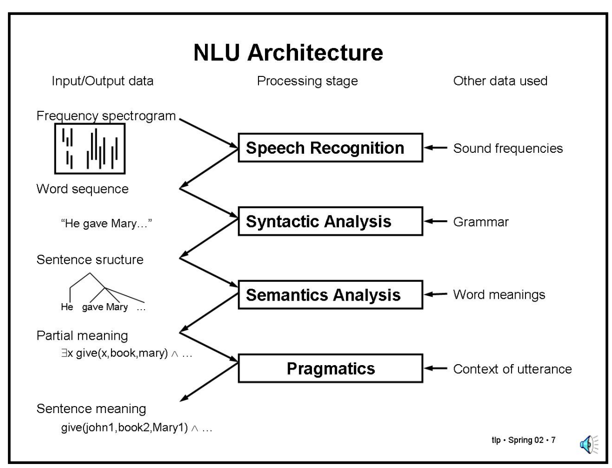


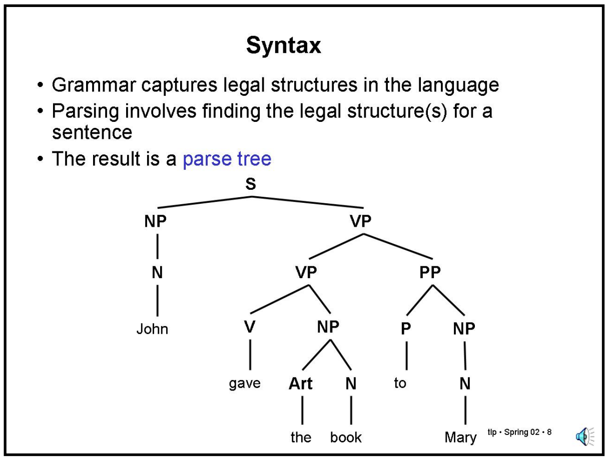
Slide 9.2.6

Given the sentence structure, we can begin trying to attach meaning to the sentence. The first such phase is known as **semantics**. The usual intent here is to translate the syntactic structure into some form of logical representation of the meaning - but without the benefit of context. For example, who is being referred to by a pronoun may not be determined at this point.

Slide 9.2.7

We will focus in this chapter on syntax and semantics, but clearly there is a great deal more work to be done before a sentence could be understood. One such step, sometimes known as **pragmatics**, involves among other things disambiguating the various possible senses of words, possible syntactic structures, etc. Also, trying to identify the referent of pronouns and descriptive phrases. Ultimately, we have to connect the meaning of the sentence with general knowledge in order to be able to act on it. This is by far the least developed aspect of the whole enterprise. In practice, this phase tends to be very application specific.





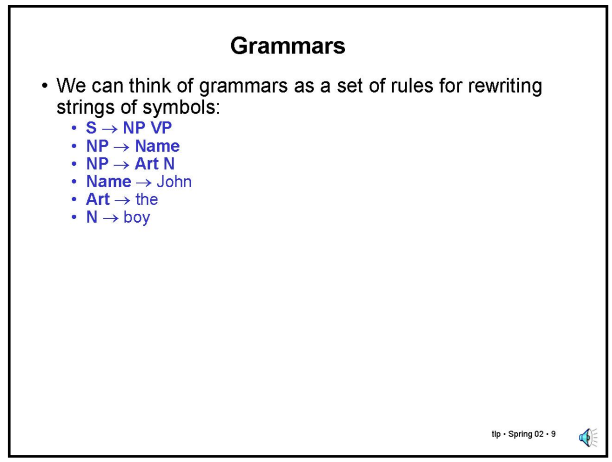
Slide 9.2.8

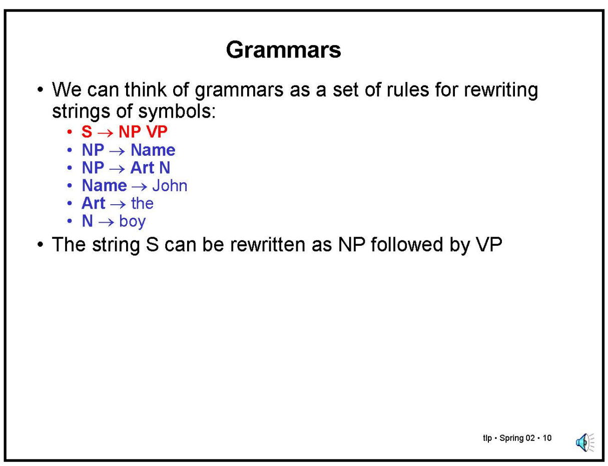
In the rest of this section, we will focus on syntax. The description of the legal structures in a language is called a **grammar**. We'll see examples of these later. Given a sentence, we use the grammar to find the legal structures for a sentence. This process is called **parsing** the sentence. The result is one or more **parse trees**, such as the one shown here, which indicates that the sentence can be broken down into two **constituents**, a noun phrase and a verb phrase. The verb phrase, in turn, is composed of another verb phrase followed by a prepositional phrase, etc.

Our attempt to understand sentences will be based on assigning meaning to the individual constituents and then combining them to construct the meaning of the sentence. So, in this sense, the constituent phrases are the atoms of meaning.

Slide 9.2.9

A grammar is typically written as a set of **rewrite rules** such as the ones shown here in blue. Bold-face symbols, such as S, NP and VP, are known as non-terminal symbols, in that they can be further re-written. The non-bold-face symbols, such as John, the and boy, are the words of the language - also known as the terminal symbols.



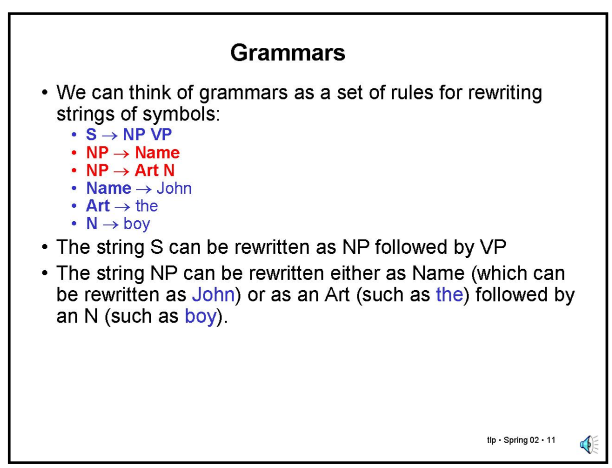


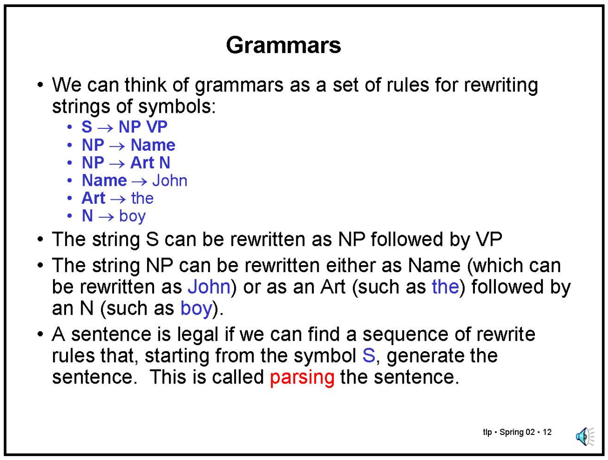
Slide 9.2.10

The first rule, S -> NP VP, indicates that the symbol S (standing for sentence) can be rewritten as NP (standing for noun phrase) followed by VP (standing for verb phrase).

Slide 9.2.11

The symbol NP, can be rewritten either as a Name or as an Art(icle), such as the, followed by a N(oun), such as boy.





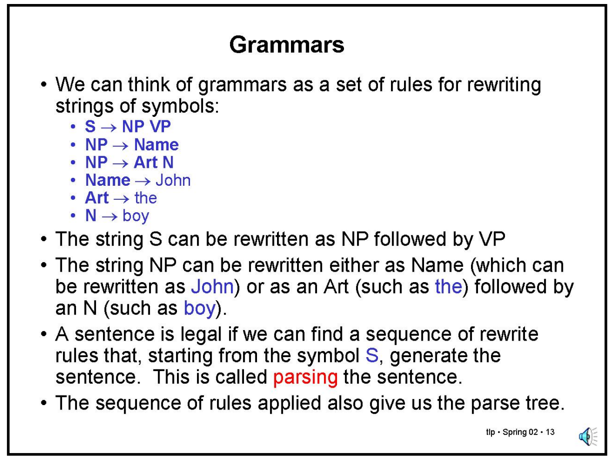
Slide 9.2.12

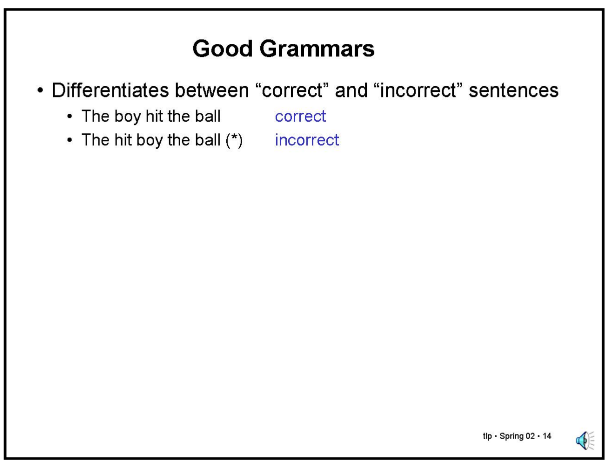
If we can find a sequence of rewrite rules that will rewrite the initial S into the input sentence, the we have successfully parsed the sentence and it is legal.

Note that this is a search process like the ones we have studied before. We have an initial state, S, at any point in time, we have to decide which grammar rule to apply (there will generally be multiple choices) and the result of the application is some sequence of symbols and words. We end the search when the words in the sentence have been obtained or when we have no more rules to try.

Slide 9.2.13

Note that the successful sequence of rules applied to achieve the rewriting give us the parse tree. Note that this excludes any "wrong turns" we might have taken during the search.





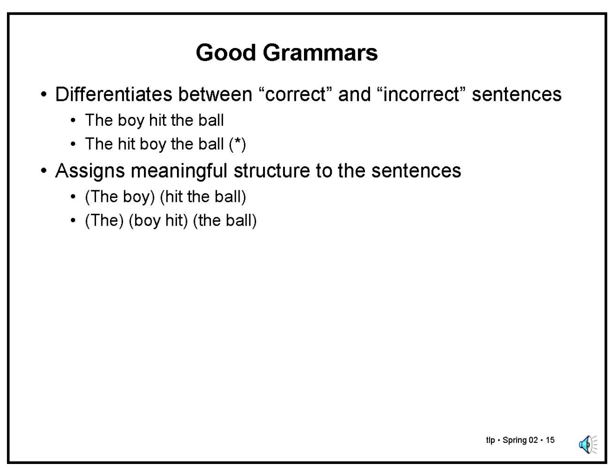
Slide 9.2.14

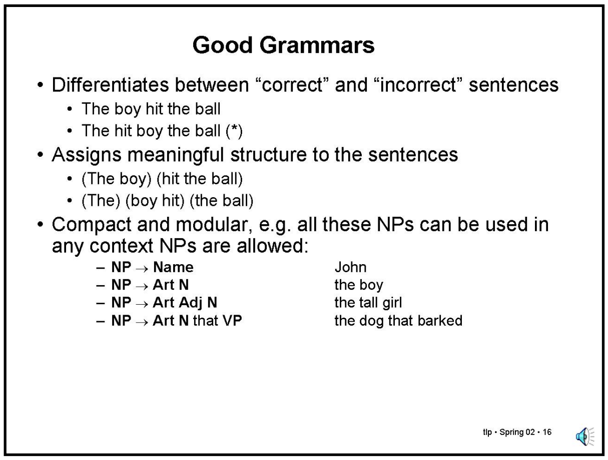
What makes a good grammar?

The primary criterion is that it differentiates correct sentences from incorrect ones. (By convention an asterisk next to a sentence indicates that it is not grammatical).

Slide 9.2.15

The other principal criterion is that it assigns "meaningful" structures to sentences. In our case, this literally means that it should be possible to assign meaning to the sub-structures. For example, a noun phrase will denote an object while a verb phrase will denote an event or an action, etc.





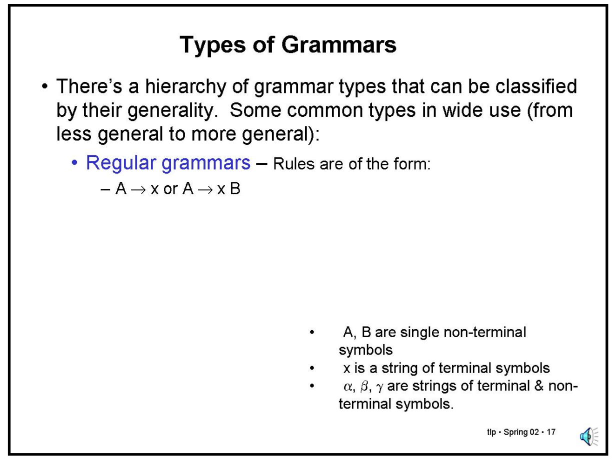
Slide 9.2.16

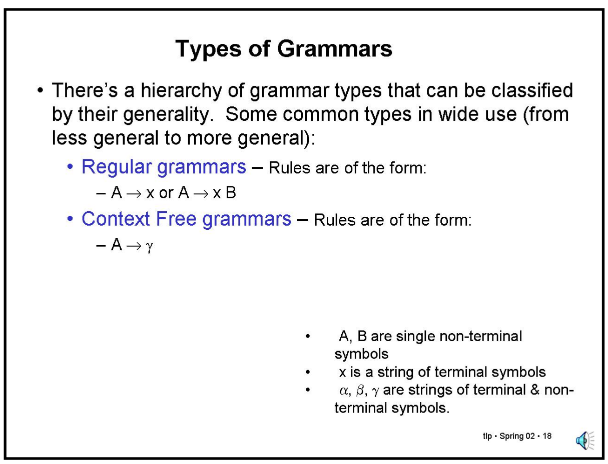
Among the grammars that meet our principal criteria we prefer grammars that are compact, that is, have fewer rules and are modular, that is, define structures that can be re-used in different contexts - such as noun-phrase in this example. This is partly for efficiency reasons in parsing, but is partly because of Occam's Razor - the simplest interpretation is best.

Slide 9.2.17

There are many possible types of grammars. The three types that are most common in computational linguistics are regular grammars, context-free grammars and context-sensitive grammars. These grammars can be arranged in a hierarchy (the Chomsky hierarchy) according to their generality. In this hierarchy, the grammars in higher levels fully contain those below and there are languages in the more general grammars not expressible in the less general grammars.

The least general grammar of some interest in computational linguistics are the **regular grammars**. These grammars are composed of rewrite rules of the form A -> x or A -> x B. That is, a non-terminal symbol can be rewritten as a string of terminal symbols or by a string of terminal symbols followed by a non-terminal symbol.





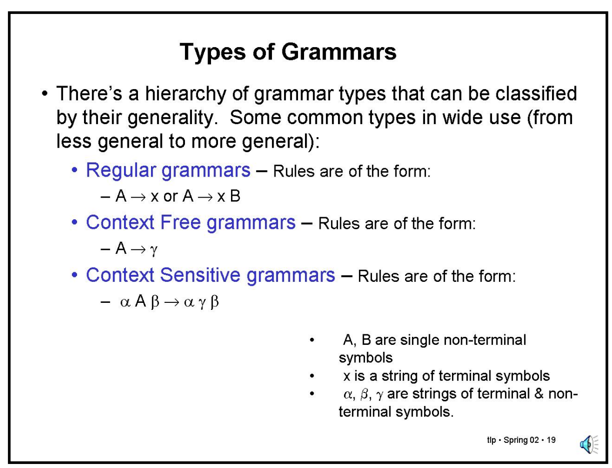
Slide 9.2.18

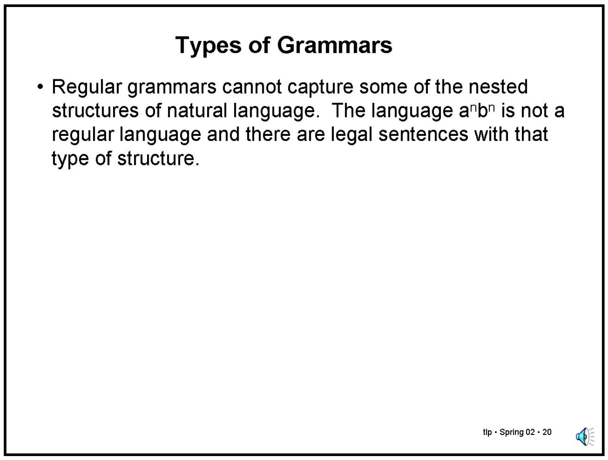
At the next level are the **context-free grammars**. In these grammars, a non-terminal symbol can be rewritten into any combination of terminal and non-terminal symbols. Note that since the non-terminal appears alone in the left-hand side (lhs) of the rule, it is re-written independent of the context in which it appears - and thus the name.

Slide 9.2.19

Finally, in **context-sensitive** grammars, we are allowed to specify a context for the rewriting operation.

There are even more general grammars (known as Type 0) which we will not deal with at all.



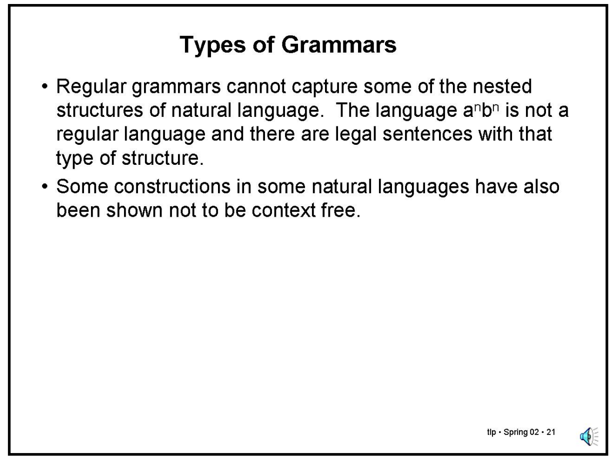


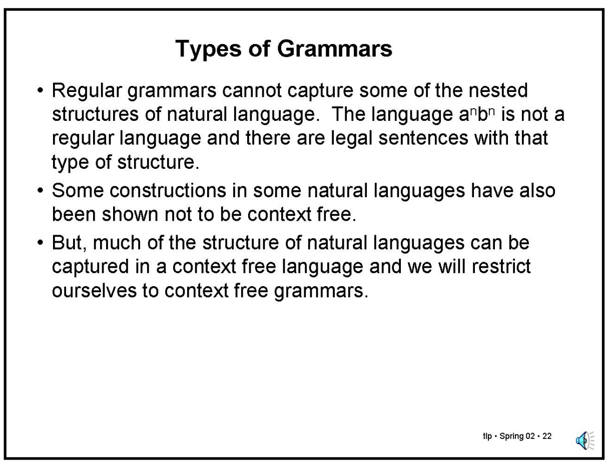
Slide 9.2.20

The language of parenthesized expressions, that is, n left parens followed by n right parens is the classic example of a non-regular language that requires us to move to context-free grammars. There are legal sentences in natural languages whose structure is isomorphic to that of parenthesized expressions (the cat likes tuna; the cat the dog chased likes tuna; the cat the dog the rat bit chased likes tuna). Therefore, we need at least a context-free grammar to capture the structure of natural languages.

Slide 9.2.21

There have been several empirical proofs that there exist natural languages that have non-context-free structure.



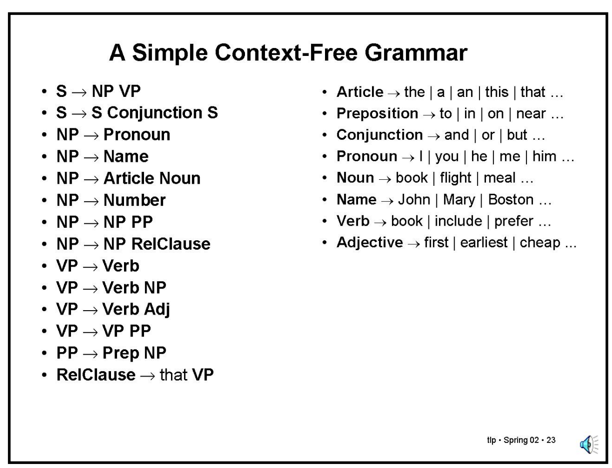


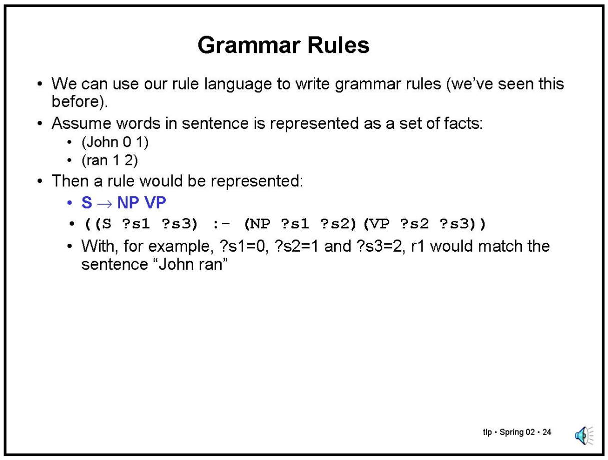
Slide 9.2.22

However, much of natural language can be expressed in context-free grammars extended in various ways. We will limit ourselves to this class.

Slide 9.2.23

Here's an example of a context free grammar for a small subset of English. Note that the vertical band is a short hand which can be read as "or"; it is a notation for combining multiple rules with identical left hand sides. Many variations on this grammar are possible but this illustrates the style of grammar that we will be considering.





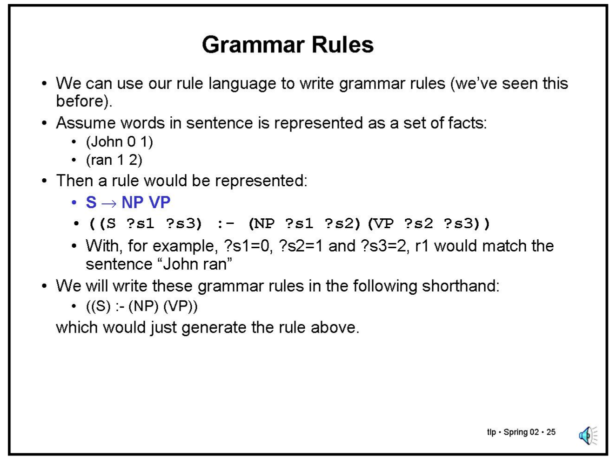
Slide 9.2.24

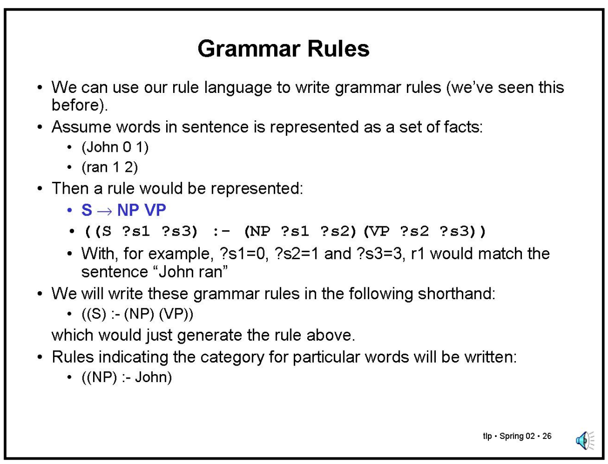
At this point, we should point out that there is a strong connection between these grammar rules that we have been discussing and the logic programming rules that we have already studied. In particular, we can write context-free grammar rules in our simple Prolog-like rule language.

We will assume that a set of facts are available that indicate where the particular words in a sentence start and end (as shown here). Then, we can write a rule such as S -> NP VP as a similar Prolog-like rule, where each non-terminal is represented by a fact that indicates the type of the constituent and the start and end indices of the words.

Slide 9.2.25

In the rest of this Chapter, we will write the rules in a simpler shorthand that leaves out the word indices. However, we will understand that we can readily convert that notation into the rules that our rule-interpreters can deal with.



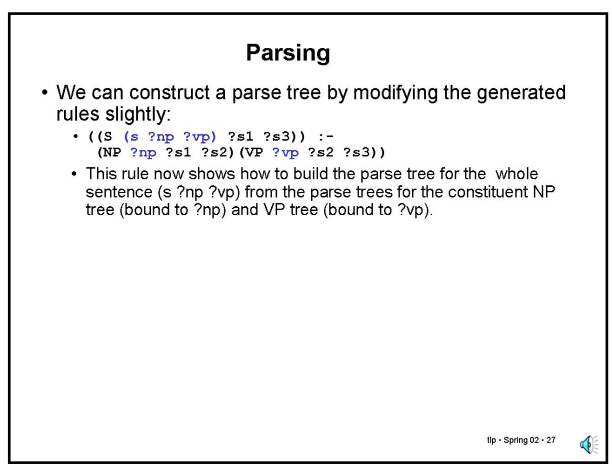


Slide 9.2.26

We can also use the same syntax to specify the word category of individual words and also turn these into rules.

Slide 9.2.27

We can make a small modification to the generated rule to keep track of the parse tree as the rules are being applied. The basic idea is to introduce a new argument into each of the facts which keeps track of the parse tree rooted at that component. So, the parse tree for the sentence is simply a list, starting with the symbol S, and whose other components are the trees rooted at the NP and VP constituents.



Parsing

- We can construct a parse tree by modifying the generated rules slightly:
  - `((S (s ?np ?vp) ?s1 ?s3)) :- (NP ?np ?s1 ?s2)(VP ?vp ?s2 ?s3))`
  - This rule now shows how to build the parse tree for the whole sentence (s ?np ?vp) from the parse trees for the constituent NP tree (bound to ?np) and VP tree (bound to ?vp).
- This can all be generated automatically from the same shorthand notation:
  - ((S) :- (NP) (VP))

tlp • Spring 02 • 28

Slide 9.2.28

This additional bit of bookkeeping can also be generated automatically from the shorthand notation for the rule.

Slide 9.2.29

Note that given the logic rules from the grammar and the facts encoding a sentence, we can use chaining (either forward or backward) to parse the sentence. Let's look at this in more detail.

Parsing

- We can construct a parse tree by modifying the generated rules slightly:
  - `((S (s ?np ?vp) ?s1 ?s3)) :- (NP ?np ?s1 ?s2)(VP ?vp ?s2 ?s3))`
  - This rule now shows how to build the parse tree for the whole sentence (s ?np ?vp) from the parse trees for the constituent NP tree (bound to ?np) and VP tree (bound to ?vp).
- This can all be generated automatically from the same shorthand notation:
  - ((S) :- (NP) (VP))
- Given a set of rules (and word facts), we can use forward or backward chaining to parse a sentence.

tlp • Spring 02 • 29

Top-Down vs Bottom-Up

- Simple parsers are often classified as top-down or bottom-up where top and bottom refer to the parse tree.
- Backwards chaining on the grammar rules we have seen is a top-down approach to parsing (starts with S and works towards the words).
- Forward chaining on the grammar rules is a bottom up approach (starts with the words and works towards S).

tlp • Spring 02 • 30

Slide 9.2.30

A word on terminology. Parsers are often classified into **top-down** and **bottom-up** depending whether they work from the top of the parse tree down towards the words or vice-versa. Therefore, backward-chaining on the rules leads to a top-down parser, while forward-chaining, which we will see later, leads to a bottom-up parser. There are more sophisticated parsers that are neither purely top-down nor bottom-up, but we will not pursue them here.

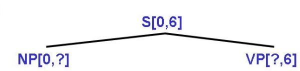
Slide 9.2.31

Let us look at how the sample grammar can be used in a top-down manner (backward-chaining) to parse the sentence "John gave the book to Mary". We start backchaining with the goal S[0,6]. The first relevant rule is the first one and so we generate two subgoals: NP[0,?] and VP[?,6].

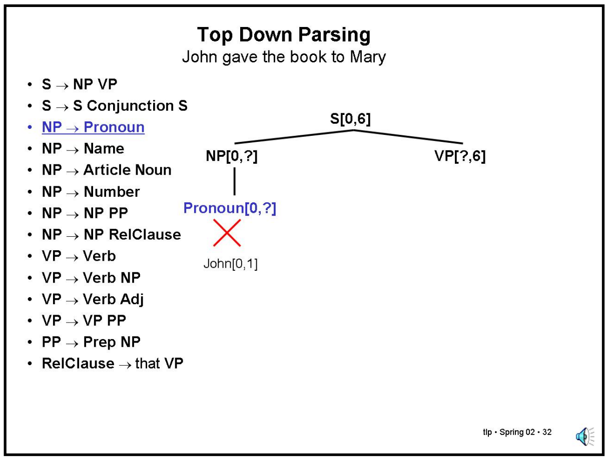
Top Down Parsing

John gave the book to Mary

- S → NP VP
- S → S Conjunction S
- NP → Pronoun
- NP → Name
- NP → Article Noun
- NP → Number
- NP → NP PP
- NP → NP RelClause
- VP → Verb
- VP → Verb NP
- VP → Verb Adj
- VP → VP PP
- PP → Prep NP
- RelClause → that VP



tlp • Spring 02 • 31

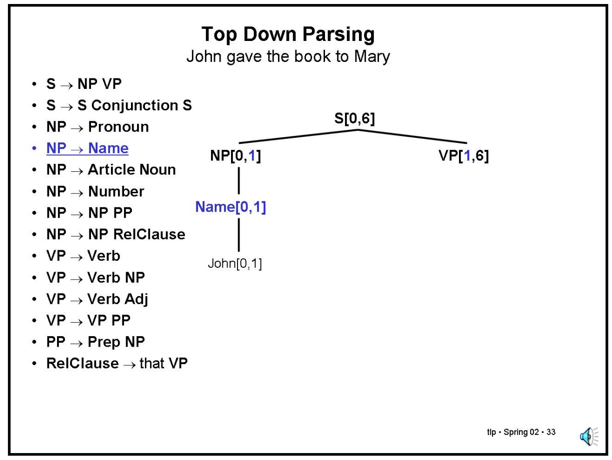


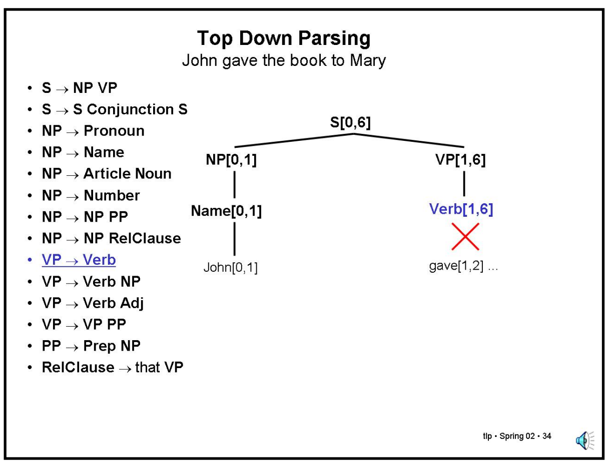
Slide 9.2.32

Assuming we examine the rules in order, we first attempt to apply the NP -> Pronoun rule. But that will fail when we actually try to find a pronoun at location 0.

Slide 9.2.33

Then we try to see if NP -> Name will work, which it does, since the first word is John and we have the rule that tells us that John is a Name. Note that this will also bind the end of the VP phrase and the start of the VP to be at position 1.



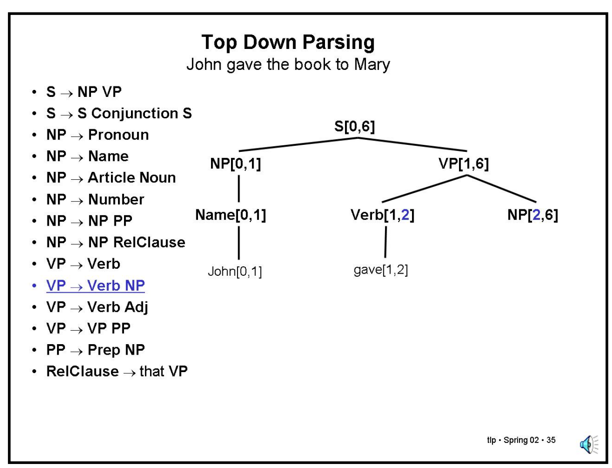


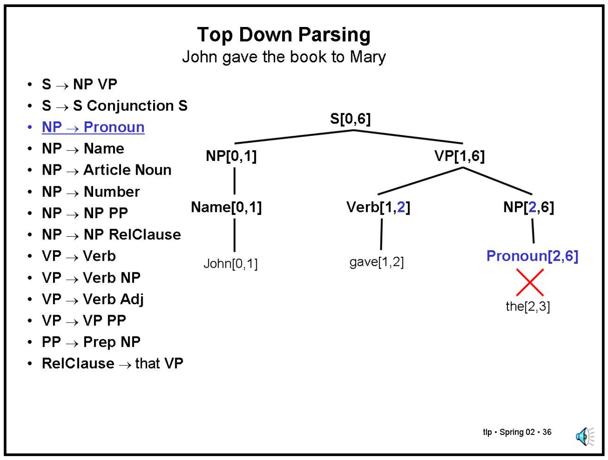
Slide 9.2.34

So, we move on to the pending VP. Our first relevant rule is VP -> Verb, which will fail. Note, however, that there is a verb starting at location 1, but at this point we are looking for a verb phrase from positions 1 to 6, while the verb only goes from 1 to 2.

Slide 9.2.35

So, we try the next VP rule, which will look for a verb followed by a noun phrase, spanning from words 1 to 6. The Verb succeeds when we find "gave" in the input.



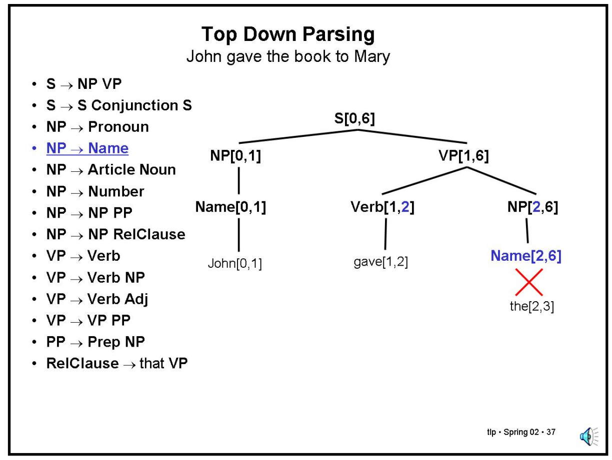


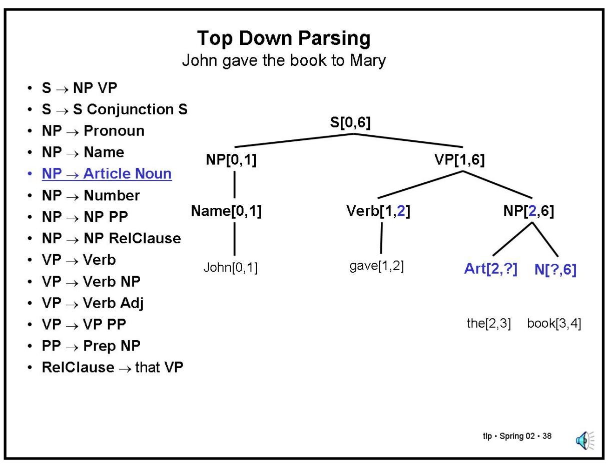
Slide 9.2.36

Now we try to find an NP starting at position 2. First we try the pronoun rule, which fails.

Slide 9.2.37

Then we try the name rule, which also fails.



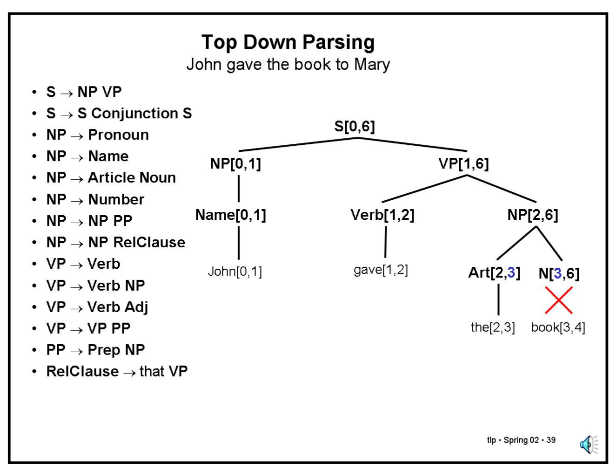


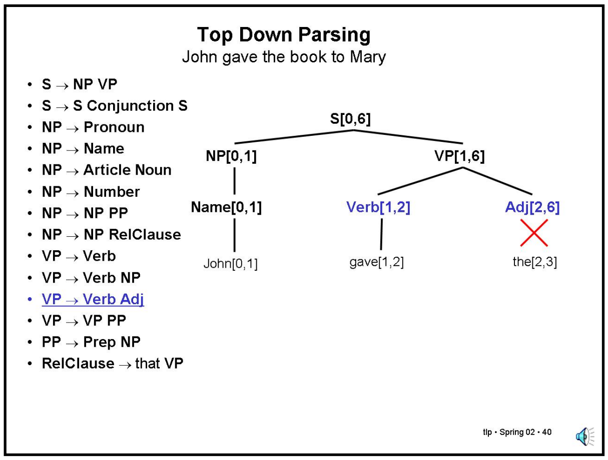
Slide 9.2.38

Then we try the article followed by a noun.

Slide 9.2.39

The article succeeds when we find "the" in the input. Now we try to find a noun spanning words 3 to 6. We have a noun in the input but it only spans one word, so we fail.



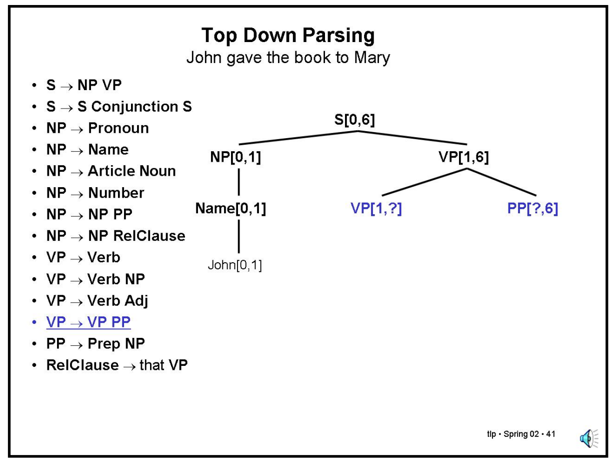


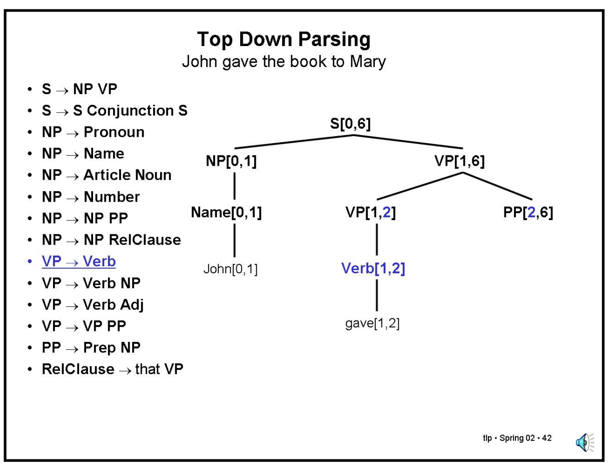
Slide 9.2.40

We eventually fail back to our choice of the VP rule and so we try the next VP rule candidate, involving a Verb followed by an adjective, which also fails.

Slide 9.2.41

The next VP rule, looks for a VP followed by prepositional phrase.



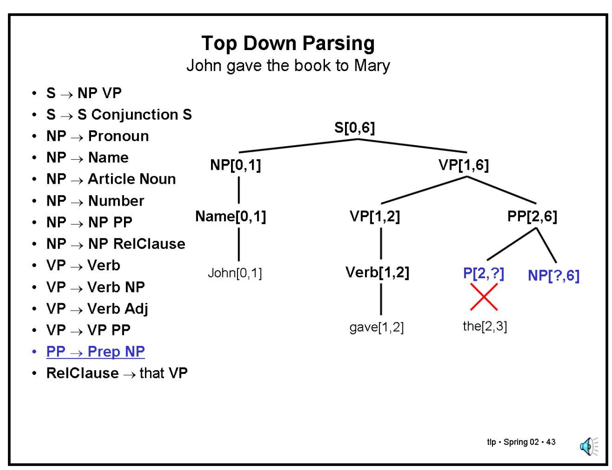


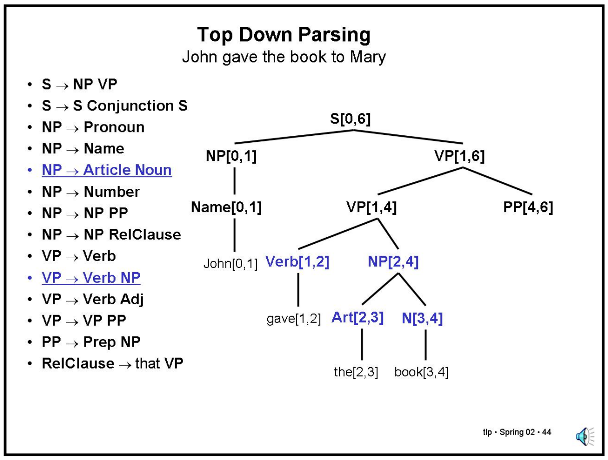
Slide 9.2.42

The first VP succeeds by finding the verb "gave", which now requires us to find a prepositional phrase starting at position 2.

Slide 9.2.43

We proceed to try to find a preposition at position 2 and fail.



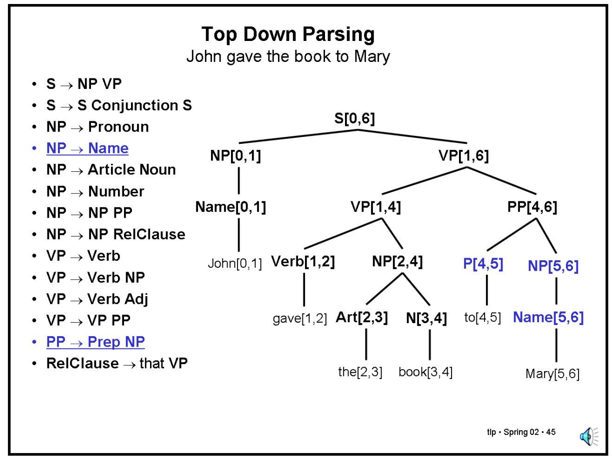


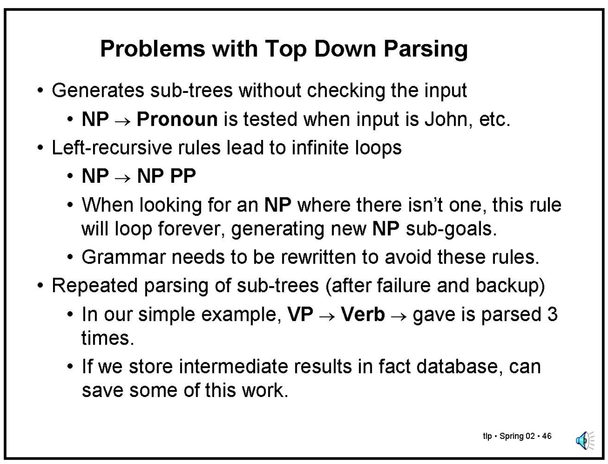
Slide 9.2.44

We fail back to trying an alternative rule (verb followed by NP) for the embedded VP, which now successfully parses "gave the book" and we proceed to look for a prepositional phrase in the range 4 to 6.

Slide 9.2.45

Which successfully parses, "to Mary", and the complete parse succeeds.





Slide 9.2.46

There are a number of problems with this top-down parsing strategy. One that substantially impacts efficiency is that rules are chosen without checking whether the next word in the input can possibly be compatible with that rule. There are simple extensions to the top-down strategy to overcome this difficulty (by keeping a table of constituent types and the lexical categories that can begin them).

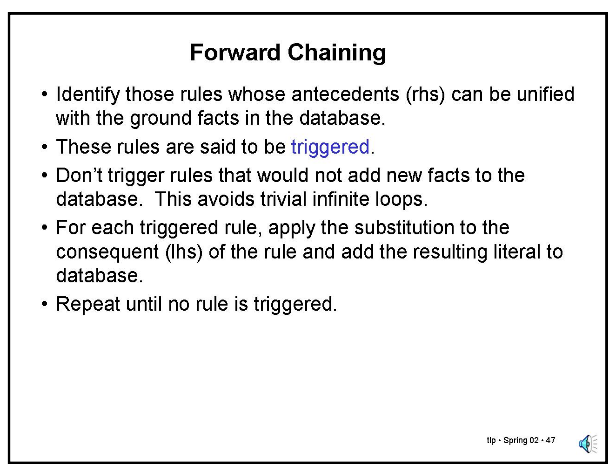
A more substantial problem, is that rules such as NP -> NP PP (left-branching rules) will cause an infinite loop for this simple top-down parsing strategy. It is possible to modify the grammar to turn such rules into right-branching rules - but that may not be the natural interpretation.

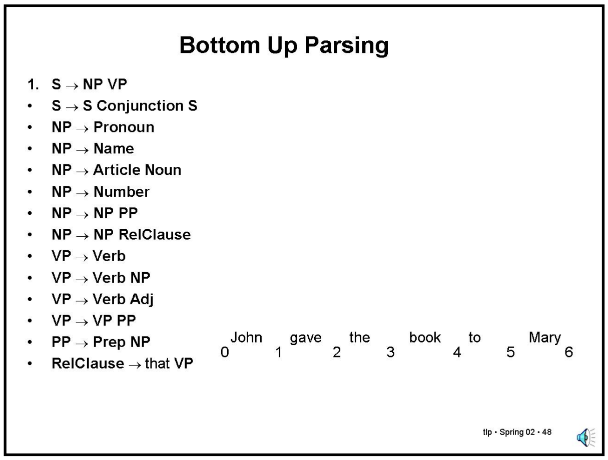
Note that the top-down strategy is carrying out a search for a correct parse and it ends up doing wasted work, repeatedly parsing parts of the sentence during its attempts. This can be avoided by building a table of parses that have been previously discovered (stored in the fact database) so they can be reused rather than re-discovered.

Slide 9.2.47

So far we have been using our rules together with our backchaining algorithm for logic programming to do top-down parsing. But, that's not the only way we can use the rules.

An alternative strategy starts by identifying any rules for which all the literals in their right hand side can be unified (with a single unifier) to the known facts. These rules are said to be **triggered**. For each of those triggered rules, we can add a new fact for the left hand side (with the appropriate variable substitution). Then, we repeat the process. This is known as **forward chaining** and corresponds to bottom-up parsing, as we will see next.



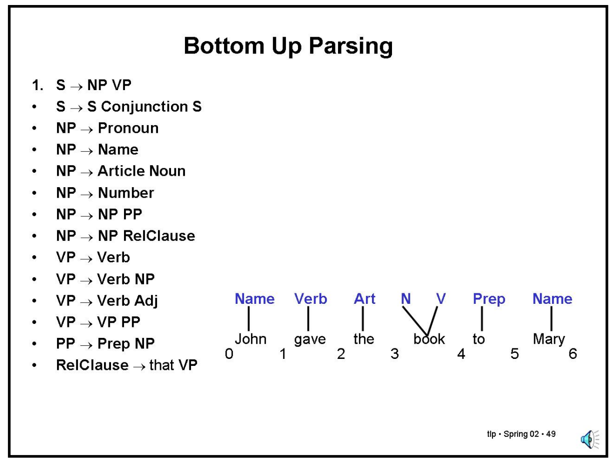


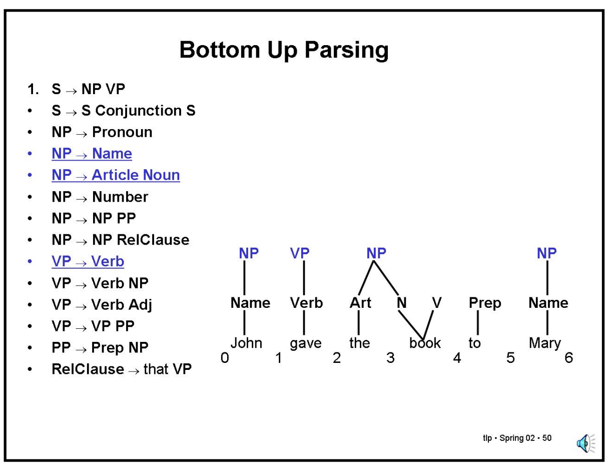
Slide 9.2.48

Now, let's look at bottom-up parsing. We start with the facts indicating the positions of the words in the input, shown here graphically.

Slide 9.2.49

Note that all the rules indicating the lexical categories of the individual words, such as Name, Verb, etc, all trigger and can all be run to add the new facts shown here. Note that book is ambiguous, both a noun and a verb, and both facts are added.



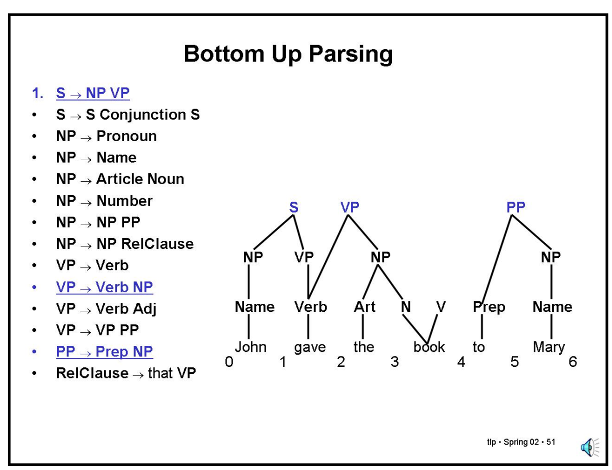


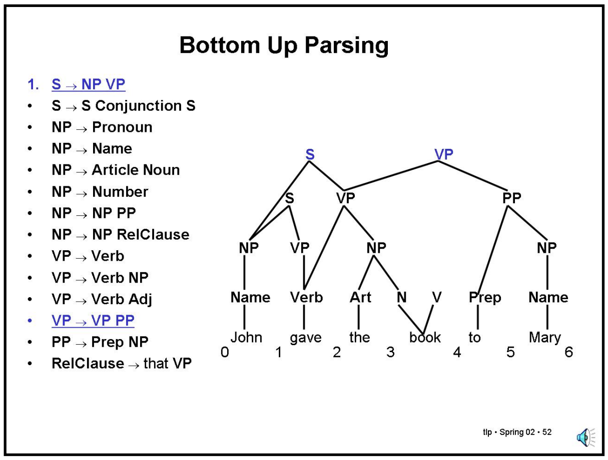
Slide 9.2.50

Now these three rules (NP -> Name, VP-> Verb and NP -> Art N) all trigger and can be run.

Slide 9.2.51

Then, another three rules (S -> NP VP, VP -> Verb NP and PP -> Prep NP) trigger and can be run. Note that we now have an S fact, but it does not span the whole input.



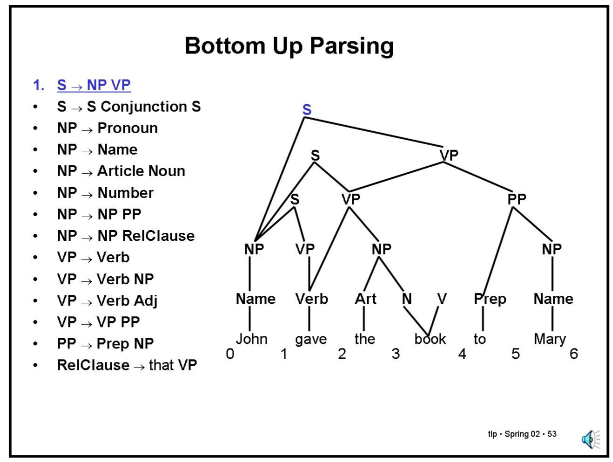


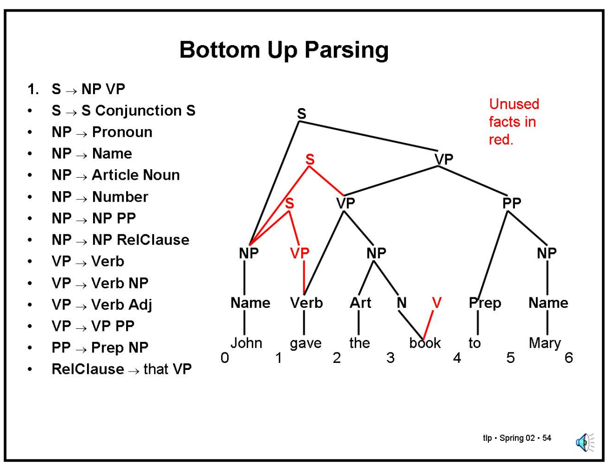
Slide 9.2.52

Now, we trigger and run the S rule again as well as the VP->VP PP rule.

Slide 9.2.53

Finally, we run the S rule covering the whole input and we can stop.



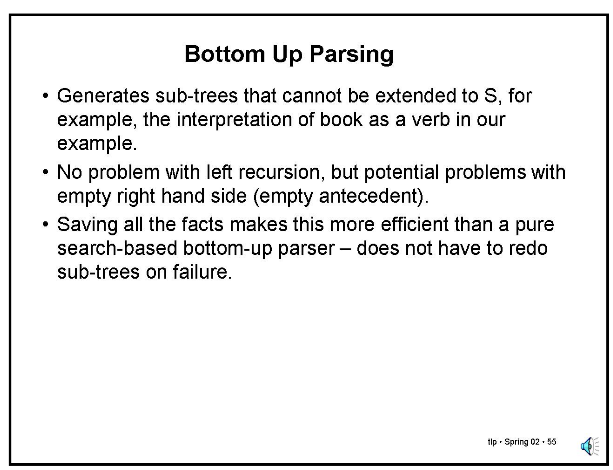


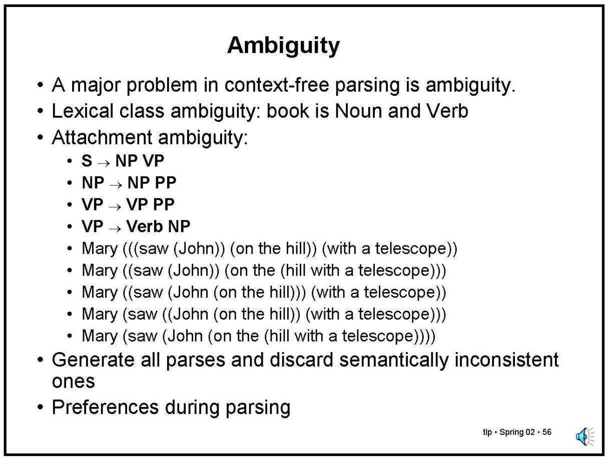
Slide 9.2.54

Note that (not surprisingly) we generated some facts that did not make it into our final structure.

Slide 9.2.55

Bottom-up parsing, like top-down parsing, generates wasted work in that it generates structures that cannot be extended to the final sentence structure. Note, however, that bottom-up parsing has no difficulty with left-branching rules, as top-down parsing did. Of course, rules with an empty right hand side can always be used, but this is not a fundamental problem if we require that triggering requires that a rule adds a new fact. In fact, by adding all the intermediate facts to the data base, we avoid some of the potential wasted work of a pure search-based bottom-up parser.





Slide 9.2.56

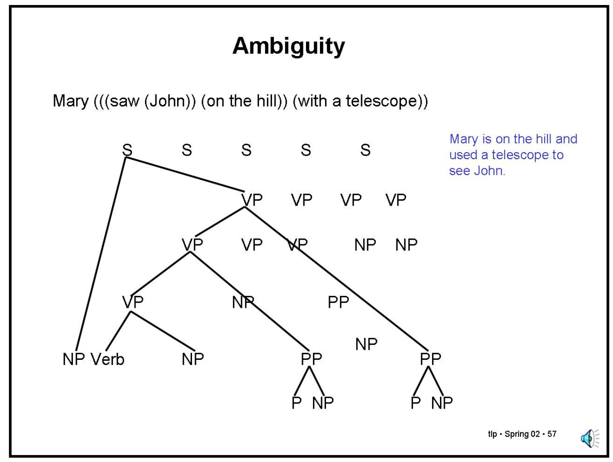
One of the key facts of natural language grammars is the presence of ambiguity of many types. We have already seen one simple example of **lexical ambiguity**, the fact that the word book is both a noun and a verb. There are many classic examples of this phenomenon, such as "Time flies like an arrow", where all of "time", "flies" and "like" are ambiguous lexical items. If you can't see the ambiguity, think about "time flies" as analogous to "fruit flies".

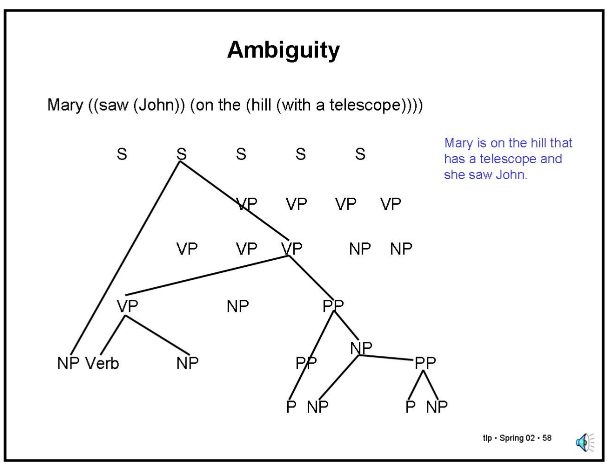
Perhaps a more troublesome form of ambiguity is known as **attachment ambiguity**. Consider the simple grammar shown here that allows prepositional phrases to attach both to VPs and NPs. So, the sentence "Mary saw John on the hill with a telescope" has five different structurally different parses, each with a somewhat different meaning (we'll look at them more carefully in a minute).

Basically we have two choices. One is to generate all the legal parses and let subsequent phases of the analysis sort them out or somehow to select one - possibly based on learned preferences based on examples. We will assume that we simply generate all legal parses.

Slide 9.2.57

Here are the various interpretations of our ambiguous sentence. In this one, both prepositional phrases are modifying the verb phrase. Thus, Mary is on the hill she used a telescope to see John.



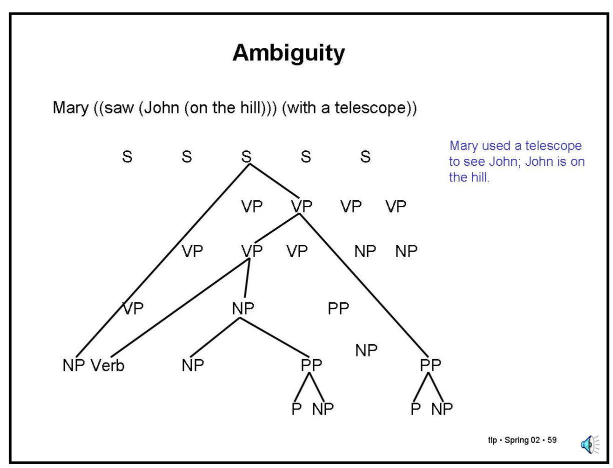


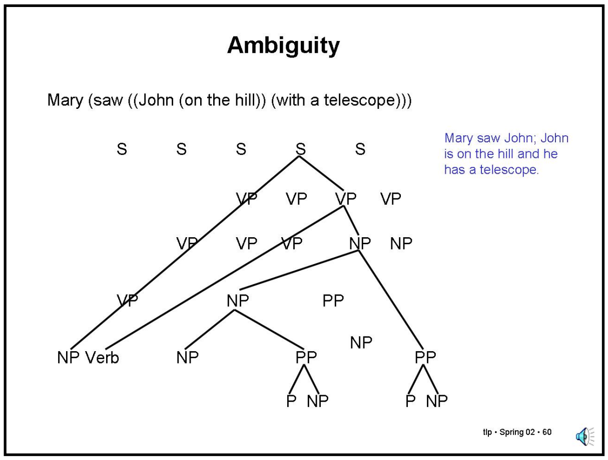
Slide 9.2.58

In this one, the telescope phrase has attached to the hill NP and so we are talking about a hill with a telescope. This whole phrase is modifying the verb phrase. Thus Mary is on the hill that has a telescope when she saw John.

Slide 9.2.59

In this one, the hill phrase is attached to John; this is clearer if you replace John with "the fool", so now Mary saw "the fool on the hill". She used a telescope for this, since that phrase is attached to the VP.





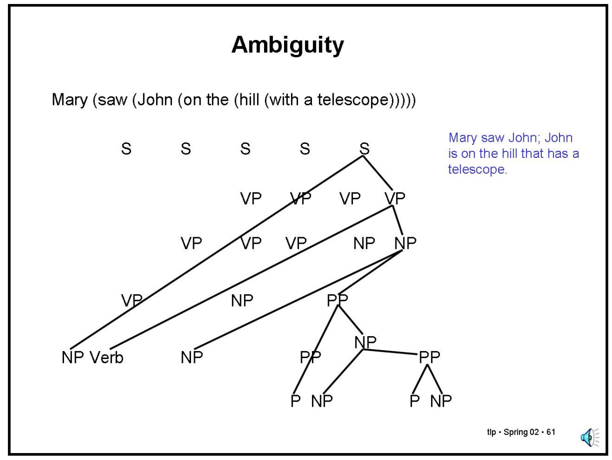
Slide 9.2.60

In this one, its the fool who is on the hill and who has the telescope that Mary saw.

Slide 9.2.61

Now its the fool who is on that hill with the telescope on it that Mary saw.

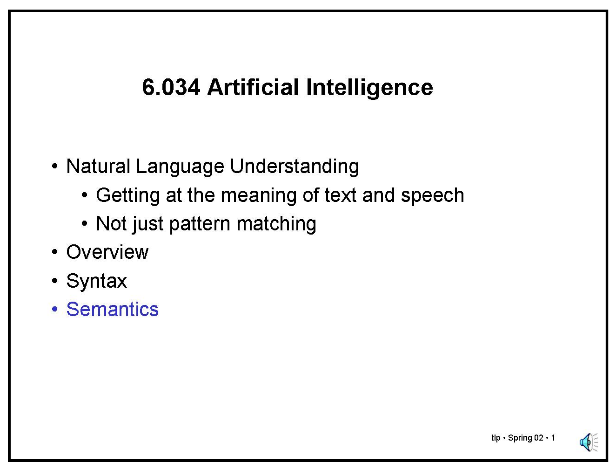
Note that the number of parses grows exponentially with the number of ambiguous prepositional phrases. This is a difficulty that only detailed knowledge of meaning and common usage can resolve.

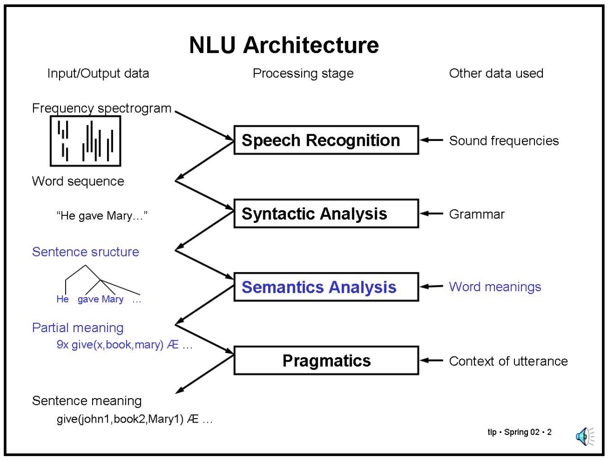


6.034 Notes: Section 9.3

Slide 9.3.1

Now, we move to consider the semantics phase of processing natural language.





Slide 9.3.2

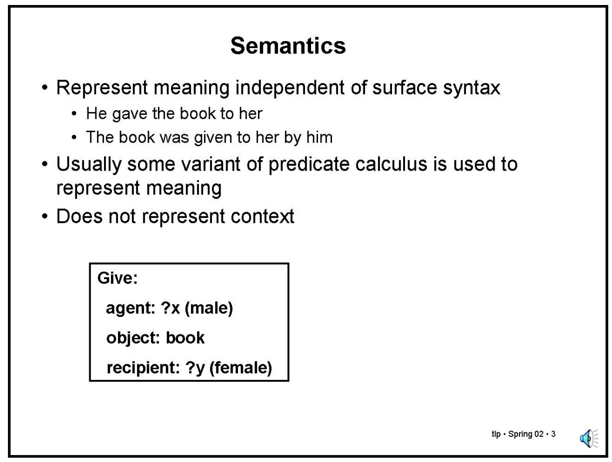
Recall that our goal is to take in the parse trees produced by syntactic analysis and produce a meaning representation.

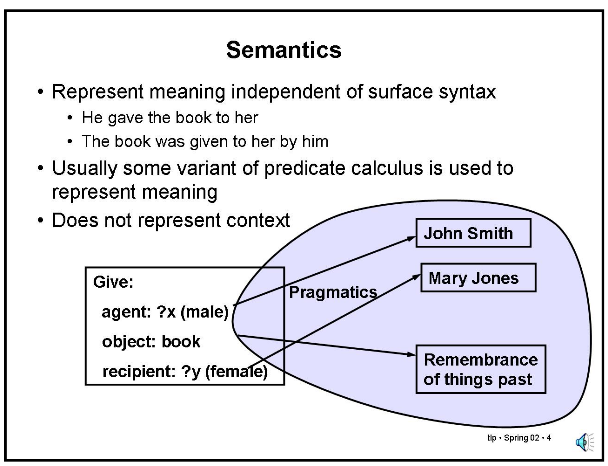
Slide 9.3.3

We want semantics to produce a representation that is somewhat independent of syntax. So, for example, we would like the equivalent active and passive voice versions of a sentence to produce equivalent semantics representations.

We will assume that the meaning representation is some variant of first order predicate logic. We will specify what type of variant later.

We have limited the scope of the role of semantics by ruling out context. So, for example, given the sentence "He gave her the book", we will be happy with indicating that some male gave the book to some female, without identifying who these people might be.



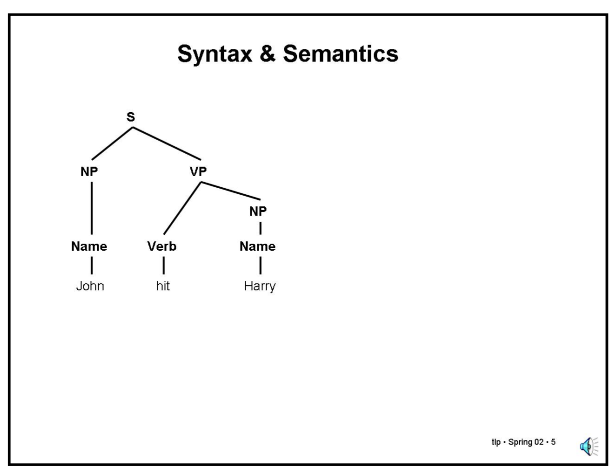


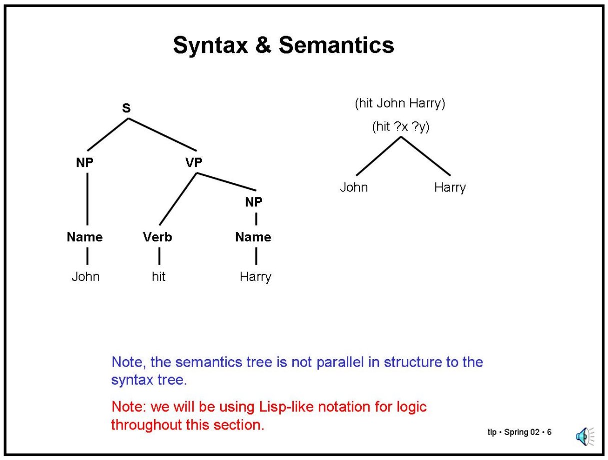
Slide 9.3.4

Part of the role of pragmatics, the next phase of processing, is to try to make those connections.

Slide 9.3.5

So, let's consider a very simple sentence "John hit Harry". We have here the simple parse tree. What should we expect the semantic representation to be?





Slide 9.3.6

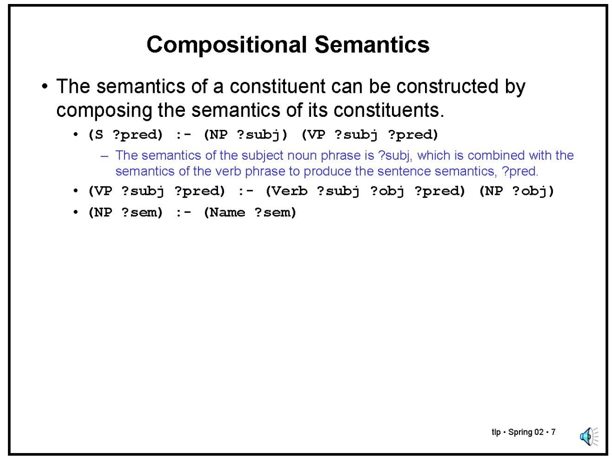
In this simple case, we might want something like this, where hit is a predicate and John and Harry are constant terms in the logical language. The key thing to notice is that even for this simple sentence the semantic structure produced is not perfectly parallel to the syntactic structure.

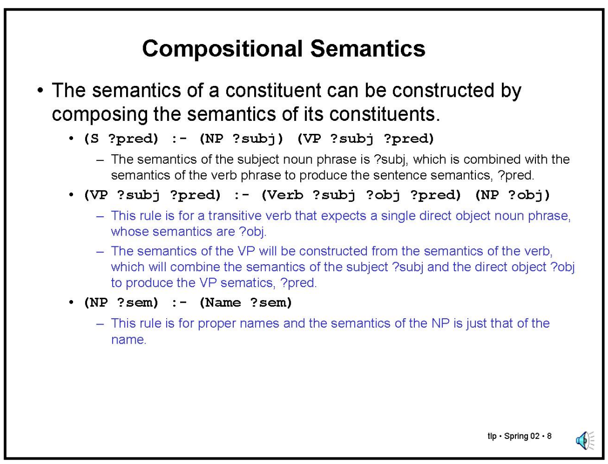
In this interpretation, the meaning of the verb is the center of the semantics. The meaning representation of the subject NP is embedded in the meaning representation of the verb phrase. This suggests that producing the semantics will not be a trivial variant of the parse tree. So, let's see how we can achieve this.

Slide 9.3.7

Our guiding principle will be that the semantics of a constituent can be constructed by composing the semantics of its constituents. However, the composition will be a bit subtle and we will be using feature values to carry it out.

Let's look at the sentence rule. We will be exploiting the "two way" matching properties of unification strongly here. This rule says that the meaning of the sentence is picked up from the meaning of the VP, since the second argument of the VP is the same as the semantics of the sentence as a whole. We already saw this in our simple example, so it comes as no surprise. Note also that the semantics of the subject NP is passed as the first argument of the VP (by using the same variable name).



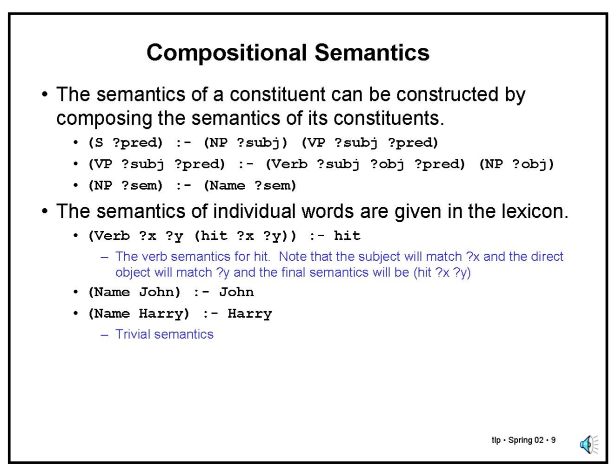


Slide 9.3.8

The VP has two arguments, the semantics of the subject NP (which will be an input) and the resulting semantics of the VP. In the VP rule, we see that the result semantics is coming from the Verb, which is combining the semantics of the subject and the object NPs to produce the result for the VP (and ultimately the sentence).

Slide 9.3.9

Let's look at the rule for a particular Verb. Note that the first two arguments are simply variables which are then included in the expression for the verb semantics, the predicate hit with two arguments (the subject and the object).



Compositional Semantics

- The semantics of a constituent can be constructed by composing the semantics of its constituents.
  - `(S ?pred) :- (NP ?subj) (VP ?subj ?pred)`
  - `(VP ?subj ?pred) :- (Verb ?subj ?obj ?pred) (NP ?obj)`
  - `(NP ?sem) :- (Name ?sem)`
- The semantics of individual words are given in the lexicon.
  - `(Verb ?x ?y (hit ?x ?y)) :- hit`
  - `(Name John) :- John`
  - `(Name Harry) :- Harry`
- The sentence: “John hit Harry”
  - `(backchain '(S ?sem 0 3))`
  - `?sem = (hit John Harry)`

tlp • Spring 02 • 10

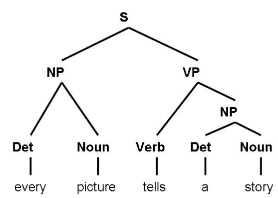
Slide 9.3.10

We can pull this altogether by simply calling backchain with the goal pattern for a successful parse. We will want to retrieve the value of the binding for ?sem, which is the semantics for the sentence.

Slide 9.3.11

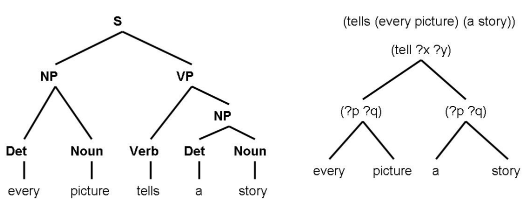
Let's look at a somewhat more complex example - "Every picture tells a story". Here is the syntactic analysis.

Syntax & Semantics



tlp • Spring 02 • 11

Syntax & Semantics



Note, the semantics tree is not parallel in structure to the syntax tree.

tlp • Spring 02 • 12

Slide 9.3.12

This is one possible semantic analysis. Note that it follows the pattern of our earlier example. The top-level predicate is derived from the verb and it includes as arguments the semantics of the subject and direct object.

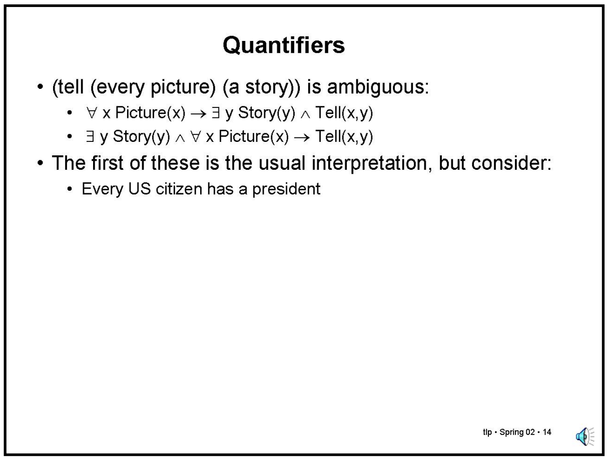
Slide 9.3.13

The only innovation in this grammar, besides the new words is a simple semantics for a noun phrase formed from a Determiner and a Noun - just placing them in a list. We can interpret the result as a quantifier operating on a predicate. But, what does this mean? It's certainly not legal logic notation.

Another Example

- The grammar
  - `(S ?pred) :- (NP ?subj) (VP ?subj ?pred)`
  - `(VP ?subj ?pred) :- (Verb ?subj ?obj ?pred) (NP ?obj)`
  - `(NP ?sem) :- (Name ?sem)`
  - `(NP (?detsem ?nsem)) :- (Det ?detsem) (Noun ?nsem)`
  - `(Verb ?x ?y (tells ?x ?y)) :- tells`
  - `(Noun picture) :- picture`
  - `(Noun story) :- story`
  - `(Det every) :- every`
  - `(Det a) :- a`
- The sentence: “Every picture tells a story”
  - `(backchain '(S ?sem 0 5))`
  - `?sem = (tell (every picture) (a story))`

tlp • Spring 02 • 13

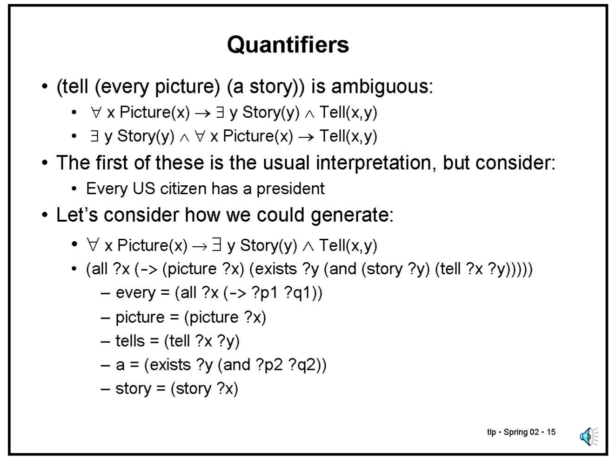


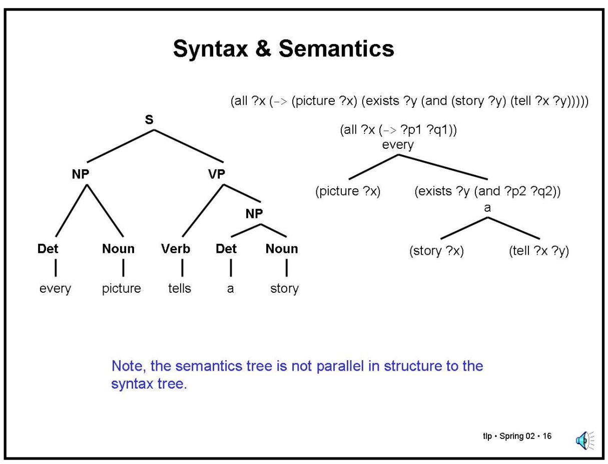
Slide 9.3.14

Furthermore, even if we are generous and consider this a legal quantified expression, then it's ambiguous - in the usual sense that "Every man loves a woman" is ambitious. That is, is there one story per picture or do all the pictures tell the same story.

Slide 9.3.15

Let's pick one of the interpretations and see how we could generate it. At the heart of this attempt is a definition of the meaning of the determiners "every" and "a", which now become patterns for universally and existentially quantified statements. Note also that the nouns become patterns for predicate expressions.



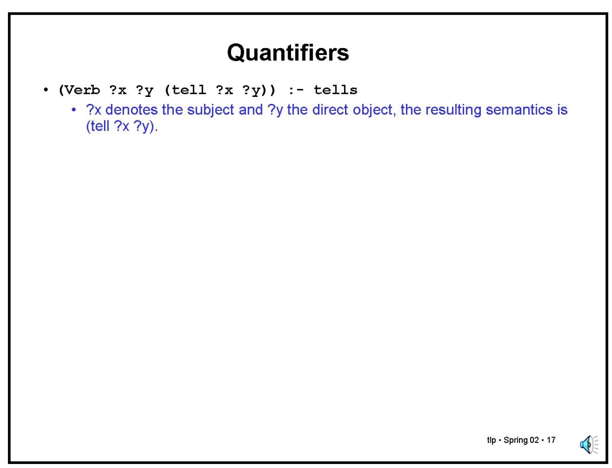


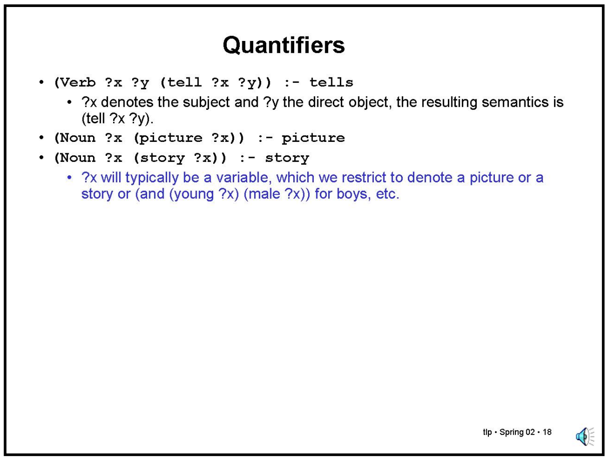
Slide 9.3.16

Our target semantic representation is shown here. Note that by requiring the semantics to be a legal logical sentence, we've had to switch the key role from the verb to the determiner. That is, the top node in the sentence semantics comes from the determiner, not the verb. The semantics of the verb is fairly deeply nested in the final semantics - but it still needs to combine the semantics of the subject and direct object NPs. Note, however, that it is incorporating them by using the quantified variable introduced by the determiners of the subject and object NPs.

Slide 9.3.17

Let's start with the definitions of the words. Here's the definition for the verb "tells". We have seen this before. It combines the semantics of the subject NP (bound to ?x) and the semantics of the object NP (bound to ?y) with the predicate representing the verb to produce the VP semantics.



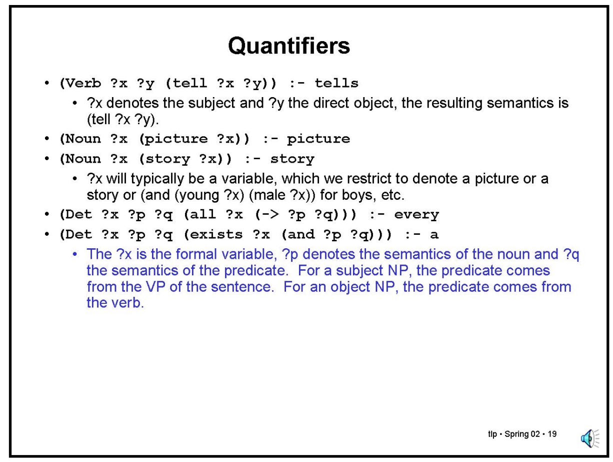


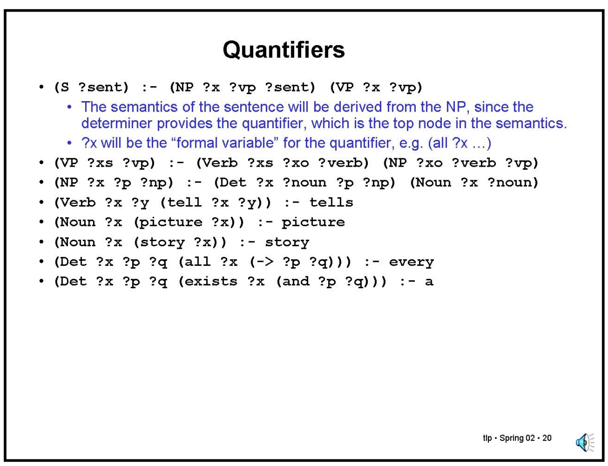
Slide 9.3.18

The nouns will be denoted by one of the quantified variables introduced by the quantifiers. The noun places a restriction on the entities that the variable can refer to. In this definition, the quantified variable will be bound to ?x and incorporated into the predicate representing the noun.

Slide 9.3.19

Finally, the determiners are represented by quantified formulas that combine the semantics derived from the noun with the semantics of the VP (for a subject NP) or of the Verb (for an object NP).





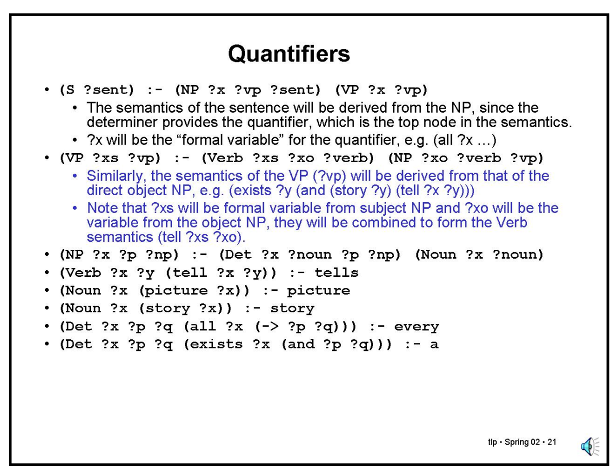
Slide 9.3.20

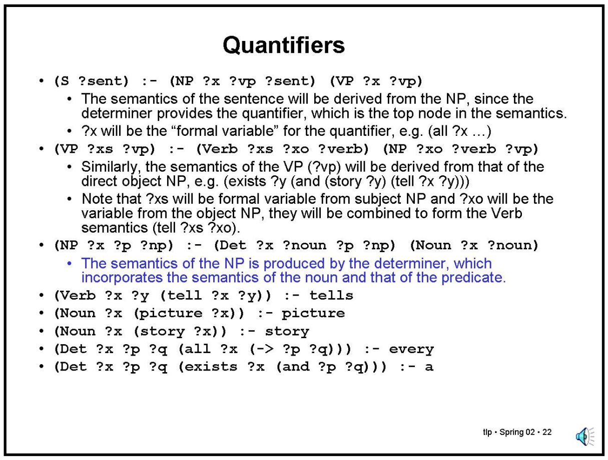
The new sentence (S) rule reflects the difference in where the top-level semantics is being assembled. Before, we passed the semantics of the subject NP into the VP, now we go the other way. The semantics of the VP is an argument to the subject NP.

Note that the variable ?x here will not be bound to anything, it is the variable that will be used as the quantified variable by the determiner's semantics.

Slide 9.3.21

The VP rule is analogous. The semantics of the Verb will combine a reference to the subject and object semantics (through their corresponding quantified variables) and the resulting semantics of the Verb will be combined into the semantics of the NP (which will ultimately be derived from the semantics of the determiner).





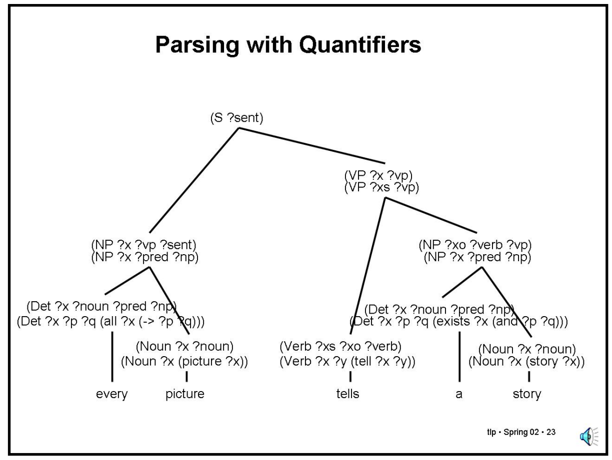
Slide 9.3.22

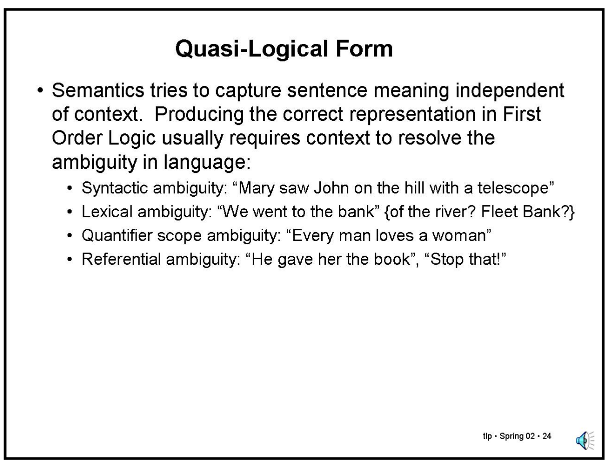
The NP rule in fact takes ?p, which will be the semantics of the Verb phrase and combine them with the semantics of the noun in the semantics of the Determiner.

Slide 9.3.23

Here we see how the parse works out. You have to follow the bindings carefully to see how it all works out.

What is remarkable about this is that we were able to map from a set of words to a first-order logic representation (which does not appear to be very similar) with a relatively compact grammar and with quite generic mechanisms.



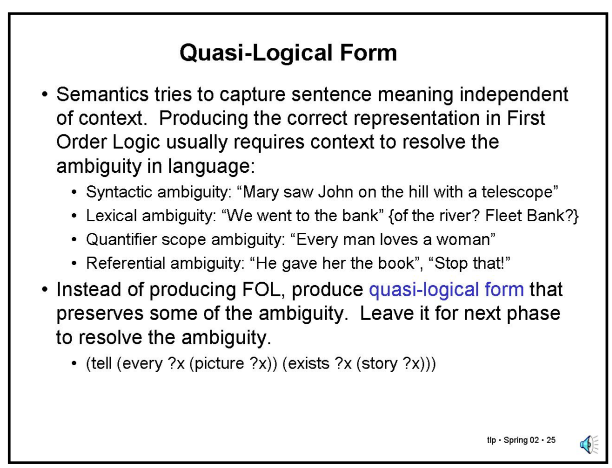


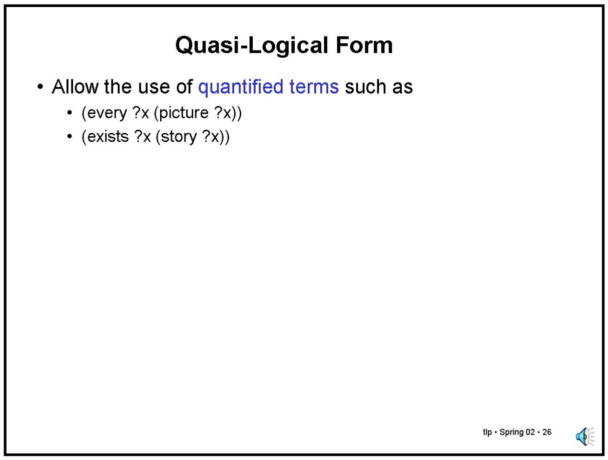
Slide 9.3.24

The quantified expression we produced in the previous example is unambiguous, as required to be able to write an expression in first order logic. However, natural language is far from unambiguous. We have seen examples of syntactic ambiguity, lexical and attachment ambiguity in particular, plus there are many examples of semantic ambiguity, for example, ambiguity in quantifier scope and ambiguity on who or what pronouns refer to are examples.

Slide 9.3.25

One common approach to semantics is to have it produce a representation that is not quite the usual logical notation, sometimes called **quasi-logical form**, that preserves some of the ambiguity in the input, leaving it to the pragmatics phase to resolve the ambiguities employing contextual information.



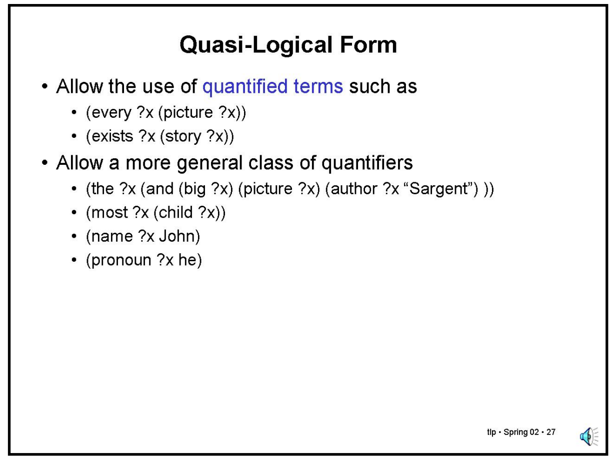


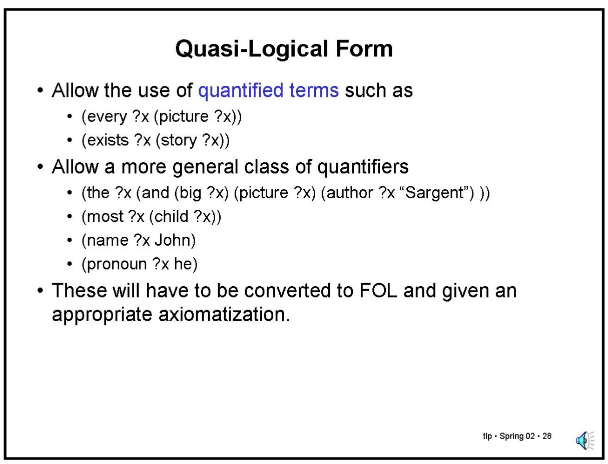
Slide 9.3.26

One common aspect of quasi-logical notation is the use of **quantified terms**. These terms indicate the nature of the intended quantification but do not specify the scope of the quantifier in the sentence and thus preserves the ambiguity in the natural language. Note that we are treating these quantified expressions as terms, and using them as arguments to functions and predicates - which is not legal FOL.

Slide 9.3.27

In quasi-logical notation, one also typically extends the range of available quantifiers to correspond more closely to the range of determiners available in natural language. One important case, is the determiner "the", which indicates a unique descriptor.



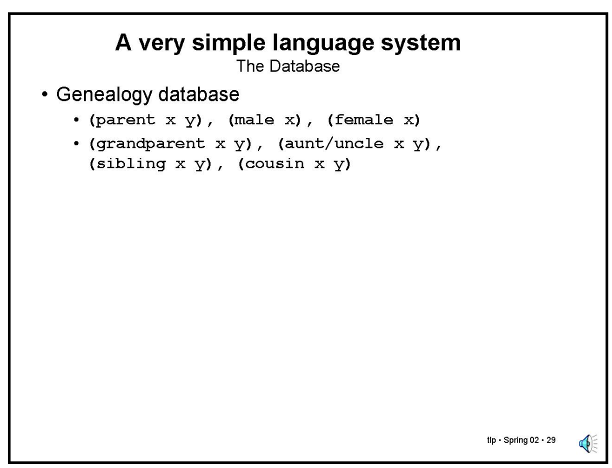


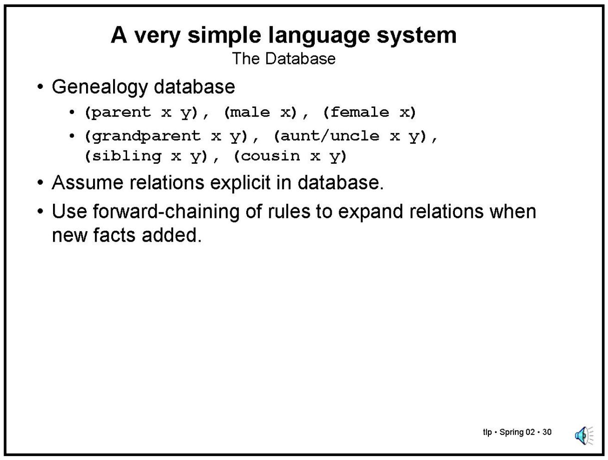
Slide 9.3.28

These quantified terms and generalized quantifiers will require conversion to standard FOL together with a careful axiomatization of their intended meaning before the resulting semantics can be used for inference.

Slide 9.3.29

Let's illustrate how the type of language processing we have been discussing here could be used to build an extremely simple database system. We'll assume that we want to deal with a simple genealogy domain. We will have facts in our database describing the family relationships between some set of people. We will not restrict ourselves to just the minimal set of facts, such as parent, male and female, we will also keep derived relationships such as grandparent and cousin.



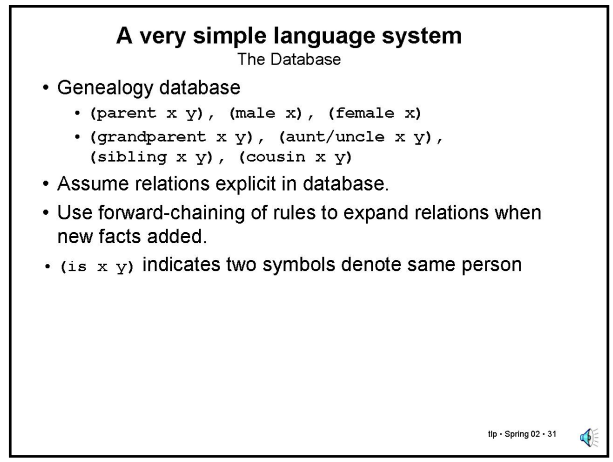


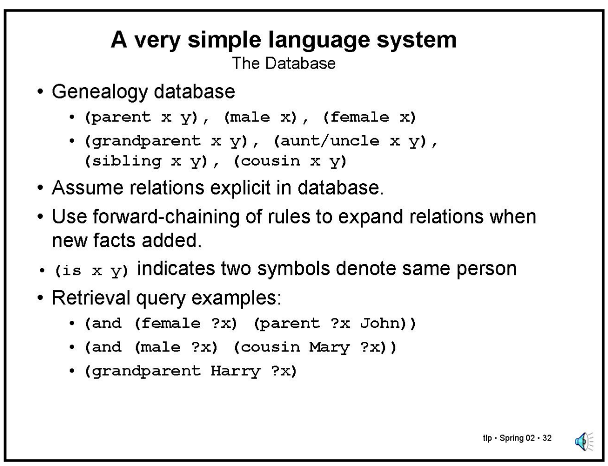
Slide 9.3.30

In fact, we will assume that all the relationships between people we know about are explicitely in the database. We can accomplish them by running a set of Prolog-like rules in forward chaining fashion whenever a new fact is added. We do this, rather than do deduction at retrieval time because of issues of equality, which we will discuss momentarily.

Slide 9.3.31

We will also allow assertions of the form (is x y) which indicate that two symbols denote the same person. We will assume that the forward chaining rules will propagate this equality to all the relevant facts. That is, we substitute equals for equals in each predicate, explicitely. This is not efficient, but it is simple.





Slide 9.3.32

We can now do very simple retrieval from this database of facts using our backchaining algorithm. We initialize the goal stack in backchaining with the query. If the query is a conjunction, we initialize the stack with all the conjuncts.

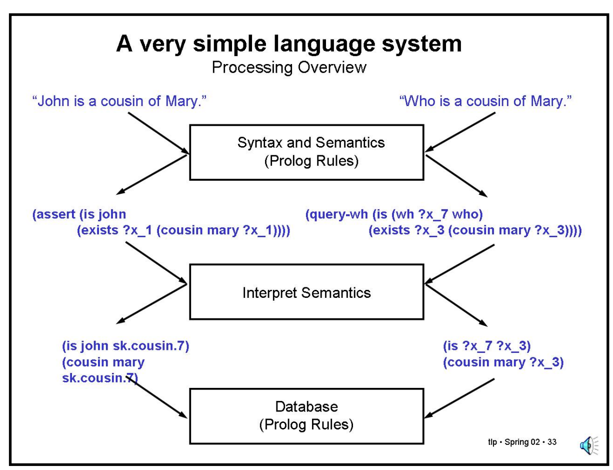
Slide 9.3.33

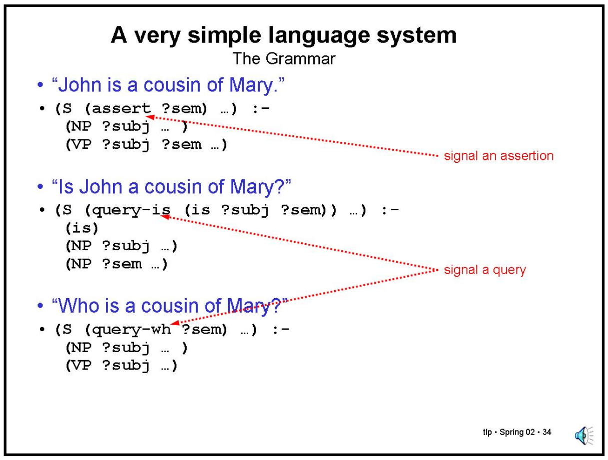
Here we see a brief overview of the processing that we will do to interact with the genealogy database.

We will be able to accept declarative sentences, such as "John is a cousin of Mary". These sentences will be processed by a grammar to obtain a semantic representation. This representation will then be interpreted as a set of facts to be added to the database.

We can also ask questions, such as "Who is a cousin of Mary". Our grammar will produce a semantic representation. The semantics of this type of sentence is converted into a database query and passed to the database.

Let's look in more detail at the steps of this process.





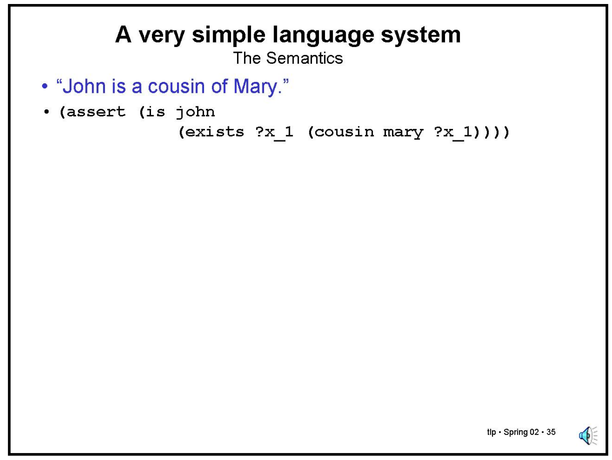
Slide 9.3.34

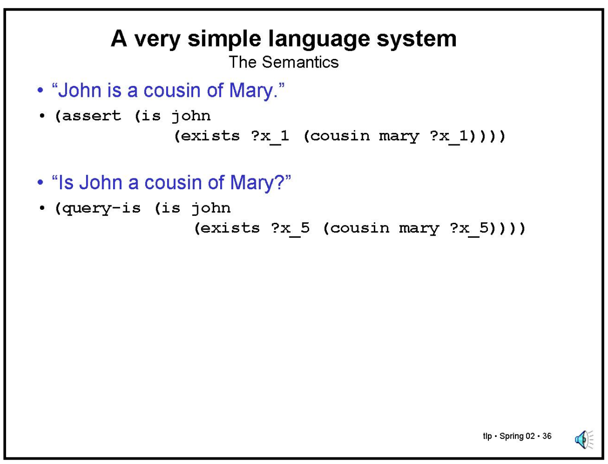
We will need a grammar built along the lines we have been discussing. One of the things the grammar does is classify the sentences into declarative sentences, such as "John is a cousin of Mary", which will cause us to assert a fact in our database, and questions, such as, "Is John a cousin of Mary" or "Who is a cousin of Mary", which will cause us to query the database.

Slide 9.3.35

Here we see one possible semantics for the declarative sentence "John is a cousin of Mary". The operation `assert` indicates the action to be taken. The body is in quasi-logical form; the quantified term `(exists ?x_1 (cousin mary ?x_1))` is basically telling us there exists a person that is in the cousin relationship to Mary. The outermost `is` assertion is saying that John denotes that person. This is basically interpreting this quasi-logical form as:

```
] x . (is John x) ^ (cousin Mary x)
```



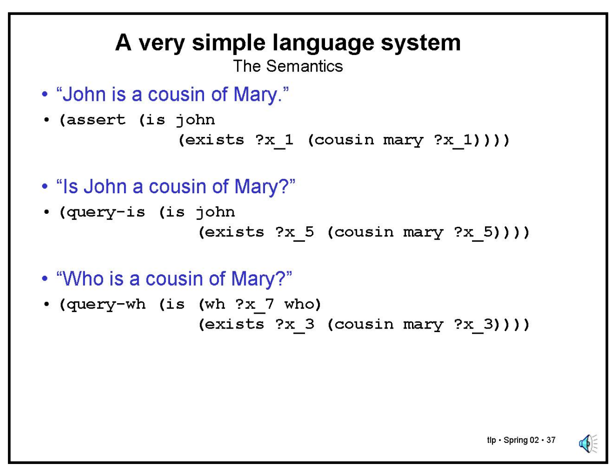


Slide 9.3.36

The semantics of the question "Is John a cousin of Mary?" is essentially identical to that of the declarative form, but it is prefixed by a query operation rather than an assertion. So, we would want to use this to query the database rather than for asserting new facts.

Slide 9.3.37

We can also have a question such as "Who is a cousin of Mary", which is similar except that John is replaced by a term indicating that we are interested in determining the value of this term.



A very simple language system
Using the Semantics

- "John is a cousin of Mary."
- `(assert (is john (exists ?x_1 (cousin mary ?x_1))))`
- Assign skolem constant for ?x_1, e.g. sk.cousin.7
- Convert body of exists into one or more facts
- Replace (exists ?x …) with skolem constant

tlp • Spring 02 • 38

Slide 9.3.38

Given the semantics, we have to actually decide how to add new facts and do the retrieval. Here we show an extremely simple approach that operates for these very simple types of queries (note that we are only using existentially quantified terms).

We are basically going to turn the assertion into a list of ground facts to add to the database. We will do this by skolemizing. Since we have only existentially quantified variables, this will eliminate all variables.

We replace the quantified terms with the corresponding skolem constant and we convert the body of the quantified term into a set of facts that describe the constant.

Slide 9.3.39

In this example, we get two new facts. One is from the outer `is` assertion which tells us that John denotes the same person as the skolem constant. The second fact comes from the body of the quantified term which tells us some properties of the person denote by the skolem constant.

A very simple language system
Using the Semantics

- "John is a cousin of Mary."
- `(assert (is john (exists ?x_1 (cousin mary ?x_1))))`
- Assign skolem constant for ?x_1, e.g. sk.cousin.7
- Convert body of exists into one or more facts
- Replace (exists ?x …) with skolem constant
- Add to the database:
  - `(is john sk.cousin.7)`
  - `(cousin mary sk.cousin.7)`

tlp • Spring 02 • 39

A very simple language system
Using the Semantics

- "Who is a cousin of Mary?"
- `(query-wh (is (wh ?x_7 who) (exists ?x_3 (cousin mary ?x_3))))`
- Convert body of exists into one or more additional conditions for query
- Replace (exists ?x …) with ?x
- Replace (wh ?y …) with ?y
- Retrieve from database:
  - `(and (is ?x_7 ?x_3) (cousin mary ?x_3))`
  - `?x_7/John`
  - `?x_3/sk.cousin.7`

tlp • Spring 02 • 40

Slide 9.3.40

We process the question in a similar way except that instead of using skolem constants we keep the variables, since we want those to match the constants in the database. When we perform the query, ?x_7 is bound to John as expected. In general, there may be multiple matches for the query, some may be skolem constants and some may be people names. We would want to return the specific names whenever possible.

Slide 9.3.41

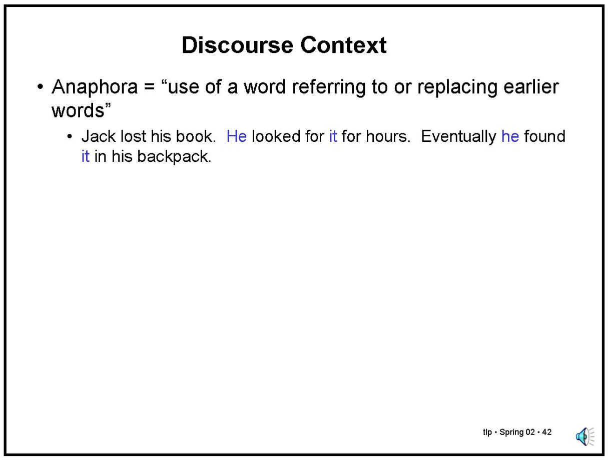
Here are some examples that show that this approach can be used to do a little inference above and beyond what is explicitely stated. Note that the assertions do not mention cousin, uncle, sister or sibling relations, those are inferred. So, we are going beyond what an Internet search engine can do, that is, pattern match on the presence of particular words.

This example has been extremely simple but hopefully it illustrates the flavor of how such a system may be built using the tools we have been developing and what could be done with such a system.

Some Examples

- Assertions
  - John is the father of Tom.
  - Mary is the female parent of Tom.
  - Bill is the brother of John.
  - Jim is the male child of Bill.
  - Jane is the daughter of John.
  - Mary is the mother of Jane.
- Questions
  - Is Jim the cousin of Tom?) Yes
  - Who is the uncle of Tom?) Bill
  - Is Bill the uncle of Tom?) Yes
  - Is Jane the sister of Tom?) Yes
  - Who is a child of Mary?) Tom
  - Who is a sibling of Tom?) Jane

tlp • Spring 02 • 41



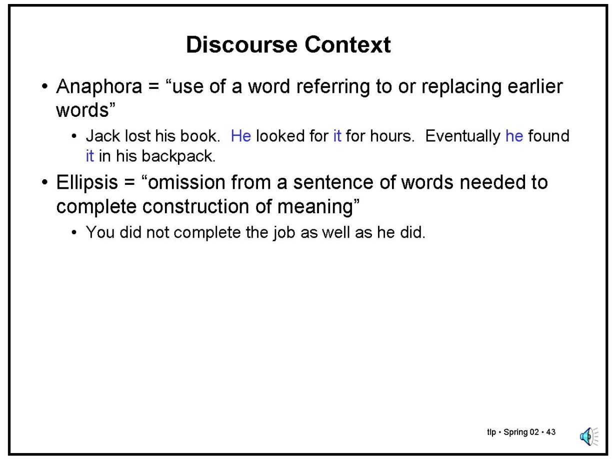
Slide 9.3.42

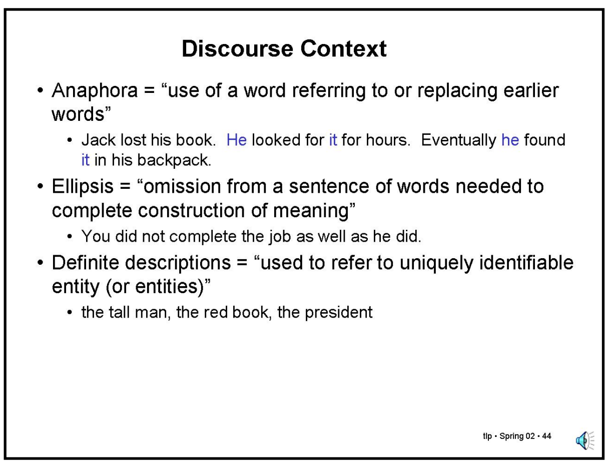
At this point, we'll touch briefly on a set of phenomena that are beyond the scope of pure semantics because they start dealing with the issue of context.

One general class of language phenomena is called **anaphora**. this includes pronoun use, where a word is used to refer to other words appearing either elsewhere in the sentence or in another sentence.

Slide 9.3.43

Another phenomenon is called **ellipsis**, when words or phrases are missing and need to be filled in from context. In this example, the phrase "complete the job" is missing from the enf of the second conjoined sentence.





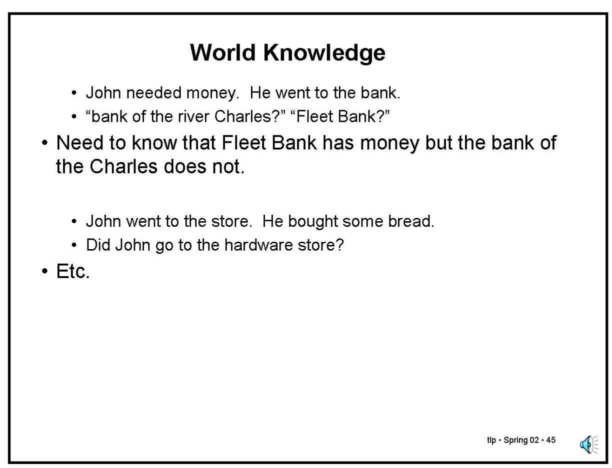
Slide 9.3.44

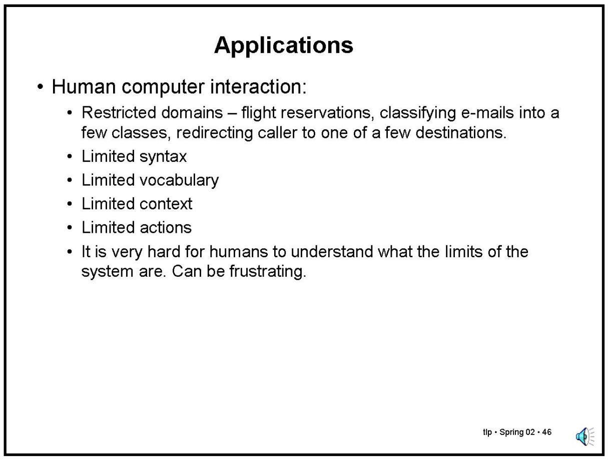
Another important mechanism in language is the use of **definite descriptions**, signaled by the determiner "the". The intent is that the listener be able to identify an entity previously mentioned or expected to be known.

All of these are linguistic mechanisms for incorporating context and require that a language understanding system that is engaged in an interaction with a human keep a context and be able to identify entities and actions in context based on the clues in the sentence. This is an area of active research and some systems with competence in this area have been built.

Slide 9.3.45

Even beyond conversational context, understanding human language requires access to the whole range of human knowledge. Even when speaking with a child, one assumes a great deal of "common sense" knowledge that computers are, as yet, sorely lacking in. The problem of language understanding at this point merges with the general problem of knowledge representation and use.





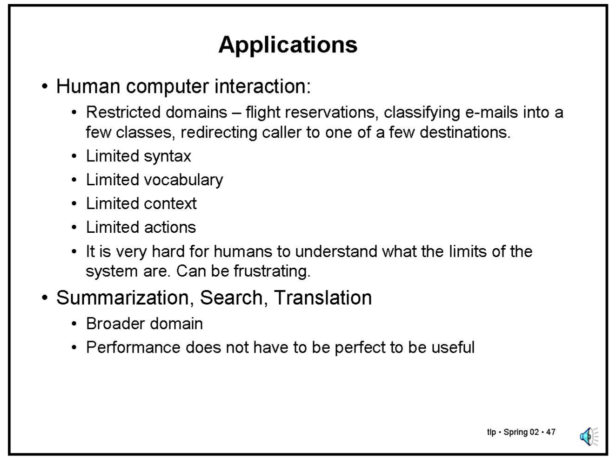
Slide 9.3.46

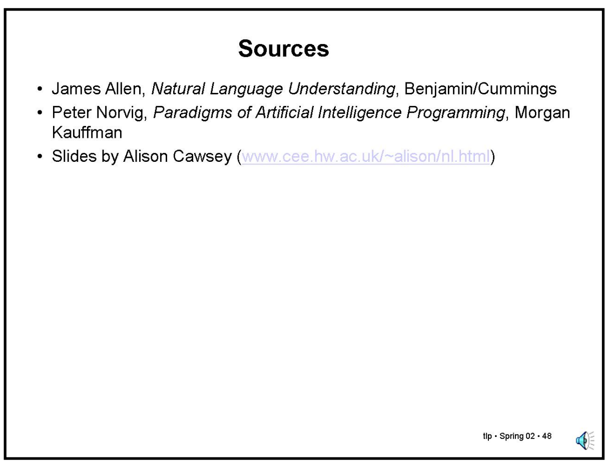
Real applications of natural language technology for human computer interfaces require a very limited scope so that the computer can get by with limited language skills and can have enough knowledge about the domain to be useful. However, it is difficult to keep people completely within the language and knowledge boundaries of the system. This is why the use of natural language interfaces is still limited.

Slide 9.3.47

There is, however, a rapidly increasing use of limited language processing in tasks that don't involve direct interaction with a human but do require some level of understanding of language. These tasks are characterized by situations where there is value in even limited capabilities, e.g. doing the first draft of a translation or a building a quick summary of a much longer news article.

I expect to see an explosion of applications of natural language technologies in the near future.





Slide 9.3.48

Here are some sources that were used in the preparation of these slides and which can serve as additional reading material.