Computational Linguistics

Shuly Wintner
shuly@cs.haifa.ac.il

What is this course about?

Natural language processing: A subfield of computer science, and in particular artificial intelligence, that is concerned with computational processing of natural languages, emulating cognitive capabilities without being committed to a true simulation of cognitive processes, in order to provide such novel products as computers that can understand everyday human speech, translate between different human languages, and otherwise interact linguistically with people in ways that suit people rather than computers.

What is this course about?

Computational linguistics: An approach to linguistics that employs methods and techniques of computer science. A formal, rigorous, computationally based investigation of questions that are traditionally addressed by linguistics: What do people know when they know a natural language? What do they do when they use this knowledge? How do they acquire this knowledge in the first place?

Applications of computational linguistics

- Machine translation
- Natural language interfaces to computer systems
- Speech recognition
- Text to speech generation
- Automatic summarization
- E-mail filtering
- Intelligent search engines

Example of an application: machine translation

The spirit is willing but the flesh is weak

The vodka is excellent but the meat is lousy

Example of an application: machine translation

From `http://babelfish.altavista.com/`,
using technology developed by SYSTRAN

Example of an application: machine translation

Language is one of the fundamental aspects of human behavior and is a crucial component of our lives. In written form it serves as a long-term record of knowledge from one generation to the next. In spoken form it serves as our primary means of coordinating our day-to-day behavior with others. This book describes research about how language comprehension and production work.

Example of an application: machine translation

Il linguaggio è una delle funzioni fondamentali di comportamento umano ed è un componente cruciale delle nostre vite. Nella forma scritta serve da record di lunga durata di conoscenza da una generazione al seguente. Nella forma parlata serve da nostri mezzi primari di coordinazione del nostro comportamento giornaliero con altri. Questo libro descrive la ricerca circa come la comprensione di una lingua e la produzione funzionano.

Example of an application: machine translation

The language is one of the fundamental functions of human behavior and is a crucial member of our screw. In the written shape servants from record of long duration of acquaintance from one generation to following. In the shape speech she serves from our primary means of coordination of our every day behavior with others. This book describes the search approximately as the understanding of a language and the production work.

Comparison

Language is one of the fundamental aspects of
The language is one of the fundamental functions of
human behavior and is a crucial component of our lives
human behavior and is a crucial member of our screw

Comparison

In written form it serves as a long-term record
In the written shape servants from record of long duration
of knowledge from one generation to the next
of acquaintance from one generation to following

Comparison

This book describes research about how
This book describes the search approximately as
language comprehension and production work
the understanding of a language and the production work

Example of an application: question answering

From `http://www.ask.com/` and `http://www.ajkids.com/`

who was the second president of the United States?

who was the US president following Washington?

Why are the results so poor?

- Language understanding is complicated
- The necessary knowledge is enormous
- Most stages of the process involve *ambiguity*
- Many of the algorithms are computationally intractable

What kind of knowledge is required?

- Phonetic and phonological knowledge
- Morphological knowledge
- Syntactic knowledge
- Semantic knowledge
- Pragmatic knowledge
- Discourse knowledge
- World knowledge

What kind of knowledge is required?

- Phonetic and phonological knowledge
- Morphological knowledge
- Syntactic knowledge
- Semantic knowledge
- Pragmatic knowledge
- Discourse knowledge
- World knowledge

Phonetics and phonology

Phonetics studies the sounds produced by the vocal tract and used in language, including the physical properties of speech sounds, their perception and their production

Phonology studies the module of the linguistic capability that relates to sound, abstracting away from their physical properties. Defines an inventory of basic units (*phonemes*), constraints on their combination and rules of pronunciation

Problems in phonological processing

Homophones (homonyms): words that are pronounced alike but are different in meaning or derivation or spelling:

weak — week;

to — too — two;

haqala — ha-qala — ha-kala

Free variation: alternation of sounds with no change in meaning:

the different pronunciations of the guttural sounds in Hebrew

Problems in phonological processing

Allophones: variants of phonemes that are in complementary distribution:

l itt l e

Phonotactic constraints: restrictions on the distribution (occurrence) of phonemes with respect to one another:

hitbatte — hictallem

What kind of knowledge is required?

- Phonetic and phonological knowledge
- Morphological knowledge
- Syntactic knowledge
- Semantic knowledge
- Pragmatic knowledge
- Discourse knowledge
- World knowledge

Morphology

Morphology studies the structure of words.

Morpheme: a minimal sound-meaning unit. Can either be *bound* (not a word) or *free* (word).

Free morphemes: book, histapper

Bound morphemes: book s, histappr u

Affix: a morphemes which is added to other morphemes, especially roots or stems.

suffixes follow the root/stem
prefixes precedes the root/stem
infixes are inserted into the root/stem

Problems in morphological processing

Derivational morphology: words are constructed from roots (or stems) and derivational affixes:

inter+national → international

international+ize → internationalize

internationalize+ation → internationalization

Inflectional morphology: inflected forms are constructed from base forms and inflectional affixes: bayt+i → beiti

Ambiguity: $mnh

What kind of knowledge is required?

- Phonetic and phonological knowledge
- Morphological knowledge
- Syntactic knowledge
- Semantic knowledge
- Pragmatic knowledge
- Discourse knowledge
- World knowledge

Syntax

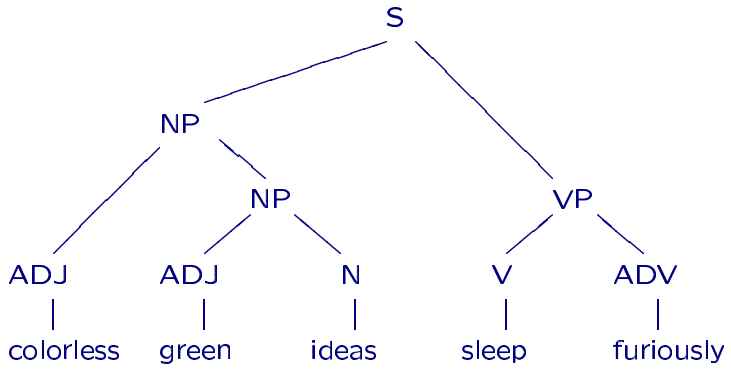
Natural language sentences have *structure*.

Young green frogs sleep quietly

Colorless green ideas sleep furiously

Furiously sleep ideas green colorless

Syntax



Problems of syntactic processing

Expressiveness: what formalism is required for describing natural languages?

Parsing: assigning structure to grammatical strings, rejecting ungrammatical ones.

- top–down vs. bottom–up
- right to left vs. left to right
- chart based vs. backtracking

Problems of syntactic processing

Ambiguity:

I saw the spy with the brown hat
I saw the bird with the telescope
I saw the spy with the telescope

Control:

Kim asked Sandy to call the plumber
Kim promised Sandy to call the plumber

Coordination:

This book describes research about how language comprehension and production work

What kind of knowledge is required?

- Phonetic and phonological knowledge
- Morphological knowledge
- Syntactic knowledge
- Semantic knowledge
- Pragmatic knowledge
- Discourse knowledge
- World knowledge

Semantics

Semantics assigns *meanings* to natural language utterances.

A semantic representation must be precise and unambiguous.

A good semantics is *compositional*: the meaning of a phrase is obtained from the meanings of its subphrases.

Problems of semantic processing

Word sense ambiguity: book; round; about; pgi$a

Scope ambiguity:

every student hates at least two courses
every student doesn't like math

Problems of semantic processing

Co-reference and anaphora:

Kim went home after she robbed the bank
After she robbed the bank, Kim went home
In the next few paragraphs, some preliminary constraints are suggested and problems with them are discussed.
Language is one of the fundamental aspects of human behavior. In written form it serves as a long-term record of knowledge.

VP anaphora: Kim loves his wife and so does Sandy.

What kind of knowledge is required?

- Phonetic and phonological knowledge
- Morphological knowledge
- Syntactic knowledge
- Semantic knowledge
- Pragmatic knowledge
- Discourse knowledge
- World knowledge

Pragmatics

Pragmatics is the study of how more gets communicated than is said.

Presupposition: the presuppositions of a sentence determine the class of contexts in which the sentence can be felicitously uttered:

The current king of France is bald

Kim regrets that he voted for Gore

Sandy's sister is a ballet dancer

Pragmatics

Implicature: what is conveyed by an utterance that was not explicitly uttered:

– How old are you? – Closer to 30 than to 20.

I have two children.

Could you pass the salt?

Non-literal use of language: metaphor, irony etc.

Pragmatics

Speech acts: the illocutionary force, the communicative force of utterances, resulting from the function associated with them:

I'll see you later

- prediction: I predict that I'll see you later
- promise: I promise that I'll see you later
- warning: I warn you that I'll see you later

I sentence you to six months in prison

I swear that I didn't do it

I'm really sorry!

What kind of knowledge is required?

- Phonetic and phonological knowledge
- Morphological knowledge
- Syntactic knowledge
- Semantic knowledge
- Pragmatic knowledge
- Discourse knowledge
- World knowledge

Discourse

A discourse is a sequence of sentences. Discourse has structure much like sentences do. Understanding discourse structure is extremely important for dialog systems.

An example dialog:

When does the train to Haifa leave?

There is one at 2:00 and one at 2:30.

Give me two tickets for the earlier one, please.

Problems of discourse processing

Non-sentential utterances: aha; to Haifa; the last one

Cross-sentential anaphora

Reference to non-NPs: Kim visited the University of Haifa.

It changed her life.

She does it every year.

It really surprised Sandy.

It was summer then.

What kind of knowledge is required?

- Phonetic and phonological knowledge
- Morphological knowledge
- Syntactic knowledge
- Semantic knowledge
- Pragmatic knowledge
- Discourse knowledge
- World knowledge

World knowledge

– Is the train to Haifa late? – It left Tel Aviv at 8:30.

Bill Clinton left for Vietnam today. This is the last foreign visit of the American president.

Processing Hebrew

- The script
- Writing direction
- Deficiencies of the Hebrew writing system
- Richness of the morphology
- Root-and-pattern word formation
- Lack of linguistic resources

Infrastructure for processing language

- Lexicons
- Dictionaries
- Morphological analyzers and generators
- Part-of-speech taggers
- Shallow parsers
- Syntactic analyzers
- Computational grammars

Hebrew processing: the state of the art

- Lexicons
- Dictionaries
- Morphological analyzers and generators
- Part-of-speech taggers
- Shallow parsers
- Syntactic analyzers
- Computational grammars

Conclusions

- Natural languages are complex
- Applications which require deep linguistic knowledge still do not perform well
- Applications which can rely on shallow knowledge or on statistical approaches perform better
- Hebrew poses additional problems for language processing
- To build Hebrew language applications, essential linguistic resources must be developed

Structure of the course

Morphology

- introduction to morphology: word structure
- inflections and derivations
- finite-state automata
- finite-state transducers

Structure of the course

Syntax

- introduction to syntax: the structure of natural languages
- context-free grammars: grammars, forms, derivations, trees, languages
- parsing: top–down, CYK algorithm, Earley algorithm, bottom–up chart parsing
- the complexity of natural languages
- the limitations of CFGs
- unification grammars: feature structures and unification

Structure of the course

Other topics

- As time permits

Practicalities

Textbook: Nothing mandatory or even recommended. Some of the material can be found in Daniel Jurafsky and James H. Martin, *Speech and Language Processing*, Prentice-Hall, 2000.

Grading: 4–6 home assignments (approximately 33% of the final grade); mid-term exam (33%); final exam (33%)

Attendance: Optional but highly recommended.

Morphology

Morphology is the area of linguistics which studies the structure of words.

Almost all natural language applications require some processing of words: lexicon lookup, morphological analysis and generation, part-of-speech determination etc.

In order to implement such functions, it is necessary to understand which morphological processes take place in a variety of languages.

Why look at many languages?

Example

hem dibbru koll ha-layla

Observations:

- dibbru is third person, plural, past form of the verb *dibber*
- this form is obtained by concatenating the suffix [u] to the base [dibber]
- in the inflected form *dibbru*, the vowel [e] of the base [dibber] is reduced to a schwa. This reduction is mandatory, as [dibberu] is ungrammatical.

Example

These simple observations shed light on a variety of issues:

- What information is encoded by morphology?

  In the example, morphology encodes details such as person, number and tense.

- How does morphology encode information?

  In the example, the final form is obtained by concatenating an affix (which is not a word) to the end of a base (which might be a word).

- Interaction of morphology and phonology

  In the example, the vowel [e] is shortened to a schwa.

Structure of this part of the course

- Typology of languages
- Inflection and derivation
- What information is encoded by morphology
- How morphology encodes information
  - concatenation, infixation, circumfixation, root and pattern, reduplication
- Interaction of morphology and phonology

Typology of languages

Isolating : no bound forms. Example: Mandarin Chinese

Agglutinative : bound forms occur and are arranged in the word like beads on a string. Example: Turkish

Polysynthetic : elements that often occur as separate words in other languages (such as arguments of the verb) are expressed morphologically. Example: Yupik (central Alaska)

Inflectional : distinct features are merged into a single bound form. Example: Latin

Isolating languages

No bound forms. Example: Mandarin Chinese

gǒu bú ài chī qīngcài
dog not like eat vegetable

Can mean any of the following (inter alia):

- the dog doesn't like to eat vegetables
- the dog didn't like to eat vegetables
- the dogs don't like to eat vegetables
- the dogs didn't like to eat vegetables
- dogs don't like to eat vegetables

Agglutinative languages

Beads on a string. Example: Turkish

çöplüklerimizdekiledenmiydi

çöp lük ler imiz de ki ler den mi y di
garbage Aff Pl 1p/Pl Loc Rel Pl Abl Int Aux Past
"was it from those that were in our garbage cans?"

"h-mi-$e-b-paxeinu?"

Polysynthetic languages

Morphology encodes units that are usually considered syntactic (as in noun incorporation). Example: Yupik

qayá:liy'u:l'u:n'i

qayá: li y'u: l'u: n'i
kayaks make excellent he Past
"he was excellent at making kayaks"

"The grammar is in the morphology"

Inflectional languages

Portmanteau morphemes: a single morpheme can encode various bits of information. Example: Latin

amó

am ó
love 1p/Sg/Pres/Indicative/Active

Inflections and derivations

Inflectional morphology takes as input a word and outputs a form of the same word appropriate to a particular context.

Example: [dibber] $\Rightarrow$ [dibbru]

The output is appropriate to a context in which the subject is third person plural and the tense is past.

Hence: words have *paradigms*, defining all possible inflected forms of a word. Words which belong to the same paradigm are all *inflected forms* of a single *lexeme*.

Inflections and derivations

Derivational morphology takes as input a word and outputs a different word that is derived from the input. This is also called *word formation*.

Example: establish+ment+ary+an+ism

Example: hexlit $\rightarrow$ haxlata $\rightarrow$ hexleti $\rightarrow$ hexletiyut

Inflections and derivations - distinctive criteria

- Inflection does not change the part-of-speech, derivation might.

  haxlata–haxlatot; haxlata–hexleti

- Inflection is sometimes required by the syntax, derivation never is.

- If a language marks an inflectional category, it marks it on all appropriate words. In other words, the relation denoted by inflectional morphology is *productive*.

haxlata	–	haxlatot	haxlata	–	hexleti
hapgana	–	hapganot	hapgana	–	*hepgeni

Verbal morphology

Verbs specify the number (and type) of arguments they may take. In many languages, morphological devices modify these lexically specified markings.

Example: passivization (Latin)

puer Cicerōnem laudat
*boy Cicero praise/3/Sg/Pres/Ind/***Act**
"the boy praises Cicero"

Cicerōnem laudātur
*Cicero praise/3/Sg/Pres/Ind/***Pass**
"Cicero is praised"

Example: causativization

napal → hippil; nasa& → hissi&

Verbal morphology

Verbs are commonly marked with indications of the time at which the situations denoted by them occurred, or the state of completion of the situation. Such markers encode *tense* and *aspect*, respectively.

Example: Latin

vir Cicerōnem laudābō
*man Cicero praise/3/Sg/***Future***/Ind*
"the man will praise Cicero"

vir Cicerōnem laudāvit
*man Cicero praise/3/Sg/***Perf***/Ind*
"the man has praised Cicero"

Verbal morphology

In many languages the verb must *agree* on person, number, gender or other features with one or more of its arguments.

Example:

The princess kisses the frog
∗*The princess kiss the frog*

hem dibbru koll ha-layla
∗*hem dibbra koll ha-layla*

In some languages (e.g., Georgian and Chicheŵa) verbs agree not only with their subjects but also with their objects.

Nominal morphology

Inflectional categories for nouns (and adjectives) include

- number (singular, plural, dual)
- case (marking various kinds of semantic function)
- gender (feminine, masculine, neuter)

Latin has five cases: nominative, genitive, dative, accusative, ablative.

Finnish has fourteen different cases!

Example: the inflection paradigm of the noun *magnus* (big) in Latin.

The inflection paradigm of Latin magnus

		masculine	feminine	neuter
sing.	nom	magn+**us**	magn+**a**	magn+**um**
	gen	magn+**ī**	magn+**ae**	magn+**ī**
	dat	magn+**ō**	magn+**ae**	magn+**ō**
	acc	magn+**um**	magn+**am**	magn+**um**
	abl	magn+**ō**	magn+**ā**	magn+**ō**
plur.	nom	magn+**ī**	magn+**ae**	magn+**a**
	gen	magn+**ōrum**	magn+**ārum**	magn+**ōrum**
	dat	magn+**īs**	magn+**īs**	magn+**īs**
	acc	magn+**ōs**	magn+**ās**	magn+**a**
	abl	magn+**īs**	magn+**īs**	magn+**īs**

Nominal morphology

Many languages distinguish between two or three grammatical genders: feminine, masculine and neuter.

In some languages, such as the Bantu languages, more detailed gender classes exist.

Example: Swahili has inflection affixes for humans, thin objects, paired things, instruments and extended body parts, inter alia.

Adjectival morphology

Many languages express comparison of adjectives morphologically.

Example: Welsh

gwyn	gwynn+**ed**	gwynn+**ach**	gwynn+**af**
white	as white	whiter	whitest
teg	tec+**ed**	tec+**ach**	tec+**af**
fair	as fair	fairer	fairest

Derivational morphology

In general, derivational morphology is not as productive as inflectional morphology.

Nominalization: destroy → destruction; $amar → $mira; pittex → pittux; hiskim → heskem

Deverbal adjectives: drink → drinkable; nazal → nazil

Denominalized adjectives: $ulxan → $ulxani

Adjective nominalization: grammatical → grammaticality; nadir → ndirut

Negation: able → unable; xuti → 'alxuti

Compounding

In contrast to derivations and inflections, where affixes are attached to a stem, in compounding two or more lexemes' stems are joint together, forming another lexeme.

Example: policeman; newspaper; beit seper; ypat &einaym

Both lexemes might undergo modification in the process.

In German, the concatenation is expressed in the orthography:

lebensversicherungsgesellschaftsangestellter

leben	s	versicherung	s	gesellschaft	s	angestellter
life		*insurance*		*company*		*employee*

What are morphemes?

In order to know what morphemes are, it is useful to check in what ways they are expressed.

The simplest model of morphology is the situation where a morphologically complex word can be analyzed as a series of morphemes concatenated together.

An example: Turkish. Not only is Turkish morphology exclusively concatenative; in addition, all affixes are suffixes. Turkish words are of the form *stem suffix**.

çöplüklerimizdekiledenmiydi

çöp	lük	ler	imiz	de	ki	ler	den	mi	y	di
garbage	*Aff*	*Pl*	*1p/Pl*	*Loc*	*Rel*	*Pl*	*Abl*	*Int*	*Aux*	*Past*

What are morphemes?

Linear concatenation is not the only way in which languages put morphemes together. Affixes may also attach as *infixes* inside words.

Example: Bontoc (Philippines)

fikas	→	f-**um**+ikas
strong		*be strong*

kilad	→	k-**um**+ilad
red		*be red*

fusul	→	f-**um**+usul
enemy		*be an enemy*

What are morphemes?

In the Bontoc case the infix must be placed after the first consonant of the word to which it attaches.

In general, the placement of infixes is governed by prosodic principles.

Example: Ulwa (Nicaragua)

suu+**ki**-lu	my dog
suu+**ma**-lu	your (Sg) dog
suu+**ka**-lu	his/her/its dog
suu+**ni**-lu	our (inclusive) dog
suu+**ki**+**na**-lu	our (exclusive) dog
suu+**ma**+**na**-lu	your (Pl) dog
suu+**ka**+**na**-lu	their dog

What are morphemes?

Some languages exhibit *circumfixes*, affixes which attach discontinuously around a stem.

Example: German participles

säuseln	**ge**+säusel+**t**
brüsten	**ge**+brüst+**et**
täuschen	**ge**+täusch+**t**

What are morphemes?

In contrast to processes of attaching an affix to a stem, there exist also nonsegmental morphological processes. A typical example is the Semitic *root and pattern* morphology.

Example: Hebrew *binyanim*

_a_a_, ni__a_, _i__el, _u__a_, hi__i_, hu__a_, hit_a__e_.

What are morphemes?

Another nonsegmental process is *reduplication*.

Example: Indonesian

orang → orang+orang
man *men*

Sometimes only part of the word is duplicated, as in Yidin (Australia) plural:

mulari → mula+mulari
man *men*

gindalba → gindal+gindalba
lizard *lizards*

So, what are morphemes?

In its most general definition, a morpheme is an ordered pair $\langle \text{cat}, \text{phon} \rangle$, where cat is the morphological category expressed by the morpheme (for example, its syntactic and semantic features), and phon represents its phonological form, including the ways in which it is attached to its stem.

Example:

$\langle (Adj \rightarrow N, \text{“state of”}), ([ut], \textit{suffix}) \rangle$ nadir → ndirut

$\langle (root \rightarrow V, causative), (\_i\_\_e\_) \rangle$ g.d.l → giddel

What are words, then?

A morpheme is a pairing of syntactic/semantic information with phonological information. In the same way, it is useful to assume that words have dual structures: phonological and morphological. The two structures are not always isomorphic.

It is a fairly traditional observation in morphology that there are really two kinds of words from a structural point of view: phonological words and syntactic words. These two notions specify overlapping but not identical sets of entities. furthermore, the orthographic word might not correspond to any of these.

What information should a morphological analyzer produce?

The answer depends on the application:

Sometimes it is sufficient to know that dibbru is an inflected form of dibber; sometimes morphological information is needed, either as a list of features (dibbru is third person, plural, past form of the verb dibber) or as a structure tree; sometimes it is better to produce a list of phonemes without determining word boundaries. For some applications, the root d.b.r might be needed.

Morphotactics

Morphotactics investigates the constraints imposed on the order in which morphemes are combined.

Various kinds of such constraints are known.

Example:

teva& $\rightarrow$ tiv&i $\rightarrow$ tiv&iyut $\rightarrow$ &al-tiv&iyut but
$*$tiv&iyut-&al; $*$&al-tiv&uti

Morphotactics

Types of constraints:

- Constraints on the type of the affix: &al is a prefix, ut is a suffix
- Syntactic constraints: [i] converts a noun to an adjective; [ut] converts an adjective to a noun
- Other constraints: in English, "Latin" affixes are attached before "native" ones:

  non+im+partial non+il+legible
  $*$in+non+partial $*$in+non+legible

Phonology

Ideally, the task of a morphological analysis system would be to break the word down to its component morphemes and determine the meaning of the resulting decomposition.

Things are not that simple because of the often quite drastic effects of phonological rules. A great deal of the effort in constructing computational models of morphology is spent on developing techniques for dealing with phonological rules.

Since most computational analyses of morphology assume *written* input, phonological rules are often confused with orthographic ones.

Phonology

Orthographic rules often do not correspond to phonological rules.

An orthographic rule that does not correspond to any phonological rule:

city+s $\rightarrow$ cities (and not $*$citys)

bake+ing $\rightarrow$ baking (and not $*$bakeing)

Phonology

A phonological rule (changing [aj] to [i]) is not reflected in the orthography:

divine+ity $\rightarrow$ divinity

A phonological rule (stress shift) is not reflected in the orthography:

grammátical $\rightarrow$ grammaticálity

Phonology

Examples of phonological rules

English: [n] changes to [m] before a labial consonant:

impossible; **im**pose; **im**mortal

Finnish: vowel harmony

NOM	PART	gloss
taivas	taivas+**ta**	sky
puhelin	puheli+**ta**	telephone
lakeus	lakeus+**ta**	plain
syy	syy+**tä**	reason
lyhyt	lyhyt+**tä**	short
ystävällinen	ystävällinen+**tä**	friendly

Implementing morphology and phonology

We begin with a simple problem: a lexicon of some natural language is given as a list of words. Suggest a data structure that will provide insertion and retrieval of data. As a first solution, we are looking for time efficiency rather than space efficiency.

The solution: *trie* (word tree).

Access time: $O(|w|)$. Space requirement: $O(\sum_w |w|)$.

A trie can be augmented to store also a morphological dictionary specifying concatenative affixes, especially suffixes. In this case it is better to turn the tree into a graph.

The obtained model is that of *finite-state automata*.

Finite-state technology

Finite-state automata are not only a good model for representing the lexicon, they are also perfectly adequate for representing dictionaries (lexicons+additional information), describing morphological processes that involve concatenation etc.

A natural extension of finite-state automata – finite-state transducers – is a perfect model for most processes known in morphology and phonology, including non-segmental ones.

Formal language theory – definitions

Formal languages are defined with respect to a given *alphabet*, which is a finite set of symbols, each of which is called a *letter*.

A finite sequence of letters is called a *string*.

Example: Strings
Let $\Sigma = \{0, 1\}$ be an alphabet. Then all binary numbers are strings over Σ.

If $\Sigma = \{a, b, c, d, \ldots, y, z\}$ is an alphabet then *cat*, *incredulous* and *supercalifragilisticexpialidocious* are strings, as are *tac*, *qqq* and *kjshdflkwjehr*.

Formal language theory – definitions

The *length* of a string w, denoted $|w|$, is the number of letters in w. The unique string of length 0 is called the *empty string* and is denoted ϵ.

If $w_1 = \langle x_1, \ldots, x_n \rangle$ and $w_2 = \langle y_1, \ldots, y_m \rangle$, the *concatenation* of w_1 and w_2, denoted $w_1 \cdot w_2$, is the string $\langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$. $|w_1 \cdot w_2| = |w_1| + |w_2|$.

For every string w, $w \cdot \epsilon = \epsilon \cdot w = w$.

Formal language theory – definitions

Example: Concatenation
Let $\Sigma = \{a, b, c, d, \ldots, y, z\}$ be an alphabet. Then *master* $\cdot$ *mind* = *mastermind*, *mind* $\cdot$ *master* = *mindmaster* and *master* $\cdot$ *master* = *mastermaster*. Similarly, *learn* $\cdot$ *s* = *learns*, *learn* $\cdot$ *ed* = *learned* and *learn* $\cdot$ *ing* = *learning*.

Formal language theory – definitions

An *exponent* operator over strings is defined in the following way: for every string w, $w^0 = \epsilon$. Then, for $n > 0$, $w^n = w^{n-1} \cdot w$.

Example: Exponent
If w = *go*, then $w^0 = \epsilon$, $w^1 = w$ = *go*, $w^2 = w^1 \cdot w = w \cdot w$ = *gogo*, w^3 = *gogogo* and so on.

Formal language theory – definitions

The *reversal* of a string w is denoted w^R and is obtained by writing w in the reverse order. Thus, if $w = \langle x_1, x_2, \ldots, x_n \rangle$, $w^R = \langle x_n, x_{n-1}, \ldots, x_1 \rangle$.

Given a string w, a *substring* of w is a sequence formed by taking contiguous symbols of w in the order in which they occur in w. If $w = \langle x_1, \ldots, x_n \rangle$ then for any i, j such that $1 \leq i \leq j \leq n$, $\langle x_i, \ldots x_j \rangle$ is a substring of w.

Two special cases of substrings are *prefix* and *suffix*: if $w = w_l \cdot w_c \cdot w_r$ then w_l is a prefix of w and w_r is a suffix of w.

Formal language theory – definitions

Example: Substrings
Let $\Sigma = \{a, b, c, d, \ldots, y, z\}$ be an alphabet and w = *indistinguishable* a string over Σ. Then ϵ, *in*, *indis*, *indistinguish* and *indistinguishable* are prefixes of w, while ϵ, *e*, *able*, *distinguishable* and *indistinguishable* are suffixes of w. Substrings that are neither prefixes nor suffixes include *distinguish*, *gui* and *is*.

Formal language theory – definitions

Given an alphabet Σ, the set of all strings over Σ is denoted by Σ^*.

A *formal language* over an alphabet Σ is a subset of Σ^*.

Formal language theory – definitions

Example: Languages
Let $\Sigma = \{a, b, c, \ldots, y, z\}$. Then Σ^* is the set of all strings over the Latin alphabet. Any subset of this set is a language. In particular, the following are formal languages:

Formal language theory – definitions

- Σ^*;
- the set of strings consisting of consonants only;
- the set of strings consisting of vowels only;
- the set of strings each of which contains at least one vowel and at least one consonant;
- the set of palindromes;
- the set of strings whose length is less than 17 letters;
- the set of single-letter strings;
- the set $\{i, you, he, she, it, we, they\}$;
- the set of words occurring in Joyce's Ulysses;
- the empty set;

Note that the first five languages are infinite while the last five are finite.

Formal language theory – definitions

The string operations can be lifted to languages.

If L is a language then the *reversal* of L, denoted L^R, is the language $\{w \mid w^R \in L\}$.

If L_1 and L_2 are languages, then
$L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$.

Example: Language operations
$L_1 = \{i, you, he, she, it, we, they\}$, $L_2 = \{smile, sleep\}$.

Then ${L_1}^R = \{i, uoy, eh, ehs, ti, ew, yeht\}$ and $L_1 \cdot L_2 = \{ismile, yousmile, hesmile, shesmile, itsmile, wesmile, theysmile, isleep, yousleep, hesleep, shesleep, itsleep, wesleep, theysleep\}$.

Formal language theory – definitions

If L is a language then $L^0 = \{\epsilon\}$.
Then, for $i > 0$, $L^i = L \cdot L^{i-1}$.

Example: Language exponentiation
Let L be the set of words {*bau*, *haus*, *hof*, *frau*}. Then $L^0 = \{\epsilon\}$, $L^1 = L$ and $L^2 = \{$*baubau*, *bauhaus*, *bauhof*, *baufrau*, *hausbau*, *haushaus*, *haushof*, *hausfrau*, *hofbau*, *hofhaus*, *hofhof*, *hoffrau*, *fraubau*, *frauhaus*, *frauhof*, *fraufrau*$\}$.

Formal language theory – definitions

The *Kleene closure* of L and is denoted L^* and is defined as $\bigcup_{i=0}^{\infty} L^i$.
$L^+ = \bigcup_{i=1}^{\infty} L^i$.

Example: Kleene closure
Let $L = \{$*dog*, *cat*$\}$. Observe that $L^0 = \{\epsilon\}$, $L^1 = \{$*dog*, *cat*$\}$, $L^2 = \{$*catcat*, *catdog*, *dogcat*, *dogdog*$\}$, etc. Thus L^* contains, among its infinite set of strings, the strings ϵ, *cat*, *dog*, *catcat*, *catdog*, *dogcat*, *dogdog*, *catcatcat*, *catdogcat*, *dogcatcat*, *dogdogcat*, etc.

The notation for Σ^* should now become clear: it is simply a special case of L^*, where $L = \Sigma$.

Regular expressions

Regular expressions are a formalism for defining (formal) languages. Their "syntax" is formally defined and is relatively simple. Their "semantics" is sets of strings: the denotation of a regular expression is a set of strings in some formal language.

Regular expressions

Regular expressions are defined recursively as follows:

- $\emptyset$ is a regular expression
- ϵ is a regular expression
- if $a \in \Sigma$ is a letter then a is a regular expression
- if r_1 and r_2 are regular expressions then so are $(r_1 + r_2)$ and $(r_1 \cdot r_2)$
- if r is a regular expression then so is $(r)^*$
- nothing else is a regular expression over Σ.

Regular expressions

Example: Regular expressions
Let Σ be the alphabet $\{a, b, c, \ldots, y, z\}$. Some regular expressions over this alphabet are:

- $\emptyset$
- a
- $((c \cdot a) \cdot t)$
- $(((m \cdot e) \cdot (o)^*) \cdot w)$
- $(a + (e + (i + (o + u))))$
- $((a + (e + (i + (o + u)))))^*$

Regular expressions

For every regular expression r its denotation, $[\![r]\!]$, is a set of strings defined as follows:

- $[\![\emptyset]\!] = \emptyset$
- $[\![\epsilon]\!] = \{\epsilon\}$
- if $a \in \Sigma$ is a letter then $[\![a]\!] = \{a\}$
- if r_1 and r_2 are regular expressions whose denotations are $[\![r_1]\!]$ and $[\![r_2]\!]$, respectively, then $[\![(r_1 + r_2)]\!] = [\![r_1]\!] \cup [\![r_2]\!]$, $[\![(r_1 \cdot r_2)]\!] = [\![r_1]\!] \cdot [\![r_2]\!]$ and $[\![(r_1)^*]\!] = [\![r_1]\!]^*$

Regular expressions

Example: Regular expressions and their denotations

$\emptyset$	$\emptyset$
a	$\{a\}$
$((c \cdot a) \cdot t)$	$\{c \cdot a \cdot t\}$
$(((m \cdot e) \cdot (o)^*) \cdot w)$	*{mew, meow, meoow, meooow, meooo*
$(a + (e + (i + (o + u))))$	$\{a, e, i, o, u\}$
$((a + (e + (i + (o + u)))))^*$	*all strings of 0 or more vowels*

Regular expressions

Example: Regular expressions
Given the alphabet of all English letters, $\Sigma = \{a, b, c, \ldots, y, z\}$, the language Σ^* is denoted by the regular expression Σ^*.

The set of all strings which contain a vowel is denoted by $\Sigma^* \cdot (a + e + i + o + u) \cdot \Sigma^*$.

The set of all strings that begin in "*un*" is denoted by $(un)\Sigma^*$.

The set of strings that end in either "*tion*" or "*sion*" is denoted by $\Sigma^* \cdot (s + t) \cdot (ion)$.

Note that all these languages are infinite.

Properties of regular languages

Closure properties:

A class of languages $\mathcal{L}$ is said to be closed under some operation '$\bullet$' if and only if whenever two languages L_1, L_2 are in the class ($L_1, L_2 \in \mathcal{L}$), also the result of performing the operation on the two languages is in this class: $L_1 \bullet L_2 \in \mathcal{L}$.

Properties of regular languages

Regular languages are closed under:

- Union
- Intersection
- Complementation
- Difference
- Concatenation
- Kleene-star
- Substitution and homomorphism

Finite-state automata

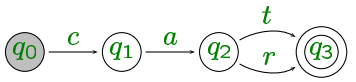
Automata are models of computation: they compute languages.

A finite-state automaton is a five-tuple $\langle Q, q_0, \Sigma, \delta, F\rangle$, where Σ is a finite set of **alphabet** symbols, Q is a finite set of **states**, $q_0 \in Q$ is the **initial state**, $F \subseteq Q$ is a set of **final** (accepting) states and $\delta : Q \times \Sigma \times Q$ is a relation from states and alphabet symbols to states.

Finite-state automata

Example: Finite-state automaton

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{c, a, t, r\}$
- $F = \{q_3\}$
- $\delta = \{\langle q_0, c, q_1\rangle, \langle q_1, a, q_2\rangle, \langle q_2, t, q_3\rangle, \langle q_2, r, q_3\rangle\}$



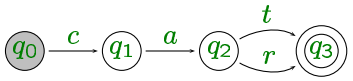
Finite-state automata

The reflexive transitive extension of the transition relation δ is a new relation, $\hat{\delta}$, defined as follows:

- for every state $q \in Q$, $(q, \epsilon, q) \in \hat{\delta}$
- for every string $w \in \Sigma^*$ and letter $a \in \Sigma$, if $(q, w, q') \in \hat{\delta}$ and $(q', a, q'') \in \delta$ then $(q, w \cdot a, q'') \in \hat{\delta}$.

Finite-state automata

Example: Paths
For the finite-state automaton:



$\hat{\delta}$ is the following set of triples:

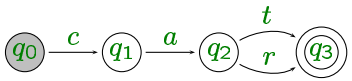
$\langle q_0, \epsilon, q_0 \rangle, \langle q_1, \epsilon, q_1 \rangle, \langle q_2, \epsilon, q_2 \rangle, \langle q_3, \epsilon, q_3 \rangle,$
$\langle q_0, c, q_1 \rangle, \langle q_1, a, q_2 \rangle, \langle q_2, t, q_3 \rangle, \langle q_2, r, q_3 \rangle,$
$\langle q_0, ca, q_2 \rangle, \langle q_1, at, q_3 \rangle, \langle q_1, ar, q_3 \rangle,$
$\langle q_0, cat, q_3 \rangle, \langle q_0, car, q_3 \rangle$

Finite-state automata

A string w is accepted by the automaton $A = \langle Q, q_0, \Sigma, \delta, F \rangle$ if and only if there exists a state $q_f \in F$ such that $(q_0, w, q_f) \in \hat{\delta}$.

The *language accepted by a finite-state automaton* is the set of all string it accepts.

Example: Language
The language of the finite-state automaton:



is {*cat*, *car*}.

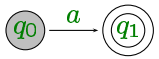
Finite-state automata

Example: Some finite-state automata

$\emptyset$

Finite-state automata

Example: Some finite-state automata



$\{a\}$

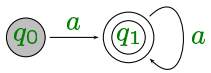
Finite-state automata

Example: Some finite-state automata

$\{\epsilon\}$

Finite-state automata

Example: Some finite-state automata



$\{a, aa, aaa, aaaa, \ldots\}$

Finite-state automata

Example: Some finite-state automata

a^*

Finite-state automata

Example: Some finite-state automata

?

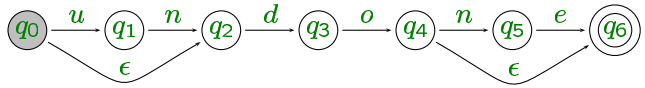
Σ^*

Finite-state automata

An extension: ϵ-moves.

The transition relation δ is extended to: $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$

Example: Automata with ϵ-moves
The language accepted by the following automaton is {*do*, *undo*, *done*, *undone*}:

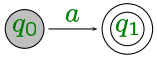
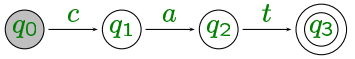
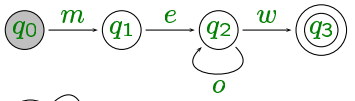
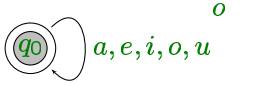


Finite-state automata

Theorem (Kleene, 1956): The class of languages recognized by finite-state automata is the class of regular languages.

Finite-state automata

Example: Finite-state automata and regular expressions

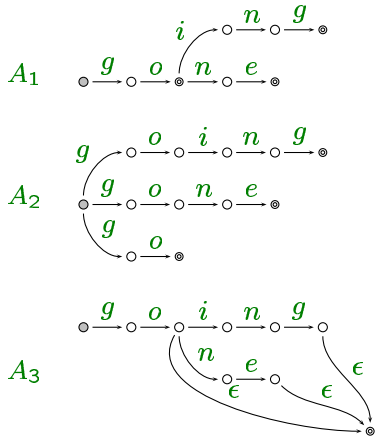
$\emptyset$	
a	
$((c \cdot a) \cdot t)$	
$(((m \cdot e) \cdot (o)^*) \cdot w)$	
$((a + (e + (i + (o + u)))))^*$	

Operations on finite-state automata

- Concatenation
- Union
- Intersection
- Minimization
- Determinization

Minimization and determinization

Example: Equivalent automata



Applications of finite-state automata in NLP

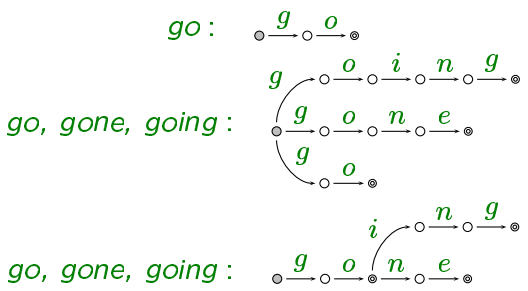
Finite-state automata are efficient computational devices for generating regular languages.

An equivalent view would be to regard them as *recognizing* devices: given some automaton A and a word w, applying the automaton to the word yields an answer to the question: Is w a member of $L(A)$, the language accepted by the automaton?

This reversed view of automata motivates their use for a simple yet necessary application of natural language processing: dictionary lookup.

Applications of finite-state automata in NLP

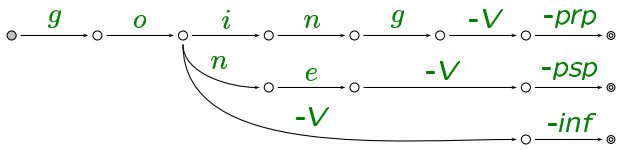
Example: Dictionaries as finite-state automata



Applications of finite-state automata in NLP

Example: Adding morphological information
Add information about part-of-speech, the number of nouns and the tense of verbs:

$\Sigma = \{a, b, c, \ldots, y, z, \text{-N}, \text{-V}, \text{-sg}, \text{-pl}, \text{-inf}, \text{-prp}, \text{-psp}\}$



The appeal of regular languages for NLP

- Most phonological and morphological process of natural languages can be straight-forwardly described using the operations that regular languages are closed under.
- Most algorithms on finite-state automata are linear.
- The closure properties of regular languages naturally support modular development of finite-state grammars.

Regular relations

While regular expressions are sufficiently expressive for some natural language applications, it is sometimes useful to define relations over two sets of strings.

Regular relations

Part-of-speech tagging:

I	know	some	new	tricks	
PRON	V	DET	ADJ	N	
said	the	Cat	in	the	Hat
V	DET	N	P	DET	N

Regular relations

Morphological analysis:

I	know	some	new
I-PRON-1-sg	know-V-pres	some-DET-indef	new-ADJ
tricks	said	the	Cat
trick-N-pl	say-V-past	the-DET-def	cat-N-sg
in	the	Hat	
in-P	the-DET-def	hat-N-sg	

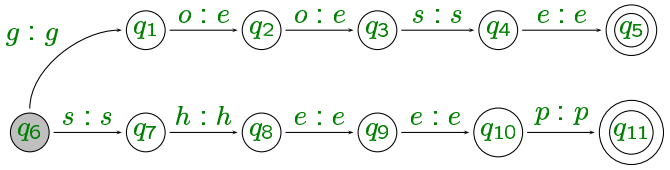
Regular relations

Singular-to-plural mapping:

cat	hat	ox	child	mouse	sheep	goose
cats	hats	oxen	children	mice	sheep	geese

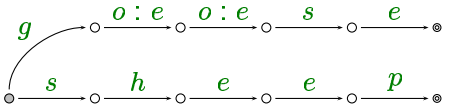
Finite-state transducers

A finite-state transducer is a six-tuple $\langle Q, q_0, \Sigma_1, \Sigma_2, \delta, F \rangle$. Similarly to automata, Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final (or accepting) states, Σ_1 and Σ_2 are alphabets: finite sets of symbols, not necessarily disjoint (or different). $\delta : Q \times \Sigma_1 \times \Sigma_2 \times Q$ is a relation from states and pairs of alphabet symbols to states.

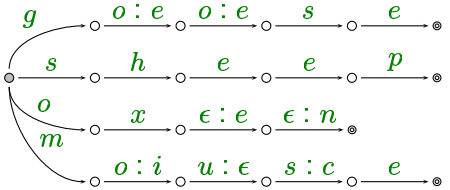


Finite-state transducers

Shorthand notation:



Adding ϵ-moves:



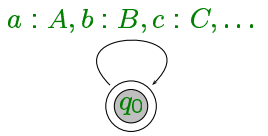
Finite-state transducers

The language of a finite-state transducer is a set of pairs: a binary relation over $\Sigma_1^* \times \Sigma_2^*$. The language is defined analogously to how the language of an automaton is defined.

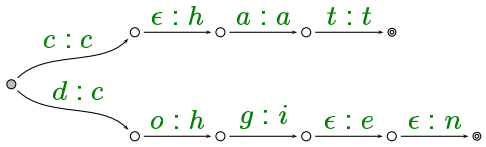
$T(w) = \{u \mid (q_0, w, u, q_f) \in \hat{\delta} \text{ for some } f \in F\}$.

Finite-state transducers

Example: The uppercase transducer



Example: English-to-French



Properties of finite-state transducers

Given a transducer $\langle Q, q_0, \Sigma_1, \Sigma_2, \delta, F\rangle$,

- its *underlying automaton* is $\langle Q, q_0, \Sigma_1 \times \Sigma_2, \delta', F\rangle$, where $(q_1, (a, b), q_2) \in \delta'$ iff $(q_1, a, b, q_2) \in \delta$
- its *upper automaton* is $\langle Q, q_0, \Sigma_1, \delta_1, F\rangle$, where $(q_1, a, q_2) \in \delta_1$ iff for some $b \in \Sigma_2$, $(q_1, a, b, q_2) \in \delta$
- its *lower automaton* is $\langle Q, q_0, \Sigma_2, \delta_2, F\rangle$, where $(q_1, b, q_2) \in \delta_2$ iff for some $a \in \Sigma_a$, $(q_1, a, b, q_2) \in \delta$

Properties of finite-state transducers

A transducer T is *functional* if for every $w \in \Sigma_1^*$, $T(w)$ is either empty or a singleton.

Transducers are closed under union: if T_1 and T_2 are transducers, there exists a transducer T such that for every $w \in \Sigma_1^*$, $T(w) = T_1(w) \cup T_2(w)$.

Transducers are closed under inversion: if T is a transducer, there exists a transducer T^{-1} such that for every $w \in \Sigma_1^*$, $T^{-1}(w) = \{u \in \Sigma_2^* \mid w \in T(u)\}$.

The inverse transducer is $\langle Q, q_0, \Sigma_2, \Sigma_1, \delta^{-1}, F\rangle$, where $(q_1, a, b, q_2) \in \delta^{-1}$ iff $(q_1, b, a, q_2) \in \delta$.

Properties of regular relations

Example: Operations on finite-state relations

$R_1 = \{$*tomato:Tomate, cucumber:Gurke, grapefruit:Grapefruit, pineapple:Ananas, coconut:Koko*$\}$

$R_2 = \{$*grapefruit:pampelmuse, coconut:Kokusnuß*$\}$

$R_1 \cup R_2 = \{$*tomato:Tomate, cucumber:Gurke, grapefruit:Grapefruit, grapefruit:pampelmuse, pineapple:Ananas, coconut:Koko ,coconut:Kokusnuß*$\}$

Properties of finite-state transducers

Transducers are closed under composition: if T_1 is a transduction from Σ_1^* to Σ_2^* and and T_2 is a transduction from Σ_2^* to Σ_3^*, then there exists a transducer T such that for every $w \in \Sigma_1^*$, $T(w) = T_2(T_1(w))$.

The number of states in the composition transducer might be $|Q_1 \times Q_2|$.

Example: Composition of finite-state relations

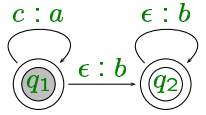
$R_1 = \{$*tomato:Tomate, cucumber:Gurke, grapefruit:Grapefruit, grapefruit:pampelmuse, pineapple:Ananas, coconut:Koko ,coconut:Kokusnuß*$\}$

$R_2 = \{$*tomate:tomato, ananas:pineapple, pampelmousse:grapefruit, concombre:cucumber, cornichon:cucumber, noix-de-coco:coconut*$\}$

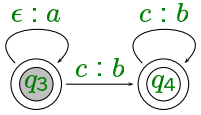
$R_2 \circ R_1 = \{$*tomate:Tomate, ananas:Ananas, pampelmousse:Grapefruit, pampelmousse:Pampelmuse, concombre:Gurke,cornichon:Gurke, noix-de-coco:Koko, noix-de-coco:Kokusnuße*$\}$

Properties of finite-state transducers

Transducers are not closed under intersection.



T_1



T_2

$T_1(c^n) = \{a^n b^m \mid m \geq 0\}$
$T_2(c^n) = \{a^m b^n \mid m \geq 0\} \Rightarrow$
$(T_1 \cap T_2)(c^n) = \{a^n b^n\}$

Transducers with no ϵ-moves are closed under intersection.

Properties of finite-state transducers

- Computationally efficient
- Denote regular relations
- Closed under concatenation, Kleene-star, union
- Not closed under intersection (and hence complementation)
- Closed under composition
- Weights

Introduction to XFST

- XFST is an interface giving access to finite-state operations (algorithms such as union, concatenation, iteration, intersection, composition etc.)
- XFST includes a regular expression compiler
- The interface of XFST includes a lookup operation (*apply up*) and a generation operation (*apply down*)
- The regular expression language employed by XFST is an extended version of standard regular expressions

Introduction to XFST

`a`	a simple symbol
`c a t`	a concatenation of three symbols
`[c a t]`	grouping brackets
`?`	denotes any single symbol
`%+`	the literal plus-sign symbol
`%*`	the literal asterisk symbol (and similarly for `%?`, `%(`, `%]` etc.
`‘‘+Noun’’`	single symbol with multicharacter print name
`%+Noun`	single symbol with multicharacter print name
`cat`	a single multicharacter symbol

Introduction to XFST

`{cat}`	equivalent to `[c a t]`
`[ ]`	the empty string
`0`	the empty string
`[A]`	bracketing; equivalent to `A`
`A|B`	union
`(A)`	optionality; equivalent to `[A|0]`
`A&B`	intersection
`A B`	concatenation
`A-B`	set difference

Introduction to XFST

`A*`	Kleene-star
`A+`	one or more iterations
`?*`	the universal language
`~A`	the complement of `A`; equivalent to `[?* - A]`
`~[?*]`	the empty language

Introduction to XFST – denoting relations

`A .x. B`	Cartesian product; relates every string in `A` to every string in `B`
`a:b`	shorthand for `[a .x. b]`
`%+Pl:s`	shorthand for `[%+Pl .x. s]`
`%+Past:ed`	shorthand for `[%+Past .x. ed]`
`%+Prog:ing`	shorthand for `[%+Prog .x. ing]`

Introduction to XFST – useful abbreviations

`$A`	the language of all the strings that contain `A`; equivalent to `[?* A ?*]`
`A/B`	the language of all the strings in `A`, ignoring any strings from `B`, e.g.,
`a*/b`	includes strings such as `a`, `aa`, `aaa`, `ba`, `ab`, `aba` etc.
`\A`	any single symbol, minus strings in `A`. Equivalent to `[?  - A]`, e.g.,
`\b`	any single symbol, except 'b'. Compare to:
`~A`	the complement of `A`, i.e., `[?* - A]`

Introduction to XFST – user interface

```
prompt% H:\class\data\shuly\xfst

xfst> help
xfst> help union net
xfst> exit
xfst> read regex [d o g | c a t];
xfst> read regex < myfile.regex
xfst> apply up dog
xfst> apply down dog
xfst> pop stack
xfst> clear stack
xfst> save stack myfile.fsm
```

Introduction to XFST – example

```
[ [l e a v e %+VBZ .x. l e a v e s] |
[l e a v e %+VB .x. l e a v e] |
[l e a v e %+VBG .x. l e a v i n g] |
[l e a v e %+VBD .x. l e f t] |
[l e a v e %+NN .x. l e a v e] |
[l e a v e %+NNS .x. l e a v e s] |
[l e a f %+NNS .x. l e a v e s ] |
[l e f t %+JJ .x. l e f t] ]
```


Introduction to XFST – example of lookup and generation

```
APPLY DOWN> leave+VBD
left
APPLY UP> leaves
leave+NNS
leave+VBZ
leaf+NNS
```


Introduction to XFST – variables

```
xfst> define Myvar;
xfst> define Myvar2 [d o g | c a t];
xfst> undefine Myvar;


xfst> define var1 [b i r d | f r o g | d o g];
xfst> define var2 [d o g | c a t];
xfst> define var3 var1 | var2;
xfst> define var4 var1 var2;
xfst> define var5 var1 & var2;
xfst> define var6 var1 - var2;
```


Introduction to XFST – variables

```
xfst> define Root [w a l k | t a l k | w o r k];
xfst> define Prefix [0 | r e];
xfst> define Suffix [0 | s | e d | i n g];
xfst> read regex Prefix Root Suffix;
xfst> words
xfst> apply up walking
```

Introduction to XFST – replace rules

Replace rules are an extremely powerful extension of the regular expression metalanguage.

The simplest replace rule is of the form

$$upper \rightarrow lower \parallel leftcontext \_ rightcontext$$

Its denotation is the relation which maps string to themselves, with the exception that an occurrence of *upper* in the input string, preceded by *leftcontext* and followed by *rightcontext*, is replaced in the output by *lower*.

Introduction to XFST – replace rules

Word boundaries can be explicitly referred to:

```
xfst> define Vowel [a|e|i|o|u];
xfst> e -> ’ || [.#.] [c | d | l | s] _ [% Vowel];
```


Introduction to XFST – replace rules

Contexts can be omitted:

```
xfst> define Rule1 N -> m || _ p ;
xfst> define Rule2 N -> n ;
xfst> define Rule3 p -> m || m _ ;
```

This can be used to clear unnecessary symbols introduced for “bookkeeping”:

```
xfst> define Rule1 %^MorphmeBoundary -> 0;
```


Introduction to XFST – replace rules

The language Bambona has an underspecified nasal morpheme *N* that is realized as a labial *m* or as a dental *n* depending on its environment: *N* is realized as *m* before *p* and as *n* elsewhere.

The language also has an assimilation rule which changes *p* to *m* when the *p* is followed by *m*.

```
xfst> clear stack ;
xfst> define Rule1 N -> m || _ p ;
xfst> define Rule2 N -> n ;
xfst> define Rule3 p -> m || m _ ;
xfst> read regex Rule1 .o. Rule2 .o. Rule3 ;
```

Introduction to XFST – replace rules

Rules can define multiple replacements:

```
[ A -> B, B -> A ]
```

or multiple replacements that share the same context:

```
[ A -> B, B -> A || L _ R ]
```

or multiple contexts:

```
[ A -> B || L1 _ R1, L2 _ R2 ]
```

or multiple replacements and multiple contexts:

```
[ A -> B, B -> A ||  L1 _ R1, L2 _ R2 ]
```


Introduction to XFST – replace rules

Rules can apply in parallel:

```
xfst> clear stack
xfst> read regex a -> b .o. b -> a ;
xfst> apply down abba
aaaa
xfst> clear stack
xfst> read regex b -> a .o. a -> b ;
xfst> apply down abba
bbbb
xfst> clear stack
xfst> read regex a -> b , b -> a ;
xfst> apply down abba
baab
```


Introduction to XFST – replace rules

When rules that have contexts apply in parallel, the rule separator is a double comma:

```
xfst> clear stack
xfst> read regex
b -> a || .#. s ?* _ ,, a -> b || _ ?* e .#. ;
xfst> apply down sabbae
sbaabe
```


Introduction to XFST – morphology

English verb morphology:

```
define Vowel [a|e|i|o|u] ;
define Cons [? - Vowel] ;
define base [{dip} | {print} | {fuss} | {toss} | {bake} | {move}] ;
define suffix [0 | {s} | {ed} | {ing}] ;
define form [base %+ suffix];
define FinalS [s %+ s] -> [s %+ e s] ;
define FinalE e -> 0 || _ %+ [e | i];
define DoubleFinal b -> [b b], d -> [d d], f -> [f f],
                   g -> [g g],k -> [k k], l -> [l l], m -> [m m],
                   n -> [n n],p -> [p p], r -> [r r], s -> [s s],
                   t -> [t t],z -> [z z] || Cons Vowel _ %+ Vowel ;
define RemovePlus %+ -> 0 ;
read regex form .o. DoubleFinal .o. FinalS .o. FinalE
               .o. RemovePlus;
```

Introduction to XFST – morphology

English spell checking:

```
clear;
define A [i e] -> [e i] || c _ ,,
         [e i] -> [i e] || [? - c] _ ;

define B [[e i] -> [i e]] .o.
         [[i e] -> [e i] || c _];
read regex B;
```


Introduction to XFST – morphology

Hebrew adjective inflection:

```
clear stack ;
define base [{gdwl} | {yph} | {xbrwty} | {rk} | {adwm} |
             {q$h} | {ap$ry}] ;
define suffix [0 | {h} | {ym} | {wt}] ;
define form [base %+ suffix];
define FinalH h -> 0 || _ %+ ? ;
define FinalY h -> t || y %+ _ ;
define RemovePlus %+ -> 0 ;
read regex form .o. FinalH .o. FinalY .o. RemovePlus;
```


Introduction to XFST – marking

The special symbol "..." in the right-hand side of a replace rule stands for whatever was matched in the left-hand side of the rule.

```
xfst> clear stack;
xfst> read regex [a|e|i|o|u] -> %[ ... %];
xfst> apply down unnecessarily
[u]nn[e]c[e]ss[a]r[i]ly
```


Introduction to XFST – marking

```
xfst> clear stack;
xfst> read regex [a|e|i|o|u]+ -> %[ ... %];
xfst> apply down feeling
f[e][e]l[i]ng
f[ee]l[i]ng
xfst> apply down poolcleaning
p[o][o]lcl[e][a]n[i]ng
p[oo]lcl[e][a]n[i]ng
p[o][o]lcl[ea]n[i]ng
p[oo]lcl[ea]n[i]ng
xfst> read regex [a|e|i|o|u]+ @-> %[ ... %];
xfst> apply down poolcleaning
p[oo]lcl[ea]n[i]ng
```

Introduction to XFST – shallow parsing

Assume that text is represented as strings of part-of-speech tags, using 'd' for determiner, 'a' for adjective, 'n' for noun, and 'v' verb, etc. In other words, in this example the regular expression symbols represent whole words rather than single letters in a text.

Assume that a noun phrase consists of an optional determiner, any number of adjectives, and one or more nouns:

```
[(d) a* n+]
```

This expression denotes an infinite set of strings, such as "n" (cats), "aan" (discriminating aristocratic cats), "nn" (cat food), "dn" (many cats), "dann" (that expensive cat food) etc.

Introduction to XFST – shallow parsing

A simple noun phrase parser can be thought of as a transducer that inserts markers, say, a pair of braces { }, around noun phrases in a text. The task is not as trivial as it seems at first glance. Consider the expression

```
[(d) a* n+ -> %{ ... %}]
```

Applied to the input "danvn" (many small cats like milk) this transducer yields three alternative bracketings:

```
xfst> apply down danvn
da{n}v{n}
d{an}v{n}
{dan}v{n}
```


Introduction to XFST – longest match

For certain applications it may be desirable to produce a unique parse, marking the maximal expansion of each NP: "{dan}v{n}". Using the left-to-right, longest-match replace operator `@->` instead of the simple replace operator `->` yields the desired result:

```
[(d) a* n+ @-> %{ ... %}]
```

```
xfst> apply down danvn
{dan}v{n}
```


Introduction to XFST – the coke machine

A vending machine dispenses drinks for 65 cents a can. It accepts any sequence of the following coins: 5 cents (represented as 'n'), 10 cents ('d') or 25 cents ('q'). Construct a regular expression that compiles into a finite-state automaton that implements the behavior of the soft drink machine, pairing "PLONK" with a legal sequence that amounts to 65 cents.

Introduction to XFST – the coke machine

The construction `A^n` denotes the concatenation of `A` with itself n times.

Thus the expression `[n .x. c^5]` expresses the fact that a nickel is worth 5 cents.

A mapping from all possible sequences of the three symbols to the corresponding value:

```
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]*
```

The solution:

```
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]*
.o.
[c^65 .x. PLONK]
```


Introduction to XFST – the coke machine

```
clear stack
define SixtyFiveCents
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]* ;
define BuyCoke
SixtyFiveCents .o. [c^65 .x. PLONK] ;
```


Introduction to XFST – the coke machine

In order to ensure that extra money is paid back, we need to modify the lower language of BuyCoke to make it a subset of `[PLONK* q* d* n*]`.

To ensure that the extra change is paid out only once, we need to make sure that quarters get paid before dimes and dimes before nickels.

```
clear stack
define SixtyFiveCents
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]* ;
define ReturnChange SixtyFiveCents .o.
[[c^65 .x. PLONK]* [c^25 .x. q]*
[c^10 .x. d]* [c^5 .x. n]*] ;
```


Introduction to XFST – the coke machine

The next refinement is to ensure that as much money as possible is converted into soft drinks and to remove any ambiguity in how the extra change is to be reimbursed.

```
clear stack
define SixtyFiveCents
[[n .x. c^5] | [d .x. c^10] | [q .x. c^25]]* ;
define ReturnChange SixtyFiveCents .o.
[[c^65 .x. PLONK]* [c^25 .x. q]*
[c^10 .x. d]* [c^5 .x. n]*] ;
define ExactChange ReturnChange .o.
[~$[q q q | [q q | d] d [d | n] | n n]] ;
```

Introduction to XFST – the coke machine

To make the machine completely foolproof, we need one final improvement. Some clients may insert unwanted items into the machine (subway tokens, foreign coins, etc.). These objects should not be accepted; they should passed right back to the client. This goal can be achieved easily by wrapping the entire expression inside an ignore operator.

```
define IgnoreGarbage
[ [ ExactChange ]/[\[q | d | n]]] ;
```


Applications of finite-state technology in NLP

- Phonology; language models for speech recognition
- Representing lexicons and dictionaries
- Morphology; morphological analysis and generation
- Shallow parsing
- Named entity recognition
- Sentence boundary detection; segmentation
- Translation...

Syntax

Syntax is the area of linguistics which studies the structure of natural languages.

The underlying assumption is that languages have *structure*: not all sequences of words over the given alphabet is valid; and when a sequence of words is valid (*grammatical*), a natural structure can be induced on it.

It is useful to think of this structure as a *tree* (although we shall see other structures later).

Given a sentence in some language, not all possible trees define the structure that native speaker of the language intuitively recognize.

Natural languages have structure

Even though I *klaw* through the valley of the shadow of death,
I will *raef* no evil

Even though I walk through the valley of the shadow of death,
I will fear no evil

Even though I *ordinary* through the valley of the shadow of death,
I will *slowly* no evil

Even though I *slowly gaze* through the valley of the shadow of death,
I will *unsurprisingly do* no evil

Even *though I* walk through the valley of the shadow of death,
I will fear no evil

Natural languages have structure

Natural languages are infinite:

- The water put out the fire
- The water put out the fire, that burned the stick
- The water put out the fire, that burned the stick, that hit the dog
- The water put out the fire, that burned the stick, that hit the dog, that chased th

But it is possible to characterize an infinite set with finite expressions.

Natural languages have structure

Intuitively, words combine to form *phrases*:

((yonatan ha-qatan) ((rac ba-boqer) ('el ha-gan)))

but not:

((yonatan (ha-qatan rac)) (ba-boqer 'el) ha-gan)

Phrases which correspond to our native speaker intuitions are called *constituents*.

Determining constituents

The criteria for defining constituents are sometimes fuzzy.

The main criterion is equivalent distribution: if two word sequences are mutually interchangeable in every context, preserving grammaticality, then both are constituents and both have the same grammatical category.

Determining constituents

- Certain grammatical operations apply only to constituents:

  Topicalization
  Cleft
  Interjection
  Question formation

- Only full constituents (of the same type) can be coordinated
- Anaphors refer to constituents

Types of constituents

Inducing structure on a grammatical string is done recursively, starting with the words. To this end, words are classified into *categories* according to their distribution.

In many languages, words are classified into *substantial* and *functional* categories.

substantial: table, dogs, walked, purple, quickly

functional: the, in, or

Another classification is according to whether the category is *open* or *close*.

Types of constituents

Word categories (parts of speech):

N	Noun	table, dogs, justice, oak
V	Verb	run, climb, love, ignore
ADJ	Adjective	green, fast, mild, imaginary
ADV	Adverb	quickly, well, alone
P	Preposition	in, to, of, after, in spite of
D	Determiner	a, the, all, some
Pron	Pronoun	I, you, she, theirs, our
PropN	Proper Noun	John, IBM, University of Haifa

Constituents

Phrases are projections of word categories:

Noun phrases are headed by nouns: table → round table → the round table → t

Verb phrases are headed by verbs: climbed → climbed a tree → climbed a tree

Adjectival phrases are headed by adjectives: high → rather high / higher than

Constituents

Phrases consist of a *head* and additional *complements* and *adjuncts*. The phrase is a *projection* of its head.

Complements are required by the head, and are mandatory. Adjuncts are optional, and can be iterated.

Example: John drinks a cup of milk every morning

A gradual description of language fragments

E_0 is a small fragment of English consisting of very simple sentences, constructed with only intransitive and transitive (but no ditransitive) verbs, common nouns, proper names, pronouns and determiners.

Typical sentences are:

A sheep drinks

Rachel herds the sheep

Jacob loves her

A gradual description of language fragments

Similar strings are not E_0- (and, hence, English-) sentences:

*Rachel feed the sheep

*Rachel feeds herds the sheep

*The shepherds feeds the sheep

*Rachel feeds

*Jacob loves she

*Jacob loves Rachel she

*Them herd the sheep

A gradual description of language fragments

There are constraints on the combination of phrases in E_0:

- The subject and the predicate must agree on number and person: if the subject is a third person singular, so must the verb be.
- Objects complement only – and all – the transitive verbs.
- When a pronoun is used, it is in the nominative case if it is in the subject position, and in the accusative case if it is an object.

Subcategorization

E_1 is a fragment of English, based on E_0, in which verbs are classified to subclasses according to the complements they "require":

Laban gave Jacob his daughter

Jacob promised Laban to marry Leah

Laban persuaded Jacob to promise him to marry Leah

Similar strings that violate this constraint are:

*Rachel feeds Jacob the sheep

*Jacob saw to marry Leah

Control

With the addition of infinitival complements in E_1, E_2 can capture constraints of argument *control* in English:

Jacob promised Laban to work seven years

Laban persuaded Jacob to work seven years

Long distance dependencies

Another extension of E_1 is E_3, typical sentences of which are:

(1) The shepherd wondered whom Jacob loved ⌙.

(2) The shepherd wondered whom Laban thought Jacob loved ⌙.

(3) The shepherd wondered whom Laban thought Rachel claimed Jacob loved ⌙.

An attempt to replace the gap with an explicit noun phrase results in ungrammaticality:

(4) $*$The shepherd wondered who Jacob loved Rachel.

Long distance dependencies

The gap need not be in the object position:

(5) Jacob wondered who ⌙ loved Leah

(6) Jacob wondered who Laban believed ⌙ loved Leah

Again, an explicit noun phrase filling the gap results in ungrammaticality:

(7) Jacob wondered who the shepherd loved Leah

Long distance dependencies

More than one gap may be present in a sentence (and, hence, more than one filler):

(8a) This is the well which Jacob is likely to ⌙ draw water from ⌙

(8b) It was Leah that Jacob worked for ⌙ without loving ⌙

In some languages (e.g., Norwegian) there is no (principled) bound on the number of gaps that can occur in a single clause.

Long distance dependencies

There are other fragments of English in which long distance dependencies are manifested in other forms. *Topicalization*:

(9) Rachel, Jacob loved ␣

(10) Rachel, every shepherd knew Jacob loved ␣

Another example is *interrogative sentences*:

(11) Who did Jacob love ␣?

(12) Who did Laban believe Jacob loved ␣?

Coordination

Coordination is accounted for in the language fragment E_4:

No man lift up his [hand] or [foot] in all the land of Egypt

Jacob saw [Rachel] and [the sheep of Laban]

Jacob [went on his journey] and [came to the land of the people of the east]

Jacob [went near], and [rolled the stone from the well's mouth], and [watered the flock of Laban his mother's brother].

every [speckled] and [spotted] sheep

Leah was [tender eyed] but [not beautiful]

[Leah had four sons], but [Rachel was barren]

She said to Jacob, "[Give me children], or [I shall die]!"

Context-free grammars

A **context-free grammar** (CFG) is a four-tuple $\langle \Sigma, V, S, P \rangle$, where:

- Σ is a finite, non-empty set of **terminals**, the alphabet;
- V is a finite, non-empty set of **grammar variables** (categories, or non-terminal symbols), such that $\Sigma \cap V = \emptyset$;
- $S \in V$ is the **start symbol**;
- P is a finite set of **production rules**, each of the form $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.

For a rule $A \to \alpha$, A is the rule's **head** and α is its **body**.

Context-free grammars: example

$\Sigma = \{\textit{the, cat, in, hat}\}$

$V = \{\textit{D, N, P, NP, PP}\}$

The start symbol is *NP*

The rules:

$D \to \textit{the}$	$NP \to D\ N$
$N \to \textit{cat}$	$PP \to P\ NP$
$N \to \textit{hat}$	$NP \to NP\ PP$
$P \to \textit{in}$	

Context-free grammars: language

Each non-terminal symbol in a grammar denotes a language.

A rule such as $N \to cat$ implies that the language denoted by the non-terminal N includes the alphabet symbol *cat*.

The symbol *cat* here is a single, atomic alphabet symbol, and not a string of symbols: the alphabet of this example consists of natural language words, not of natural language letters.

For a more complex rule such as $NP \to D\ N$, the language denoted by NP contains the concatenation of the language denoted by D with that denoted by N: $L(NP) \supseteq L(D) \cdot L(N)$.

Matters become more complicate when we consider recursive rules such as $NP \to NP\ PP$.

Context-free grammars: derivation

Given a grammar $G = \langle V, \Sigma, P, S \rangle$, we define the set of *forms* to be $(V \cup \Sigma)^*$: the set of all sequences of terminal and non-terminal symbols.

Derivation is a relation that holds between two forms, each a sequence of grammar symbols.

A form α *derives* a form β, denoted by $\alpha \Rightarrow \beta$, if and only if $\alpha = \gamma_l A \gamma_r$ and $\beta = \gamma_l \gamma_c \gamma_r$ and $A \to \gamma_c$ is a rule in P.

A is called the *selected symbol*. The rule $A \to \gamma$ is said to be **applicable** to α.

Derivation: example

The set of non-terminals of G is $V = \{D, N, P, NP, PP\}$ and the set of terminals is $\Sigma = \{the, cat, in, hat\}$.

The set of forms therefore contains all the (infinitely many) sequences of elements from V and Σ, such as $\langle\rangle$, $\langle NP \rangle$, $\langle D\ cat\ P\ D\ hat \rangle$, $\langle D\ N \rangle$, $\langle the\ cat\ in\ the\ hat \rangle$, etc.

Let us start with a simple form, $\langle NP \rangle$. Observe that it can be written as $\gamma_l NP \gamma_r$, where both γ_l and γ_r are empty. Observe also that NP is the head of some grammar rule: the rule $NP \to D\ N$. Therefore, the form is a good candidate for derivation: if we replace the selected symbol NP with the body of the rule, while preserving its environment, we get $\gamma_l D\ N \gamma_r = D\ N$. Therefore, $\langle NP \rangle \Rightarrow \langle D\ N \rangle$.

Derivation: example

We now apply the same process to $\langle D\ N \rangle$. This time the selected symbol is D (we could have selected N, of course). The left context is again empty, while the right context is $\gamma_r = N$. As there exists a grammar rule whose head is D, namely $D \to the$, we can replace the rule's head by its body, preserving the context, and obtain the form $\langle the\ N \rangle$. Hence $\langle D\ N \rangle \Rightarrow \langle the\ N \rangle$.

Derivation: example

Given the form $\langle$*the N*$\rangle$, there is exactly one non-terminal that we can select, namely *N*. However, there are two rules that are headed by *N*: $N \to$ *cat* and $N \to$ *hat*. We can select either of these rules to show that both $\langle$*the N*$\rangle \Rightarrow \langle$*the cat*$\rangle$ and $\langle$*the N*$\rangle \Rightarrow \langle$*the hat*$\rangle$.

Since the form $\langle$*the cat*$\rangle$ consists of terminal symbols only, no non-terminal can be selected and hence it derives no form.

Extended derivation

$\alpha \overset{k}{\Rightarrow}_G \beta$ if α derives β in k steps: $\alpha \Rightarrow_G \alpha_1 \Rightarrow_G \alpha_2 \Rightarrow_G \ldots \Rightarrow_G \alpha_k$ and $\alpha_k = \beta$.

The reflexive-transitive closure of '$\Rightarrow_G$' is '$\overset{*}{\Rightarrow}_G$': $\alpha \overset{*}{\Rightarrow}_G \beta$ if $\alpha \overset{k}{\Rightarrow}_G \beta$ for some $k \geq 0$.

A **G-derivation** is a sequence of forms $\alpha_1, \ldots, \alpha_n$, such that for every $i, 1 \leq i < n$, $\alpha_i \Rightarrow_G \alpha_{i+1}$.

Extended derivation: example

(1) $\langle$*NP*$\rangle$ $\Rightarrow$ $\langle$*D N*$\rangle$
(2) $\langle$*D N*$\rangle$ $\Rightarrow$ $\langle$*the N*$\rangle$
(3) $\langle$*the N*$\rangle$ $\Rightarrow$ $\langle$*the cat*$\rangle$

Therefore, we trivially have:

Extended derivation: example

(4) $\langle$*NP*$\rangle$ $\overset{*}{\Rightarrow}$ $\langle$*D N*$\rangle$
(5) $\langle$*D N*$\rangle$ $\overset{*}{\Rightarrow}$ $\langle$*the N*$\rangle$
(6) $\langle$*the N*$\rangle$ $\overset{*}{\Rightarrow}$ $\langle$*the cat*$\rangle$

From (2) and (6) we get

(7) $\langle$*D N*$\rangle$ $\overset{*}{\Rightarrow}$ $\langle$*the cat*$\rangle$

and from (1) and (7) we get

(7) $\langle$*NP*$\rangle$ $\overset{*}{\Rightarrow}$ $\langle$*the cat*$\rangle$

Languages

A form α is a **sentential form** of a grammar G iff $S \stackrel{*}{\Rightarrow}_G \alpha$, i.e., it can be derived in G from the start symbol.

The (formal) **language** generated by a grammar G with respect to a category name (non-terminal) A is $L_A(G) = \{w \mid A \stackrel{*}{\Rightarrow} w\}$. The language generated by the grammar is $L(G) = L_S(G)$.

A language that can be generated by some CFG is a context-free language and the class of context-free languages is the set of languages every member of which can be generated by some CFG. If no CFG can generate a language L, L is said to be trans-context-free.

Language of a grammar

For the example grammar (with *NP* the start symbol):

$D \rightarrow$ *the*	$NP \rightarrow D\ N$
$N \rightarrow$ *cat*	$PP \rightarrow P\ NP$
$N \rightarrow$ *hat*	$NP \rightarrow NP\ PP$
$P \rightarrow$ *in*	

it is fairly easy to see that $L(D) = \{the\}$.

Similarly, $L(P) = \{in\}$ and $L(N) = \{cat,\ hat\}$.

Language of a grammar

It is more difficult to define the languages denoted by the non-terminals *NP* and *PP*, although is should be straight-forward that the latter is obtained by concatenating $\{in\}$ with the former.

Proposition: $L(NP)$ is the denotation of the regular expression

$$the \cdot (cat + hat) \cdot (in \cdot\ the \cdot (cat + hat))^*$$

Language: a formal example G_e

S	$\rightarrow$	$V_a\ S\ V_b$
S	$\rightarrow$	ϵ
V_a	$\rightarrow$	a
V_b	$\rightarrow$	b

$L(G_e) = \{a^n b^n \mid n \geq 0\}$.

Recursion

The language $L(G_e)$ is infinite: it includes an infinite number of words; G_e is a finite grammar.

To be able to produce infinitely many words with a finite number of rules, a grammar must be recursive: there must be at least one rule whose body contains a symbol, from which the head of the rule can be derived.

Put formally, a grammar $\langle \Sigma, V, S, P \rangle$ is recursive if there exists a chain of rules, $p_1, \ldots, p_n \in P$, such that for every $1 < i \leq n$, the head of p_{i+1} occurs in the body of p_i, and the head of p_1 occurs in the body of p_n.

In G_e, the recursion is simple: the chain of rules is of length 0, namely the rule $S \to V_a \ S \ V_b$ is in itself recursive.

Derivation tree

Sometimes derivations provide more information than is actually needed. In particular, sometimes two derivations of the same string differ not in the rules that were applied but only in the order in which they were applied.

Starting with the form $\langle NP \rangle$ it is possible to derive the string *the cat* in two ways:

(1) $\langle NP \rangle \Rightarrow \langle D\ N \rangle \Rightarrow \langle D\ cat \rangle \Rightarrow \langle the\ cat \rangle$
(2) $\langle NP \rangle \Rightarrow \langle D\ N \rangle \Rightarrow \langle the\ N \rangle \Rightarrow \langle the\ cat \rangle$

Since both derivations use the same rules to derive the same string, it is sometimes useful to collapse such "equivalent" derivations into one. To this end the notion of *derivation trees* is introduced.

Derivation tree

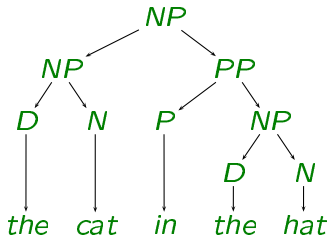
A derivation tree (sometimes called *parse* tree, or simply tree) is a visual aid in depicting derivations, and a means for imposing structure on a grammatical string.

Trees consist of vertices and branches; a designated vertex, the *root* of the tree, is depicted on the top. Then, branches are simply connections between two vertices.

Intuitively, trees are depicted "upside down", since their root is at the top and their leaves are at the bottom.

Derivation tree: example

An example for a derivation tree for the string *the cat in the hat*:



Derivation tree

Formally, a tree consists of a finite set of vertices and a finite set of branches (or arcs), each of which is an ordered pair of vertices.

In addition, a tree has a designated vertex, the *root*, which has two properties: it is not the target of any arc, and every other vertex is accessible from it (by following one or more branches).

When talking about trees we sometimes use family notation: if a vertex v has a branch leaving it which leads to some vertex u, then we say that v is the *mother* of u and u is the *daughter*, or *child*, of v. If u has two daughters, we refer to them as *sisters*.

Derivation trees

Derivation trees are defined with respect to some grammar G, and must obey the following conditions:

1. every vertex has a *label*, which is either a terminal symbol, a non-terminal symbol or ϵ;
2. the label of the root is the start symbol;
3. if a vertex v has an outgoing branch, its label must be a non-terminal symbol, the head of some grammar rule; and the elements in body of the same rule must be the labels of the children of v, in the same order;
4. if a vertex is labeled ϵ, it is the only child of its mother.

Derivation trees

A *leaf* is a vertex with no outgoing branches.

A tree induces a natural "left-to-right" order on its leaves; when read from left to right, the sequence of leaves is called the *frontier*, or *yield* of the tree.

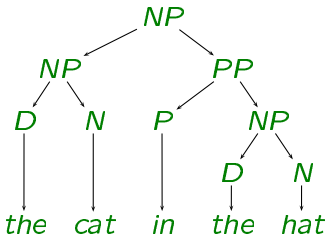
Correspondence between trees and derivations

Derivation trees correspond very closely to derivations.

For a form α, a non-terminal symbol A derives α if and only if α is the yield of some parse tree whose root is A.

Sometimes there exist different derivations of the same string that correspond to a single tree. In fact, the tree representation collapses exactly those derivations that differ from each other only in the order in which rules are applied.

Correspondence between trees and derivations



Each non-leaf vertex in the tree corresponds to some grammar rule (since it must be labeled by the head of some rule, and its children must be labeled by the body of the same rule).

Correspondence between trees and derivations

This tree represents the following derivations (among others):

(1) $NP \Rightarrow NP\ PP \Rightarrow D\ N\ PP \Rightarrow D\ N\ P\ NP$
$\Rightarrow D\ N\ P\ D\ N \Rightarrow the\ N\ P\ D\ N$
$\Rightarrow the\ cat\ P\ D\ N \Rightarrow the\ cat\ in\ D\ N$
$\Rightarrow the\ cat\ in\ the\ N \Rightarrow the\ cat\ in\ the\ hat$

(2) $NP \Rightarrow NP\ PP \Rightarrow D\ N\ PP \Rightarrow the\ N\ PP$
$\Rightarrow the\ cat\ PP \Rightarrow the\ cat\ P\ NP$
$\Rightarrow the\ cat\ in\ NP \Rightarrow the\ cat\ in\ D\ N$
$\Rightarrow the\ cat\ in\ the\ N \Rightarrow the\ cat\ in\ the\ hat$

(3) $NP \Rightarrow NP\ PP \Rightarrow NP\ P\ NP \Rightarrow NP\ P\ D\ N$
$\Rightarrow NP\ P\ D\ hat \Rightarrow NP\ P\ the\ hat$
$\Rightarrow NP\ in\ the\ hat \Rightarrow D\ N\ in\ the\ hat$
$\Rightarrow D\ cat\ in\ the\ hat \Rightarrow the\ cat\ in\ the\ hat$

Correspondence between trees and derivations

While exactly the same rules are applied in each derivation (the rules are uniquely determined by the tree), they are applied in different orders. In particular, derivation (2) is a *leftmost* derivation: in every step the leftmost non-terminal symbol of a derivation is expanded. Similarly, derivation (3) is *rightmost*.

Ambiguity

Sometimes, however, different derivations (of the same string!) correspond to different trees.

This can happen only when the derivations differ in the rules which they apply.

When more than one tree exists for some string, we say that the string is *ambiguous*.

Ambiguity is a major problem when grammars are used for certain formal languages, in particular programming languages. But for natural languages, ambiguity is unavoidable as it corresponds to properties of the natural language itself.

Ambiguity: example

Consider again the example grammar and the following string:

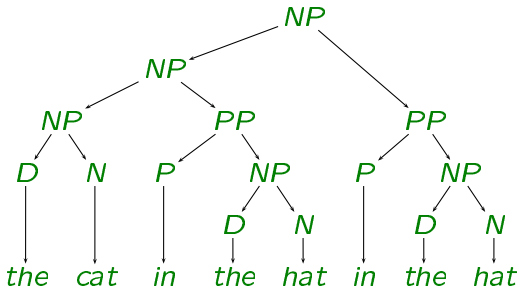
the cat in the hat in the hat

Intuitively, there can be (at least) two readings for this string: one in which a certain cat wears a hat-in-a-hat, and one in which a certain cat-in-a-hat is inside a hat:

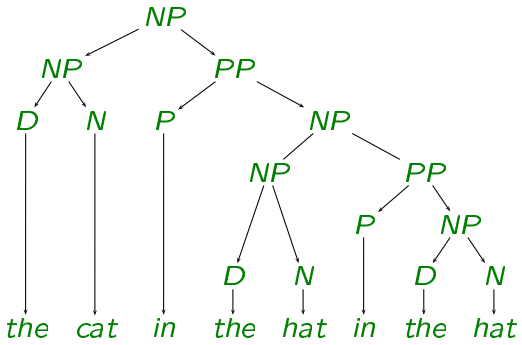
((the cat in the hat) in the hat)
(the cat in (the hat in the hat))

This distinction in intuitive meaning is reflected in the grammar, and hence two different derivation trees, corresponding to the two readings, are available for this string:

Ambiguity: example



Ambiguity: example



Ambiguity: example

Using linguistic terminology, in the left tree the second occurrence of the prepositional phrase *in the hat* modifies the noun phrase *the cat in the hat*, whereas in the right tree it only modifies the (first occurrence of) the noun phrase *the hat*. This situation is known as *syntactic* or *structural* ambiguity.

Grammar equivalence

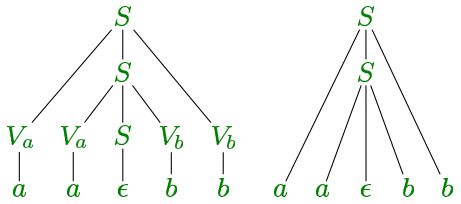
It is common in formal language theory to relate different grammars that generate the same language by an equivalence relation:

Two grammars G_1 and G_2 (over the same alphabet Σ) are **equivalent** (denoted $G_1 \equiv G_2$) iff $L(G_1) = L(G_2)$.

We refer to this relation as weak equivalence, as it only relates the generated languages. Equivalent grammars may attribute totally different syntactic structures to members of their (common) languages.

Grammar equivalence

Example: Equivalent grammars, different trees
Following are two different tree structures that are attributed to the string $aabb$ by the grammars G_e and G_f, respectively.

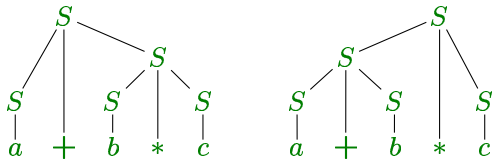


Grammar equivalence

Example: Structural ambiguity
A grammar, G_{arith}, for simple arithmetic expressions:

$$S \rightarrow a \mid b \mid c \mid S + S \mid S * S$$

Two different trees can be associated by G_{arith} with the string $a + b * c$:



Grammar equivalence

Weak equivalence relation is stated in terms of the generated language. Consequently, equivalent grammars do not have to be described in the same formalism for them to be equivalent. We will later see how grammars, specified in different formalisms, can be compared.

Normal form

It is convenient to divide grammar rules into two classes: one that contains only phrasal rules of the form $A \to \alpha$, where $\alpha \in V^*$, and another that contains only terminal rules of the form $B \to \sigma$ where $\sigma \in \Sigma$. It turns out that every CFG is equivalent to some CFG of this form.

Normal form

A grammar G is in **phrasal/terminal normal form** iff for every production $A \to \alpha$ of G, either $\alpha \in V^*$ or $\alpha \in \Sigma$. Productions of the form $A \to \sigma$ are called **terminal rules**, and A is said to be a **pre-terminal category**, the **lexical entry** of σ. Productions of the form $A \to \alpha$, where $\alpha \in V^*$, are called **phrasal rules**. Furthermore, every category is either pre-terminal or phrasal, but not both. For a phrasal rule with $\alpha = A_1 \cdots A_n, w = w_1 \cdots w_n, w \in L_A(G)$ and $w_i \in L_{A_i}(G)$ for $i = 1, \ldots, n$, we say that w is a phrase of category A, and each w_i is a **sub-phrase** (of w) of category A_i. A sub-phrase w_i of w is also called a **constituent** of w.

Context-free grammars for natural languages

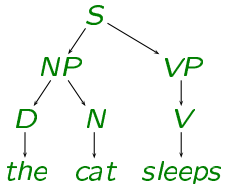
A context-free grammar for English sentences: $G = \langle V, \Sigma, P, S \rangle$ where $V = \{D, N, P, NP, PP, V, VP, S\}$, $\Sigma = \{$*the, cat, in, hat, sleeps, smile, loves, saw*$\}$, the start symbol is S and P is the following set of rules:

$S \to NP\ VP$	$D \to$ *the*
$NP \to D\ N$	$N \to$ *cat*
$NP \to NP\ PP$	$N \to$ *hat*
$PP \to P\ NP$	$V \to$ *sleeps*
$VP \to V$	$P \to$ *in*
$VP \to VP\ NP$	$V \to$ *smile*
$VP \to VP\ PP$	$V \to$ *loves*
	$V \to$ *saw*

Context-free grammars for natural languages

The augmented grammar can derive strings such as *the cat sleeps* or *the cat in the hat saw the hat*.

A derivation tree for *the cat sleeps* is:



Context-free grammars for natural languages

There are two major problems with this grammar.

1. it ignores the valence of verbs: there is no distinction among subcategories of verbs, and an intransitive verb such as *sleep* might occur with a noun phrase complement, while a transitive verb such as *love* might occur without one. In such a case we say that the grammar *overgenerates*: it generates strings that are not in the intended language.

2. there is no treatment of subject–verb agreement, so that a singular subject such as *the cat* might be followed by a plural form of verb such as *smile*. This is another case of overgeneration.

Both problems are easy to solve.

Verb valence

To account for valence, we can replace the non-terminal symbol V by a set of symbols: *Vtrans*, *Vintrans*, *Vditrans* etc. We must also change the grammar rules accordingly:

$VP \rightarrow Vintrans$	$Vintrnas \rightarrow sleeps$
$VP \rightarrow Vtrans\ NP$	$Vintrans \rightarrow smile$
$VP \rightarrow Vditrans\ NP\ PP$	$Vtrans \rightarrow loves$
	$Vditrans \rightarrow give$

Agreement

To account for agreement, we can again extend the set of non-terminal symbols such that categories that must agree reflect in the non-terminal that is assigned for them the features on which they agree. In the very simple case of English, it is sufficient to multiply the set of "nominal" and "verbal" categories, so that we get *Dsg*, *Dpl*, *Nsg*, *Npl*, *NPsg*, *NPpl*, *Vsg*, *Vlp*, *VPsg*, *VPpl* etc. We must also change the set of rules accordingly:

Agreement

$Nsg \rightarrow cat$	$Npl \rightarrow cats$
$Nsg \rightarrow hat$	$Npl \rightarrow hats$
$P \rightarrow in$	
$Vsg \rightarrow sleeps$	$Vpl \rightarrow sleep$
$Vsg \rightarrow smiles$	$Vpl \rightarrow smile$
$Vsg \rightarrow loves$	$Vpl \rightarrow love$
$Vsg \rightarrow saw$	$Vpl \rightarrow saw$
$Dsg \rightarrow a$	$Dpl \rightarrow many$

Agreement

$S \rightarrow NPsg\ VPsg$	$S \rightarrow NPpl\ VPpl$
$NPsg \rightarrow Dsg\ Nsg$	$NPpl \rightarrow Dpl\ Npl$
$NPsg \rightarrow NPsg\ PP$	$NPpl \rightarrow NPpl\ PP$
$PP \rightarrow P\ NP$	
$VPsg \rightarrow Vsg$	$VPpl \rightarrow Vpl$
$VPsg \rightarrow VPsg\ NP$	$VPpl \rightarrow VPpl\ NP$
$VPsg \rightarrow VPsg\ PP$	$VPpl \rightarrow VPpl\ PP$

Context-free grammars for natural languages

Context-free grammars can be used for a variety of syntactic constructions, including some non-trivial phenomena such as unbounded dependencies, extraction, extraposition etc.

However, some (formal) languages are not context-free, and therefore there are certain sets of strings that cannot be generated by context-free grammars.

The interesting question, of course, involves natural languages: are there natural languages that are not context-free? Are context-free grammars sufficient for generating every natural language?

Parsing

Recognition: Given a (context-free) grammar G and a string of words w, determine whether $w \in L(G)$.

Parsing: If $w \in L(G)$, produce the (tree) structure that is assigned by G to w.

Parsing

General requirements for a parsing algorithm:

- Generality: the algorithm must be applicable to *any* grammar
- Completeness: the algorithm must produce *all* the results in case of ambiguity
- Efficiency
- Flexibility: a good algorithm can be easily modified

Parsing

Parameters that define different parsing algorithms:

Orientation: Top-down vs. bottom-up vs. mixed

Direction: Left-to-right vs. right-to-left vs. mixed (e.g., island-driven)

Handling multiple choice: Dynamic programming vs. parallel processing vs. backtracking

Search: Breadth-first or Depth-first

An example grammar

$D \rightarrow the$	$NP \rightarrow D\ N$
$N \rightarrow cat$	$PP \rightarrow P\ NP$
$N \rightarrow hat$	$NP \rightarrow NP\ PP$
$P \rightarrow in$	

Example sentences:

the cat in the hat

the cat in the hat in the hat

A bottom-up recognition algorithm

Assumptions:

- The grammar is given in Chomsky Normal Form: each rule is either of the form $A \rightarrow B\ C$ (where A, B, C are non-terminals) or of the form $A \rightarrow a$ (where a is a terminal).
- The string to recognize is $w = w_1 \cdots w_n$.
- A set of indices $\{0, 1, \ldots, n\}$ is defined to point to positions between the input string's words:

  0 the 1 cat 2 in 3 the 4 hat 5

The CYK algorithm

Bottom-up, chart-based recognition algorithm for grammars in CNF

To recognize a string of length n, uses a *chart*: a bi-dimensional matrix of size $n \times (n+1)$

Invariant: a non-terminal A is stored in the $[i, j]$ entry of the chart iff $A \Rightarrow w_{i+1} \cdots w_j$

Consequently, the chart is triangular. A word w is recognized iff the start symbol S is in the $[0, n]$ entry of the chart

The idea: build all constituents up to the i-th position before constructing the $i+1$ position; build smaller constituents before constructing larger ones

The CYK algorithm

```
for j := 1 to n do
   for all rules A → w_j do
     chart[j-1,j] := chart[j-1,j] ∪ {A}
   for i := j-2 downto 0 do
     for k := i+1 to j-1 do
       for all B ∈ chart[i,k] do
         for all C ∈ chart[k,j] do
           for all rules A → B C do
             chart[i,j] := chart[i,j] ∪ {A}
if S ∈ chart[0,n] then accept else reject
```


The CYK algorithm

Extensions:

- Parsing in addition to recognition
- Support for ϵ-rules
- General context-free grammars (not just CNF)

Parsing schemata

To provide a unified framework for discussing various parsing algorithms we use *parsing schemata*, which are generalized schemes for describing the principles behind specific parsing algorithms. This is a generalization of the *parsing as deduction* paradigm.

A parsing schema consists of four components:

- a set of items
- a set of axioms
- a set of deduction rules
- a set of goal items

Parsing schema: CYK

Given a grammar $G = \langle \Sigma, V, S, P \rangle$ and a string $w = w_1 \cdots w_n$:

Items: $[i, A, j]$ for $A \in V$ and $0 \leq i, j \leq n$
(state that $A \stackrel{*}{\Rightarrow} w_{i+1} \cdots w_j$)

Axioms: $[i, A, i+1]$ when $A \to w_{i+1} \in P$

Goals: $[0, S, n]$

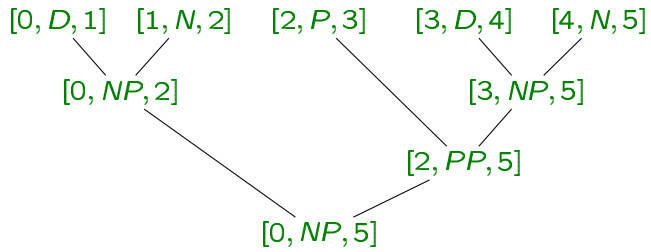
Inference rules:

$$\frac{[i, B, j] \quad [j, C, k]}{[i, A, k]} \quad A \to B\ C$$

CYK parsing schema: deduction example

$D \to$ *the* $\quad NP \to D\ N$
$N \to$ *cat* $\quad PP \to P\ NP$
$N \to$ *hat* $\quad NP \to NP\ PP$
$P \to$ *in*

0 the 1 cat 2 in 3 the 4 hat 5



Parsing: bottom-up schema (Shift–Reduce)

Items: $[\alpha\bullet, j]$ (state that $\alpha w_{j+1} \cdots w_n \stackrel{*}{\Rightarrow} w_1 \cdots w_n$)

Axioms: $[\bullet, 0]$

Goals: $[S\bullet, n]$

Inference rules:

Shift $\dfrac{[\alpha\bullet, j]}{[\alpha w_{j+1}\bullet, j+1]}$

Reduce $\dfrac{[\alpha\gamma\bullet, j]}{[\alpha B\bullet, j]} \quad B \to \gamma$

Bottom-up deduction: example

Parsing: top-down schema

Item form: $[\bullet\beta, j]$ (state that $S \stackrel{*}{\Rightarrow} w_1 \cdots w_j \beta$)

Axioms: $[\bullet S, 0]$

Goals: $[\bullet, n]$

Inference rules:

Scan $\dfrac{[\bullet w_{j+1}\beta, j]}{[\bullet\beta, j+1]}$

Predict $\dfrac{[\bullet B\beta, j]}{[\bullet\gamma\beta, j]} \quad B \to \gamma$

Top-down deduction: example

Input: 0 the 1 cat 2 in 3 the 4 hat 5

$[\bullet NP, 0]$	*axiom*
$[\bullet NP\ PP, 0]$	*predict* $NP \rightarrow NP\ PP$
$[\bullet D\ N\ PP, 0]$	*predict* $NP \rightarrow D\ N$
$[\bullet the\ N\ PP, 0]$	*predict* $D \rightarrow the$
$[\bullet N\ PP, 1]$	*scan*
$[\bullet cat\ PP, 1]$	*predict* $N \rightarrow cat$
$[\bullet PP, 2]$	*scan*
$[\bullet P\ NP, 2]$	*predict* $PP \rightarrow P\ NP$
$[\bullet in\ NP, 2]$	*predict* $P \rightarrow in$
$[\bullet NP, 3]$	*scan*
$[\bullet D\ N, 3]$	*predict* $NP \rightarrow D\ N$
$[\bullet the\ N, 3]$	*predict* $D \rightarrow the$
$[\bullet N, 4]$	*scan*
$[\bullet hat, 4]$	*predict* $N \rightarrow hat$
$[\bullet, 5]$	*scan*

Top-down parsing: algorithm

```
Parse(β,j) ::
   if β = w_{j+1} · β' then return parse(β', j + 1)
   else if β = B · β' then
     for every rule B → γ ∈ P
       if Parse(γ · β', j) then return true
   return false

if Parse(S,0) then accept else reject
```


Top-down vs. Bottom-up parsing

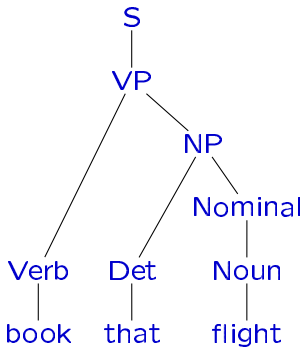
Two inherent constraints:

1. The root of the tree is S
2. The yield of the tree is the input word

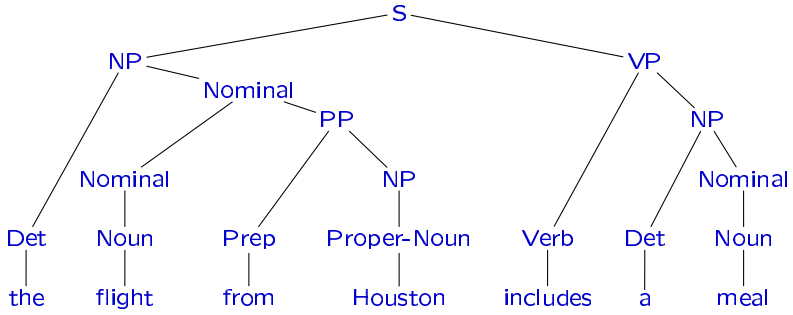
An example grammar

$S \rightarrow NP\ VP$	$Det \rightarrow that \mid this \mid a$
$S \rightarrow Aux\ NP\ VP$	$Noun \rightarrow book \mid flight \mid meal$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid includes$
$VP \rightarrow Verb$	$Prep \rightarrow from \mid to \mid on$
$VP \rightarrow Verb\ NP$	$Proper\text{-}Noun \rightarrow Houston \mid TWA$
$NP \rightarrow Det\ Nominal$	$Aux \rightarrow does$
$NP \rightarrow Proper\text{-}Noun$	
$Nominal \rightarrow Noun$	
$Nominal \rightarrow Noun\ Nominal$	
$Nominal \rightarrow Nominal\ PP$	
$PP \rightarrow Prep\ NP$	

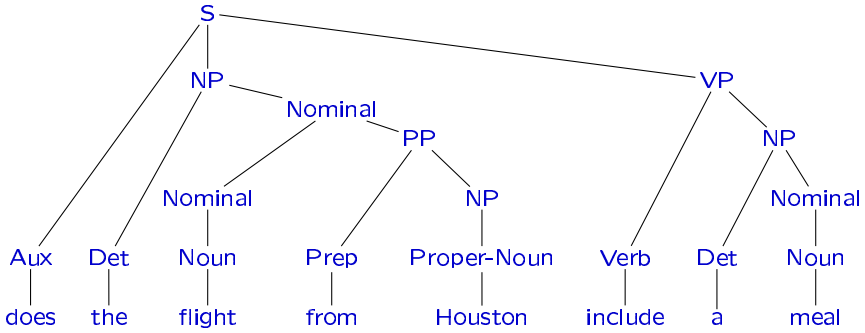
An example derivation tree



An example derivation tree



An example derivation tree



Top-down vs. Bottom-up parsing

When expanding the top-down search space, which local trees are created?

Top-down vs. Bottom-up parsing

To reduce "blind" search, add bottom-up filtering.

Observation: when trying to `Parse(`β`,j)`, where $\beta = B\gamma$, the parser succeeds only if $B \stackrel{*}{\Rightarrow} w_{j+1}\beta$.

Definition: A word w is a **left-corner** of a non-terminal B iff $B \stackrel{*}{\Rightarrow} w\beta$ for some β.

Top-down parsing with bottom-up filtering

```
Parse(β,j) ::
   if β = w_{j+1} · β' then return parse(β', j + 1)
   else if β = B · β' then
      if w_{j+1} is a left-corner of B then
        for every rule B → γ ∈ P
          if Parse(γ · β', j) then return true
   return false

if Parse(S,0) then accept else reject
```

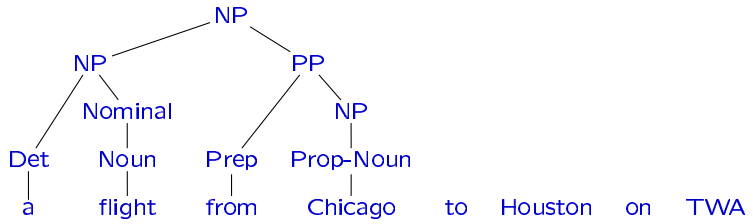

Top-down vs. Bottom-up parsing

Even with bottom-up filtering, top-down parsing suffers from the following problems:

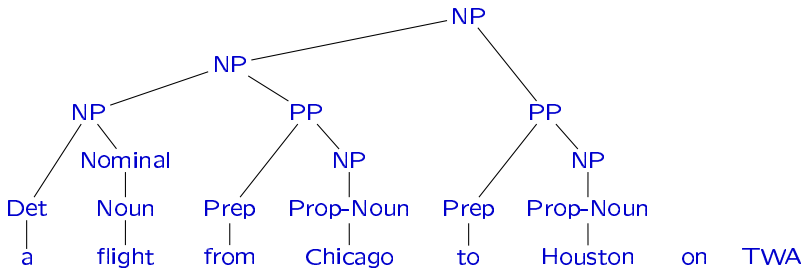
- Left recursive rules can cause non-termination: $NP \rightarrow NP\ PP$.
- Even when parsing terminates, it might take exponentially many steps.
- Constituents are computed over and over again

Top-down parsing: repeated generation of sub-trees

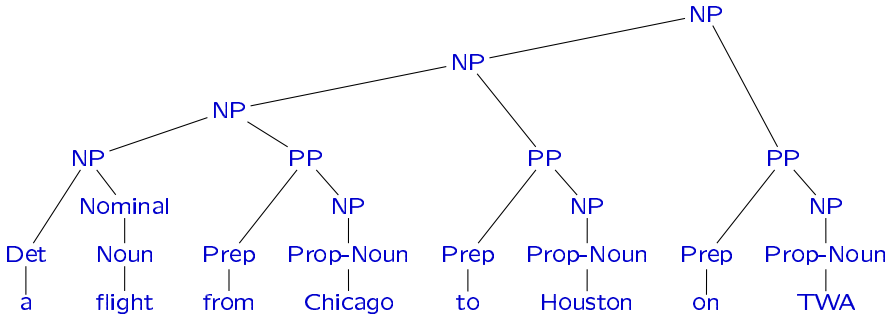
Top-down parsing: repeated generation of sub-trees



Top-down parsing: repeated generation of sub-trees



Top-down parsing: repeated generation of sub-trees



Top-down parsing: repeated generation of sub-trees

Reduplication:

Constituent	#
a flight	4
from Chicago	3
to Houston	2
on TWA	1
a flight from Chicago	3
a flight from Chicago to Houston	2
a flight from Chicago to Houston on TWA	1

Top-down vs. Bottom-up parsing

When expanding the bottom-up search space, which local trees are created?

Top-down vs. Bottom-up parsing

Bottom-up parsing suffers from the following problems:

- All possible analyses of every substring are generated, even when they can never lead to an S, or can never combine with their neighbors
- ϵ-rules can cause performance degradation
- Reduplication of effort

Earley's parsing algorithm

- Dynamic programming: partial results are stored in a chart
- Combines top-down predictions with bottom-up scanning
- No reduplication of computation
- Left-recursion is correctly handled
- ϵ-rules are handled correctly
- Worst-case complexity: $O(n^3)$

Earley's parsing algorithm

Basic concepts:

Dotted rules: if $A \to \alpha\beta$ is a grammar rule then $A \to \alpha \bullet \beta$ is a dotted rule.

Edges: if $A \to \alpha \bullet \beta$ is a dotted rule and i, j are indices into the input string then $[i, A \to \alpha \bullet \beta, j]$ is an edge. An edge is **passive** (or **complete**) if $\beta = \epsilon$, **active** otherwise.

Actions: The algorithm performs three operations: *scan*, *predict* and *complete*.

Earley's parsing algorithm

scan: read an input word and add a corresponding complete edge to the chart.

predict: when an active edge is added to the chart, predict all possible edges that can follow it

complete: when a complete edge is added to the chart, combine it with appropriate active edges

Earley's parsing algorithm

rightsisters: given an active edge $A \to \alpha \bullet B\beta$, return all dotted rules $B \to \bullet\gamma$

leftsisters: given a complete edge $B \to \gamma\bullet$, return all dotted edges $A \to \alpha \bullet B\beta$

combination:

$$[i, A \to \alpha \bullet B\beta, k] * [k, B \to \gamma\bullet, j] = [i, A \to \alpha B \bullet \beta, j]$$

Parsing: Earley deduction

Item form: $[i, A \to \alpha \bullet \beta, j]$ (state that $S \stackrel{*}{\Rightarrow} w_1 \cdots w_i A\gamma$, and also that $\alpha \stackrel{*}{\Rightarrow} w_{i+1} \cdots w_j$)

Axioms: $[0, S' \to \bullet\ S, 0]$

Goals: $[0, S' \to S\ \bullet, n]$

Parsing: Earley deduction

Inference rules:

Scan
$$\frac{[i, A \to \alpha \bullet w_{j+1}\beta, j]}{[i, A \to \alpha w_{j+1} \bullet \beta, j+1]}$$

Predict
$$\frac{[i, A \to \alpha \bullet B\beta, j]}{[j, B \to \bullet\gamma, j]} \quad B \to \gamma$$

Complete
$$\frac{[i, A \to \alpha \bullet B\beta, k] \quad [k, B \to \gamma\bullet, j]}{[i, A \to \alpha B \bullet \beta, j]}$$

Earley's parsing algorithm

```
Parse ::
   enteredge([0, S' → • S, 0])
   for j := 1 to n do
     for every rule A → w_j do
       enteredge([j-1, A → w_j•, j])

if S' → S • ∈ C[0,n] then accept else reject
```


Earley's parsing algorithm

```
enteredge(i,edge,j) ::
   if edge ∉ C[i,j] then /* occurs check */
     C[i,j] := C[i,j] ∪ {edge}
     if edge is active then /* predict */
       for edge' ∈ rightsisters(edge) do
         enteredge([j,edge',j])
     if edge is passive then /* complete */
       for edge' ∈ leftsisters(edge) do
         for k such that edge' ∈ C[k,i] do
           enteredge([k,edge'*edge,j])
```


Complexity of natural languages

Computational complexity: the expressive power of (and the resources needed in order to process) classes of languages

Linguistic complexity: what makes individual constructions or sentences more difficult to understand

> This is the dog, that worried the cat, that killed the rat, that ate the malt, that lay in the house that Jack built.
> This is the malt that the rat that the cat that the dog worried killed ate.

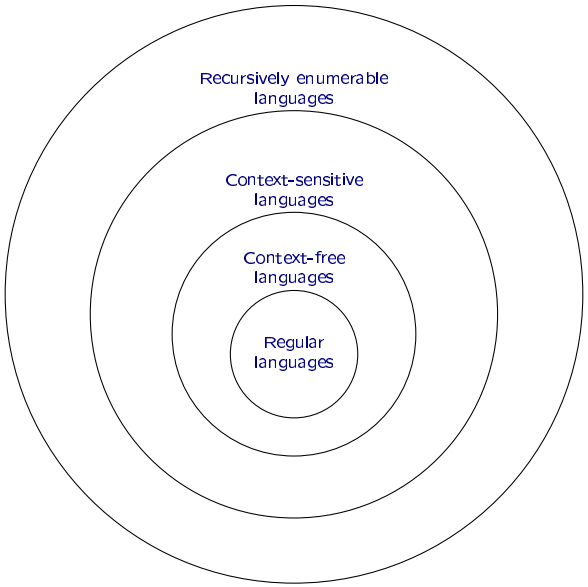
The Chomsky hierarchy of languages

A hierarchy of *classes* of languages, viewed as sets of strings, ordered by their "complexity". The higher the language is in the hierarchy, the more "complex" it is.

In particular, the class of languages in one class properly includes the languages in lower classes.

There exists a correspondence between the class of languages and the format of phrase-structure rules necessary for generating all its languages. The more restricted the rules are, the lower in the hierarchy the languages they generate are.

The Chomsky hierarchy of languages



The Chomsky hierarchy of languages

Regular (type-3) languages:

Grammar: right-linear or left-linear grammars
Rule form: $A \to \alpha$ where $A \in V$ and $\alpha \in \Sigma^* \cdot V \cup \{\epsilon\}$.
Computational device: finite-state automata

The Chomsky hierarchy of languages

Context-free (type-2) languages:

Grammar: context-free grammars
Rule form: $A \to \alpha$ where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$
Computational device: push-down automata

The Chomsky hierarchy of languages

Context-sensitive (type-1) languages:

Grammar: context-sensitive grammars
Rule form: $\alpha \to \beta$ where $|\beta| \geq |\alpha|$
Computational device: linear-bounded automata (Turing machines with a finite tape, linearly bounded by the length of the input string)

The Chomsky hierarchy of languages

Recursively-enumerable (type-0) languages:

Grammar: general rewriting systems
Rule form: $\alpha \to \beta$ where $\alpha \neq \epsilon$
Computational device: Turing machines

Where are natural languages located?

Why is it interesting?

The hierarchy represents some informal notion of the complexity of natural languages

It can help accept or reject linguistic theories

It can shed light on questions of human processing of language

Where are natural languages located?

What exactly is the question?

When viewed as a set of strings, is English a regular language? Is it context-free? How about Hebrew?

Competence vs. Performance

> This is the dog, that worried the cat, that killed the rat, that ate the malt, that lay in the house that Jack built.
> This is the malt that the rat that the cat that the dog worried killed ate.

Where are natural languages located?

Chomsky (1957):

"English is not a regular language"

As for context-free languages, "I do not know whether or not English is itself literally outside the range of such analyses"

How not to do it

An introduction to the principles of transformational syntax (Akmajian and Heny, 1976):

> Since there seem to be no way of using such PS rules to represent an obviously significant generalization about one language, namely, English, we can be sure that phrase structure grammars cannot possibly represent *all* the significant aspects of language structure. We must introduce a new kind of rule that will permit us to do so.

How not to do it

Syntax (Peter Culicover, 1976):

> In general, for any phrase structure grammar containing a finite number of rules like (2.5), (2.52) and (2.54) it will always be possible to construct a sentence that the grammar will not generate. In fact, because of recursion there will always be an infinite number of such sentences. Hence, the phrase structure analysis will not be sufficient to generate English.

How not to do it

Transformational grammar (Grinder & Elgin, 1973)

the girl saw the boy

$*$*the girl kiss the boy*

this well-known syntactic phenomenon demonstrates clearly the inadequacy of ... context-free phrase-structure grammars...

How not to do it

the defining characteristic of a context-free rule is that the symbol to be rewritten is to be rewritten without reference to the context in which it occurs... Thus, by definition, one cannot write a context-free rule that will expand the symbol V into *kiss* in the context of being immediately preceded by the sequence *the girls* and that will expand the symbol V into *kisses* in the context of being immediately preceded by the sequence *the girl*. In other words, any set of context-free rules that generate (correctly) the sequences *the girl kisses the boy* and *the girls kiss the boy* will also generate (incorrectly) the sequences *the girl kiss the boy* and *the girls kisses the boy*.

How not to do it

The grammatical phenomenon of Subject-Predicate agreement is sufficient to guarantee the accuracy of: "English is not a context-free language".

How not to do it

Syntactic theory (Bach 1974):

These examples show that to describe the facts of English number agreement is literally impossible using a simple agreement rule of the type given in a phrase-structure grammar, since we cannot guarantee that the noun phrase that determines the agreement will precede (or even be immediately adjacent) to the present-tense verb.

How not to do it

A realistic transformational grammar (Bresnan, 1978):

> in many cases the number of a verb agrees with that of a noun phrase at some distance from it... this type of syntactic dependency can extend as far as memory or patience permits... the distant type of agreement... cannot be adequately described even by context-sensitive phrase-structure rules, for the possible context is not correctly describable as a finite string of phrases.

How not to do it

What is the source for this confusion?

The notion of "context-freeness"

How to do it right

Proof techniques:

- The pumping lemma for regular languages
- Closure under intersection
- Closure under homomorphism

English is not a regular language

Center embedding:

The following is a sequence of grammatical English sentences:

A white male hired another white male.

A white male – whom a white male hired – hired another white male.

A white male – whom a white male, whom a white male hired, hired – hired another white male.

Therefore, the language L_{trg} is a subset of English:

$L_{trg} = \{$ *A white male (whom a white male)*^{n} *(hired)*^{n} *hired another white male* $\mid n > 0\}$

English is not a regular language

- L_{trg} is *not* a regular language
- L_{trg} is the intersection of the natural language English with the regular set

  $L_{reg} = \{$ *A white male (whom a white male)* (hired)* hired another white male* $\}$
- L_{reg} is regular, as it is defined by a regular expression.

Since the regular languages are closed under intersection, and since L_{reg} is a regular language, then if English were regular, its intersection with L_{reg}, namely L_{trg}, would be regular. Since L_{trg} is trans-regular, English is *not* a regular language.

English is not a regular language

Similar constructions:

The cat likes tuna fish

The cat the dog chased likes tuna fish

The cat the dog the rat bit chased likes tuna fish

The cat the dog the rat the elephant admired bit chased likes tuna fish

$$(the + N)^n \; Vt^{n-1} \; likes \; tuna \; fish$$

English is not a regular language

Another construction:

If S_1 then S_2

Either S_3 or S_4

Through a homomorphism that maps *if, then* to a and *either, or* to b, and all other words to ϵ, English can be mapped to the trans-context-free language $\{xx^R \mid x \in \{a+b\}^*\}$

Is English context-free?

The common proof technique for showing that a language is not context-free is the pumping lemma for context-free languages, and two closure properties: closure under homomorphisms and under intersection with regular languages.

Some languages that are *not* context-free:

$$\{xx \mid x \in \{a+b\}^*\}$$
$$\{a^m b^n c^m d^n\}$$

Natural languages are not context-free

Data from Swiss-German:

Jan säit das (Jan said that)

mer em Hans es huus hälfed aastriiche
we Hans-DAT the house-ACC helped paint
"we helped Hans paint the house"

mer d'chind em Hans es huus haend
we the kids-ACC Hans-DAT the house-ACC have
wele laa hälfe aastriiche
wanted to let help paint
"we have wanted to let the kids help Hans paint the house"

Natural languages are not context-free

Dative NP's must precede accusative NP's, and dative-taking verbs must precede accusative-taking verbs:

Jan säit das mer (d'chind)* (em Hans)* es huus haend wele laa* hälfe* aastriiche

However, the number of verbs requiring dative objects (*hälfe*) must equal the number of dative NP's (*em Hans*), and similarly for accusatives. Intersecting the language defined by the above regular expression with Swiss-German yields

Jan säit das mer (d'chind)^m (em Hans)^n es huus haend wele laa^m hälfe^n aastriiche

which is trans-context-free.

Linguistic complexity

Why are some sentences more difficult to understand than others?

The cat the dog the rat bit chased likes tuna fish

Limitations on stack size?

Weak and strong generative capacity

If formal language theory, the natural equivalence relation on grammars is $G_1 \equiv G_2$ iff $L(G_1) = L(G_2)$.

When grammars for natural languages are involved, we say that G_1 and G_2 are *weakly equivalent* if their string languages are identical.

Two grammars are *strongly equivalent* if they are weakly equivalent and, in addition, assign the same structure to each sentence.

Weak and strong generative capacity

The *strong generative capacity* (or power) of a linguistic formalism is its ability to associate structure to strings.

Even if context-free grammars are weakly sufficient for generating all natural languages, their strong generative capacity is probably not sufficient for natural languages.

A fragment of English

E_0 is a small fragment of English consisting of very simple sentences, constructed with only intransitive and transitive (but no ditransitive) verbs, common nouns, proper names, pronouns and determiners. Typical sentences are:

A sheep drinks

Rachel herds the sheep

Jacob loves her

A fragment of English

Similar strings are not E_0- (and, hence, English-) sentences:

*Rachel feed the sheep

*Rachel feeds herds the sheep

*The shepherds feeds the sheep

*Rachel feeds

*Jacob loves she

*Jacob loves Rachel the sheep

*Them herd the sheep

A fragment of English

All E_0 sentences have two components, a subject, realized as a noun phrase, and a predicate, realized as a verb phrase.

A noun phrase can either be a proper name, such as Rachel, or a pronoun, such as they, or a common noun, possibly preceded by a determiner: the lamb or three sheep.

A verb phrase consists of a verb, such as feed or sleeps, with a possible additional object, which is a noun phrase.

A fragment of English

Furthermore, there are constraints on the combination of phrases in E_0:

- The subject and the predicate must agree on number and person: if the subject is a third person singular, so must the verb be.
- Objects complement only – and all – the transitive verbs.
- When a pronoun is used, it is in the nominative case if it is in the subject position, and in the accusative case if it is an object.

A context-free grammar, G_0, for E_0

S	$\rightarrow$	NP VP
VP	$\rightarrow$	V
VP	$\rightarrow$	V NP
NP	$\rightarrow$	D N
NP	$\rightarrow$	Pron
NP	$\rightarrow$	PropN
D	$\rightarrow$	the, a, two, every, ...
N	$\rightarrow$	sheep, lamb, lambs, shepherd, water ...
V	$\rightarrow$	sleep, sleeps, love, loves, feed, feeds, herd, herds, ...
Pron	$\rightarrow$	I, me, you, he, him, she, her, it, we, us, they, them
PropN	$\rightarrow$	Rachel, Jacob, ...

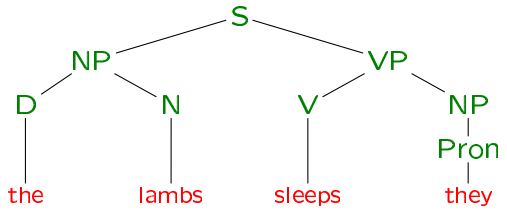
Problems of G_0

Over-generation (agreement constraints are not imposed):

- *Rachel feed the sheep
- *The shepherds feeds the sheep
- *Rachel feeds
- *Jacob loves she
- *Them herd the sheep

Problems of G_0

Over-generation:



Problems of G_0

Over-generation (subcategorization constraints are not imposed):

the lambs sleep

Jacob loves Rachel

*the lambs sleep the sheep

*Jacob loves

Methodological properties of the CFG formalism

1. Concatenation is the only string combination operation
2. Phrase structure is the only syntactic relationship
3. The terminal symbols have no properties
4. Non-terminal symbols (grammar variables) are atomic
5. Most of the information encoded in a grammar lies in the production rules
6. Any attempt of extending the grammar with a semantics requires extra means.

Alternative methodological properties

1. Concatenation is not necessarily the only way by which phrases may be combined to yield other phrases.
2. Even if concatenation is the sole string operation, other syntactic relationships are being put forward.
3. Modern computational formalisms for expressing grammars adhere to an approach called lexicalism.
4. Some formalisms do not retain any context-free backbone. However, if one is present, its categories are not atomic.
5. The expressive power added to the formalisms allows also a certain way for representing semantic information.

Extending the CFG formalism

Formal issues:

- Motivation: imposing on a grammar for E_0 two of the restrictions violated by G_0: person and number agreement
- Introducing feature structures
- Extending the terminal symbols of a grammar
- Generalizing phrases and rules
- Unification grammars
- Imposing case control

Overview

Linguistic issues:

- Subcategorization
- Feature structures for representing complex categories
- Subcategorization revisited
- Long-distance dependencies
- Subject/object control
- Coordination

Motivation

The core idea is to incorporate into the grammar *properties* of symbols, in terms of which the violations of G_0 were stated.

CFGs can be extended by associating feature structures with the terminal and non-terminal symbols of the grammar.

To represent feature structures graphically we use a notation known as attribute-value matrices (AVMs).

Adding features to words

Words (terminal symbols) are endowed with structural information. The collection of enriched terminals is the grammar's lexicon.

Example: A lexicon

lamb
$$\begin{bmatrix} \text{NUM}: & sg \\ \text{PERS}: & third \end{bmatrix}$$

lambs
$$\begin{bmatrix} \text{NUM}: & pl \\ \text{PERS}: & third \end{bmatrix}$$

I
$$\begin{bmatrix} \text{NUM}: & sg \\ \text{PERS}: & first \end{bmatrix}$$

sheep
$$\begin{bmatrix} \text{NUM}: & [\,] \\ \text{PERS}: & third \end{bmatrix}$$

dreams
$$\begin{bmatrix} \text{NUM}: & sg \\ \text{PERS}: & third \end{bmatrix}$$

Complex values

Values can either be atomic, such as *sg*, or complex:

Example: A complex feature structure

lambs
$$\begin{bmatrix} \text{AGR}: & \begin{bmatrix} \text{NUM}: & pl \\ \text{PERS}: & third \end{bmatrix} \end{bmatrix}$$

How to group features?

Variables

Example: Two notations for variables

$$\left[\text{AGR}: \; X\left(\left[\begin{array}{ll}\text{NUM}: & \textit{pl}\\ \text{PERS}: & \textit{third}\end{array}\right]\right)\right] \qquad \left[\text{AGR}: \; \boxed{1}\left[\begin{array}{ll}\text{NUM}: & \textit{pl}\\ \text{PERS}: & \textit{third}\end{array}\right]\right]$$

Properties of feature structures

- Attribute-value matrices
- Equality and reentrancy
- Subsumption
- Unification
- Generalization
- Representing lists

Attribute-value matrices

An AVM is a *syntactic* object, which can be either *atomic* or a *complex*.

Each AVM is associated with a variable.

An atomic feature structure is a variable, associated with an *atom*, drawn from a fixed set ATOMS.

A complex feature structure is a variable, associated with a finite, possibly empty set of pairs, where each pair consists of a *feature* and a *value*. Features are drawn from a fixed (per grammar), pre-defined set FEATS; values are, recursively, AVMs themselves.

Attribute-value matrices

$$\boxed{1}\left[\text{AGR}: \; \boxed{2}\left[\begin{array}{ll}\text{NUM}: & \boxed{3}\,\textit{pl}\\ \text{PERS}: & \boxed{4}\,\textit{third}\end{array}\right]\right]$$

- *pl*, *third* $\in$ ATOMS
- AGR, NUM, PERS $\in$ FEATS

Attribute-value matrices

Thus, FEATS and ATOMS, as well as the (infinite) set of variables, are parameters for the collection of AVMs, and are referred to as the signature.

We use meta-variables $\mathrm{F}, \mathrm{G}, \mathrm{H}$ to range over FEATS and A, B, C to range over feature structures.

Attribute-value matrices

Let

$$A = \boxed{i_0} \begin{bmatrix} \mathrm{F}_1 : & \boxed{i_1} A_1 \\ \vdots & \vdots \\ \mathrm{F}_n : & \boxed{i_n} A_n \end{bmatrix}$$

- $dom(A)$
- $[\,]$
- functionality: $\mathrm{F}_i \neq \mathrm{F}_j$
- $val(A, \mathrm{F}_i) = A_i$

Well-formed AVMs

Since variables are used to denote value sharing, there is not much sense in associating the same variable with two different values.

Example: Well-formed AVMs

$$A = Z\left(\begin{bmatrix} \mathrm{F} : & X(a) \\ \mathrm{G} : & Y\left(\begin{bmatrix} \mathrm{H} : & X(a) \end{bmatrix}\right) \end{bmatrix}\right), B = \boxed{4} \begin{bmatrix} \mathrm{F} : & \boxed{1} \begin{bmatrix} \mathrm{H} : & \boxed{2} a \end{bmatrix} \\ \mathrm{G} : & \boxed{1} b \end{bmatrix},$$

$$C = Z\left(\begin{bmatrix} \mathrm{F} : & X\left(\begin{bmatrix} \mathrm{H} : & Y(a) \end{bmatrix}\right) \\ \mathrm{G} : & X\left(\begin{bmatrix} \mathrm{K} : & W(b) \end{bmatrix}\right) \end{bmatrix}\right)$$

Conventions

We assume in the sequel that AVMs are well-formed.

Since multiple occurrences of the same variables always are associated with the same values, we usually only make explicit one instance of this value, leaving the other occurrences of the same variable unspecified.

Whenever a variable is associated with the empty AVM we omit the AVM itself and leave the variable only.

If a variable occurs only once in an AVM, we usually do not depict it (as it carries no additional information).

Conventions

Example: Shorthand notation for AVMs
Using our shorthand notation, the AVM A of example is depicted thus:

$$\begin{bmatrix} \text{F}: & X(a) \\ \text{G}: & \begin{bmatrix} \text{H}: & X \end{bmatrix} \end{bmatrix}$$

Conventions

A well-formed AVM:

$$\boxed{4}\begin{bmatrix} \text{F}: & \boxed{1}a \\ \text{G}: & \boxed{2}\begin{bmatrix} \text{F}: & \boxed{3}[\,] \\ \text{H}: & \boxed{1}a \end{bmatrix} \end{bmatrix}$$

Conventions

Removing multiple values of multiply-occurring variables:

$$\boxed{4}\begin{bmatrix} \text{F}: & \boxed{1}a \\ \text{G}: & \boxed{2}\begin{bmatrix} \text{F}: & \boxed{3}[\,] \\ \text{H}: & \boxed{1} \end{bmatrix} \end{bmatrix}$$

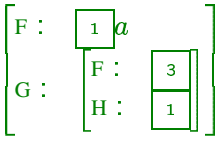
Conventions

Removing the empty AVM:

$$\boxed{4}\begin{bmatrix} \text{F}: & \boxed{1}a \\ \text{G}: & \boxed{2}\begin{bmatrix} \text{F}: & \boxed{3} \\ \text{H}: & \boxed{1} \end{bmatrix} \end{bmatrix}$$

Conventions

Removing non-informative variables:



Paths

A *path* is a (possibly empty) sequence of features that can be used to pick a value in a feature structure.

We use angular brackets '$\langle \ldots \rangle$' to depict paths explicitly.

Paths

For example, in the AVM A of example , the single feature $\langle \text{F} \rangle$ constitutes a path; and so does the sequence $\langle \text{G,H} \rangle$, since $\langle \text{G} \rangle$ can be used to pick the value $\begin{bmatrix} \text{H}: & X(a) \end{bmatrix}$ (because $val(A, \text{G}) = \begin{bmatrix} \text{H}: & X(a) \end{bmatrix}$).

$$A = \begin{bmatrix} \text{F}: & X(a) \\ \text{G}: & \begin{bmatrix} \text{H}: & X \end{bmatrix} \end{bmatrix}$$

Paths

The notion of values is extended from features to paths: $val(A, \pi)$ is the value obtained by following the path π in A; this value (if defined) is again a feature structure.

If A_i is the value of some path π in A then A_i is said to be a *sub-AVM* of A. The *empty path* is denoted ϵ, and $val(A, \epsilon) = A$ for every feature structure A.

Paths

Example: Basic notions
Let

$$A = \begin{bmatrix} \text{AGR} : & \begin{bmatrix} \text{NUM} : & pl \\ \text{PERS} : & third \end{bmatrix} \end{bmatrix}$$

Then

$$dom(A) = \{\text{AGR}\}, val(A, \text{AGR}) = \begin{bmatrix} \text{NUM} : & pl \\ \text{PERS} : & third \end{bmatrix}.$$

The paths of A are $\{\epsilon, \langle \text{AGR} \rangle, \langle \text{AGR,NUM} \rangle, \langle \text{AGR,PERS} \rangle\}$. The values of these paths are: $val(A, \epsilon) = A$, $val(A, \langle \text{AGR,NUM} \rangle) = pl$, $val(A, \langle \text{AGR,PERS} \rangle) = third$. Since there is no path $\langle \text{NUM,AGR} \rangle$ in A, $val(A, \langle \text{NUM,AGR} \rangle)$ is undefined.

Equality and reentrancy

When are two *atomic* feature structures equal?

Two atomic feature structures, whose variables are associated with one and the same atom, are not necessarily identical:

$$\begin{bmatrix} \text{F} : & X(a) \\ \text{G} : & Y(a) \end{bmatrix}$$

To ensure such identity, associate the same variable with the two values:

$$\begin{bmatrix} \text{F} : & X(a) \\ \text{G} : & X \end{bmatrix}$$

Equality and reentrancy

The features F and G are *reentrant*; a feature structure is reentrant if it contains (at least two) reentrant features.

To denote reentrancy in some feature structure A we use the symbol '$\overset{A}{\leftrightsquigarrow}$':

$$A_3 = \begin{bmatrix} \text{F}_1 : & \begin{bmatrix} \text{G} : & \boxed{1} \begin{bmatrix} \text{H} : & b \end{bmatrix} \end{bmatrix} \\ \text{F}_2 : & \begin{bmatrix} \text{G} : & \boxed{1} \end{bmatrix} \end{bmatrix}$$

$\langle \text{F}_1, \text{G} \rangle \overset{A_3}{\leftrightsquigarrow} \langle \text{F}_2, \text{G} \rangle$.

Equality and reentrancy

Token identity vs. *type identity*

There is no path that can distinguish between the following two (non-identical!) structures:

$$A = \begin{bmatrix} \text{F} : & a \\ \text{G} : & a \end{bmatrix} \quad B = \begin{bmatrix} \text{F} : & \boxed{1} a \\ \text{G} : & \boxed{1} \end{bmatrix}$$

Thus, feature structures are intensional objects.

When referring to type identity, we use the '=' symbol; to denote token identity we use the symbol '$\doteq$'.

$val(A, \pi_1) \doteq val(A, \pi_2)$ when $\pi_1 \overset{A}{\leftrightsquigarrow} \pi_2$.

Type-identity vs. token-identity

Example: Type-identity vs. token-identity

$$A = \begin{bmatrix} \text{SUBJ} : & \begin{bmatrix} \text{AGR} : & \begin{bmatrix} \text{NUM} : & sg \\ \text{PERS} : & third \end{bmatrix} \end{bmatrix} \\ \text{OBJ} : & \begin{bmatrix} \text{AGR} : & \begin{bmatrix} \text{NUM} : & sg \\ \text{PERS} : & third \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$$B = \begin{bmatrix} \text{SUBJ} : & \boxed{4} \begin{bmatrix} \text{AGR} : & \begin{bmatrix} \text{NUM} : & sg \\ \text{PERS} : & third \end{bmatrix} \end{bmatrix} \\ \text{OBJ} : & \boxed{4} \end{bmatrix}$$

Type-identity vs. token-identity

Example: Type-identity and token-identity revisited
Consider again the following feature structures:

$$A = \begin{bmatrix} \text{SUBJ} : & \begin{bmatrix} \text{AGR} : & \begin{bmatrix} \text{NUM} : & sg \\ \text{PERS} : & third \end{bmatrix} \end{bmatrix} \\ \text{OBJ} : & \begin{bmatrix} \text{AGR} : & \begin{bmatrix} \text{NUM} : & pl \\ \text{PERS} : & third \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$$B = \begin{bmatrix} \text{SUBJ} : & \boxed{4} \begin{bmatrix} \text{AGR} : & \begin{bmatrix} \text{NUM} : & sg \\ \text{PERS} : & third \end{bmatrix} \end{bmatrix} \\ \text{OBJ} : & \boxed{4} \end{bmatrix}$$

$val(A, \text{SUBJ}) = val(A, \text{OBJ})$ and $val(B, \text{SUBJ}) = val(B, \text{OBJ})$.
However, while $val(B, \text{SUBJ}) \doteq val(B, \text{OBJ})$, $val(A, \text{SUBJ}) \neq val(A, \text{OBJ})$.

Example: (continued)
Suppose that by some action a feature CASE with the value *nom* is added to the value of SUBJ in both A and B:

$$A' = \begin{bmatrix} \text{SUBJ} : & \begin{bmatrix} \text{AGR} : & \begin{bmatrix} \text{NUM} : & sg \\ \text{PERS} : & third \end{bmatrix} \\ \text{CASE} : & nom \end{bmatrix} \\ \text{OBJ} : & \begin{bmatrix} \text{AGR} : & \begin{bmatrix} \text{NUM} : & pl \\ \text{PERS} : & third \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$val(A', \text{SUBJ}) \neq val(A', \text{OBJ})$!

Example: (continued)

$$B' = \begin{bmatrix} \text{SUBJ} : & \boxed{4} \begin{bmatrix} \text{AGR} : & \begin{bmatrix} \text{NUM} : & sg \\ \text{PERS} : & third \end{bmatrix} \\ \text{CASE} : & nom \end{bmatrix} \\ \text{OBJ} : & \boxed{4} \end{bmatrix}$$

Here, $val(B', \text{SUBJ}) \doteq val(B', \text{OBJ})$, implying $val(B', \text{SUBJ}) = val(B', \text{OBJ})$.

Cycles

A special case of reentrancy is cyclicity: an AVM can contain a path whose value is the AVM itself. In other words, an AVM can be reentrant with a sub-structure of itself:

$$A = \begin{bmatrix} \text{F}: & \boxed{2} \begin{bmatrix} \text{G}: & a \\ \text{H}: & \boxed{2} \end{bmatrix} \end{bmatrix}$$

Renaming

If an AVM can be obtained from some other AVM through a systematic renaming of its variables, we say that each of the AVMs is a *renaming* of the other.

Example: Renaming
The following AVMs are renamings, as B can be obtained by renaming all the occurrences of the variable $\boxed{1}$ in A to $\boxed{4}$, and all the occurrences of $\boxed{2}$ to $\boxed{6}$:

$$A = \boxed{3} \begin{bmatrix} \text{F}: & \boxed{2} \begin{bmatrix} \text{G}: & \boxed{1} a \\ \text{H}: & \boxed{2} \end{bmatrix} \\ \text{G}: & \boxed{1} \end{bmatrix} \qquad B = \boxed{3} \begin{bmatrix} \text{F}: & \boxed{6} \begin{bmatrix} \text{G}: & \boxed{4} a \\ \text{H}: & \boxed{6} \end{bmatrix} \\ \text{G}: & \boxed{4} \end{bmatrix}$$

This is a good time to stop.

Exercises?

Subsumption

Let A, B be feature structures over the same signature. We say that A subsumes B ($A \sqsubseteq B$; also, A is more general than B, and B is subsumed by, or is more specific than, A) if the following conditions hold:

1. if A is an atomic AVM then B is an atomic AVM with the same atom;
2. for every $\text{F} \in \textsc{Feats}$, if $\text{F} \in dom(A)$ then $\text{F} \in dom(B)$, and $val(A, \text{F})$ subsumes $val(B, \text{F})$; and
3. if two paths are reentrant in A, they are also reentrant in B: if $\pi_1 \overset{A}{\leftrightsquigarrow} \pi_2$ then $\pi_1 \overset{B}{\leftrightsquigarrow} \pi_2$.

Subsumption

Example: Subsumption

$$[\,] \sqsubseteq \begin{bmatrix}\text{NUM}: & sg\end{bmatrix}$$

$$\begin{bmatrix}\text{NUM}: & X\end{bmatrix} \sqsubseteq \begin{bmatrix}\text{NUM}: & sg\end{bmatrix}$$

$$\begin{bmatrix}\text{NUM}: & sg\end{bmatrix} \sqsubseteq \begin{bmatrix}\text{NUM}: & sg\\ \text{PERS}: & third\end{bmatrix}$$

$$\begin{bmatrix}\text{NUM1}: & sg\\ \text{NUM2}: & sg\end{bmatrix} \sqsubseteq \begin{bmatrix}\text{NUM1}: & \boxed{1}\ sg\\ \text{NUM2}: & \boxed{1}\end{bmatrix}$$

Subsumption

Subsumption is a partial relation: not every pair of feature structures is comparable:

$$\begin{bmatrix}\text{NUM}: & sg\end{bmatrix} \not\sqsubseteq \not\sqsupseteq \begin{bmatrix}\text{NUM}: & pl\end{bmatrix}$$

A different case of incomparability is caused by the existence of different features in the two structures:

$$\begin{bmatrix}\text{NUM}: & sg\end{bmatrix} \not\sqsubseteq \not\sqsupseteq \begin{bmatrix}\text{PERS}: & third\end{bmatrix}$$

Subsumption

Example: Subsumption
While subsumption informally encodes an order of information content among AVMs, sometimes the informal notion can be misleading:

$$\begin{bmatrix}\text{NUM}: & sg\\ \text{PERS}: & third\end{bmatrix} \not\sqsubseteq \not\sqsupseteq \begin{bmatrix}\text{AGR}: & \begin{bmatrix}\text{NUM}: & sg\\ \text{PERS}: & third\end{bmatrix}\end{bmatrix}$$

Subsumption

Some properties of subsumption:

Least element: the empty feature structure subsumes every feature structure: for every feature structure A, $[\,] \sqsubseteq A$

Refelxivity: for every feature structure A, $A \sqsubseteq A$

Transitivity: If $A \sqsubseteq B$ and $B \sqsubseteq C$ than $A \sqsubseteq C$.

Antisymmetry: Subsumption is antisymmetric: if $A \sqsubseteq B$ and $B \sqsubseteq A$ then $A = B$.

To summarize, subsumption is a partial, reflexive, transitive and antisymmetric relation; it is therefore a *partial order*.

Unification

The unification operation, denoted '$\sqcup$', is defined over pairs of feature structures, and yields the most general feature structure that is more specific than both operands, if one exists:

$A = B \sqcup C$ if and only if A is the most general feature structure such that $B \sqsubseteq A$ and $C \sqsubseteq A$.

If such a structure exists, the unification succeeds, and the two arguments are said to be unifiable (or consistent).

If none exists, the unification fails, and the operands are said to be inconsistent.

Unification

Example: Unification

Unification combines consistent information:

$$\left[\text{NUM}: \quad sg\right] \sqcup \left[\text{PERS}: \quad third\right] = \begin{bmatrix}\text{NUM}: & sg \\ \text{PERS}: & third\end{bmatrix}$$

Different atoms are inconsistent: $\left[\text{NUM}: \quad sg\right] \sqcup \left[\text{NUM}: \quad pl\right] = \top$

Example: (continued)

Atoms and non-atoms are inconsistent: $\left[\text{NUM}: \quad sg\right] \sqcup sg = \top$

Example: (continued)

Unification is absorbing:

$$\left[\text{NUM}: \quad sg\right] \sqcup \begin{bmatrix}\text{NUM}: & sg \\ \text{PERS}: & third\end{bmatrix} = \begin{bmatrix}\text{NUM}: & sg \\ \text{PERS}: & third\end{bmatrix}$$

Example: (continued)

Empty feature structures are identity elements:

$$[\,] \sqcup \begin{bmatrix} \text{AGR}: & \begin{bmatrix} \text{NUM}: & sg \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \text{AGR}: & \begin{bmatrix} \text{NUM}: & sg \end{bmatrix} \end{bmatrix}$$

Example: (continued)

Reentrancy causes two consistent values to coincide:

$$\begin{bmatrix} \text{F}: & \begin{bmatrix} \text{NUM}: & sg \end{bmatrix} \\ \text{G}: & \begin{bmatrix} \text{PERS}: & third \end{bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \text{F}: & \boxed{1} \\ \text{G}: & \boxed{1} \end{bmatrix} = \begin{bmatrix} \text{F}: & \boxed{1} \begin{bmatrix} \text{NUM}: & sg \\ \text{PERS}: & third \end{bmatrix} \\ \text{G}: & \boxed{1} \end{bmatrix}$$

Example: (continued)

Variables can be (partially) instantiated:

$$\begin{bmatrix} \text{F}: & X \end{bmatrix} \sqcup \begin{bmatrix} \text{F}: & \begin{bmatrix} \text{H}: & b \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \text{F}: & X(\begin{bmatrix} \text{H}: & b \end{bmatrix}) \end{bmatrix}$$

Example: (continued)

Unification acts differently depending on whether the values are equal:

$$\begin{bmatrix} \text{F}: & \begin{bmatrix} \text{NUM}: & sg \end{bmatrix} \\ \text{G}: & \begin{bmatrix} \text{NUM}: & sg \end{bmatrix} \end{bmatrix} \sqcup \begin{bmatrix} \text{F}: & \begin{bmatrix} \text{PERS}: & third \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \text{F}: & \begin{bmatrix} \text{NUM}: & sg \\ \text{PERS}: & 3rd \end{bmatrix} \\ \text{G}: & \begin{bmatrix} \text{NUM}: & sg \end{bmatrix} \end{bmatrix}$$

...or identical:

$$\begin{bmatrix} \text{F}: & \boxed{1} \begin{bmatrix} \text{NUM}: & sg \end{bmatrix} \\ \text{G}: & \boxed{1} \end{bmatrix} \sqcup \begin{bmatrix} \text{F}: & \begin{bmatrix} \text{PERS}: & 3rd \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \text{F}: & \boxed{1} \begin{bmatrix} \text{NUM}: & sg \\ \text{PERS}: & 3rd \end{bmatrix} \\ \text{G}: & \boxed{1} \end{bmatrix}$$

Variable binding

Unification *binds* variables together. Let:

$$A = \begin{bmatrix} \text{F}: & \boxed{1} \begin{bmatrix} \text{NUM}: & sg \end{bmatrix} \end{bmatrix} \qquad B = \begin{bmatrix} \text{F}: & \boxed{2} \begin{bmatrix} \text{PERS}: & third \end{bmatrix} \end{bmatrix}$$

Then:

$$A \sqcup B = \begin{bmatrix} \text{F}: & \boxed{1}\boxed{2} \begin{bmatrix} \text{NUM}: & sg \\ \text{PERS}: & third \end{bmatrix} \end{bmatrix}$$

Of course, since the variables $\boxed{1}$ and $\boxed{2}$ occur nowhere else, they can be simply omitted and the result is equal to:

$$A \sqcup B = \begin{bmatrix} \text{F}: & \begin{bmatrix} \text{NUM}: & sg \\ \text{PERS}: & third \end{bmatrix} \end{bmatrix}$$

Variable binding

However, had either $\boxed{1}$ or $\boxed{2}$ occurred elsewhere (for example, as the value of some feature G in A), their values would have been modified as a result of the unification:

$$\begin{bmatrix} \text{F}: & \boxed{1} \begin{bmatrix} \text{NUM}: & sg \end{bmatrix} \\ \text{G}: & \boxed{1} \end{bmatrix} \sqcup \begin{bmatrix} \text{F}: & \boxed{2} \begin{bmatrix} \text{PERS}: & third \end{bmatrix} \end{bmatrix} =$$

$$\begin{bmatrix} \text{F}: & \boxed{3} \begin{bmatrix} \text{NUM}: & sg \\ \text{PERS}: & third \end{bmatrix} \\ \text{G}: & \boxed{3} \end{bmatrix}$$

Unification

Some properties of unification:

Idempotency: $A \sqcup A = A$

Commutativity: $A \sqcup B = B \sqcup A$

Associativity: $A \sqcup (B \sqcup C) = (A \sqcup B) \sqcup C$

Absorption: If $A \sqsubseteq B$ then $A \sqcup B = B$

Monotonicity: If $A \sqsubseteq B$ then for every C, $A \sqcup C \sqsubseteq B \sqcup C$ (if both exist).

Generalization

Generalization (denoted $\sqcap$) is the operation that returns the most specific (or least general) feature structure that is still more general than both arguments.

Unlike unification, generalization can never fail. For every pair of feature structures there exists a feature structure that is more general than both: in the most extreme case, pick the empty feature structure, which is more general than every other structure.

Generalization

Example: Generalization

Generalization reduces information:

$$\begin{bmatrix}\text{NUM}: & sg\end{bmatrix} \sqcap \begin{bmatrix}\text{PERS}: & third\end{bmatrix} = [\,]$$

Different atoms are inconsistent:

$$\begin{bmatrix}\text{NUM}: & sg\end{bmatrix} \sqcap \begin{bmatrix}\text{NUM}: & pl\end{bmatrix} = \begin{bmatrix}\text{NUM}: & [\,]\end{bmatrix}$$

Example: (continued)

Generalization is restricting:

$$\begin{bmatrix}\text{NUM}: & sg\end{bmatrix} \sqcap \begin{bmatrix}\text{NUM}: & sg \\ \text{PERS}: & third\end{bmatrix} = \begin{bmatrix}\text{NUM}: & sg\end{bmatrix}$$

Example: (continued)

Empty feature structures are zero elements:

$$[\,] \sqcap \begin{bmatrix}\text{AGR}: & \begin{bmatrix}\text{NUM}: & sg\end{bmatrix}\end{bmatrix} = [\,]$$

Reentrancies can be lost:

$$\begin{bmatrix}\text{F}: & \boxed{1}\ \begin{bmatrix}\text{NUM}: & sg\end{bmatrix} \\ \text{G}: & \boxed{1}\end{bmatrix} \sqcap \begin{bmatrix}\text{F}: & \begin{bmatrix}\text{NUM}: & sg\end{bmatrix} \\ \text{G}: & \begin{bmatrix}\text{NUM}: & sg\end{bmatrix}\end{bmatrix} = \begin{bmatrix}\text{F}: & \begin{bmatrix}\text{NUM}: & sg\end{bmatrix} \\ \text{G}: & \begin{bmatrix}\text{NUM}: & sg\end{bmatrix}\end{bmatrix}$$

Generalization

Some properties of generalization:

Idempotency: $A \sqcap A = A$

Commutativity: $A \sqcap B = B \sqcap A$

Absorption: If $A \sqsubseteq B$ then $A \sqcap B = A$

Using feature structures for representing lists

Feature structures can be easily used to encode (finite) lists. As an example, consider the following representation of the list $\langle 1,2,3\rangle$ (assuming a signature whose atoms include the numbers $1,2,3$):

Example: Feature structure encoding of a list

$$\begin{bmatrix} \text{FIRST}: & 1 \\ \text{REST}: & \begin{bmatrix} \text{FIRST}: & 2 \\ \text{REST}: & \begin{bmatrix} \text{FIRST}: & 3 \\ \text{REST}: & \mathit{elist} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Using feature structures for representing lists

Example: A nested list
The list $\langle\langle 1,2,6,7\rangle,\langle 3,4\rangle,\langle 5\rangle\rangle$:

$$\begin{bmatrix} \text{FIRST}: & \begin{bmatrix} \text{FIRST}: & 1 \\ \text{REST}: & \begin{bmatrix} \text{FIRST}: & 2 \\ \text{REST}: & \begin{bmatrix} \text{FIRST}: & 6 \\ \text{REST}: & \begin{bmatrix} \text{FIRST}: & 7 \\ \text{REST}: & \mathit{elist} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \text{REST}: & \begin{bmatrix} \text{FIRST}: & \begin{bmatrix} \text{FIRST}: & 3 \\ \text{REST}: & \begin{bmatrix} \text{FIRST}: & 4 \\ \text{REST}: & \mathit{elist} \end{bmatrix} \end{bmatrix} \\ \text{REST}: & \begin{bmatrix} \text{FIRST}: & \begin{bmatrix} \text{FIRST}: & 5 \\ \text{REST}: & \mathit{elist} \end{bmatrix} \\ \text{REST}: & \mathit{elist} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Adding features to rules

Phrases, like words, have valued features and consequently, grammar non-terminals, too, are decorated with features.

When a feature is assigned to a non-terminal symbol C, it means that this feature is appropriate for *all* the phrases of category C: it makes sense for it to appear in all the instances of C.

Such categories interact, in the grammar, with other AVMs, through application of rules, and the specified values might thus undergo changes. In general, AVMs are changed as a result of rule application.

We refer to such enriched categories as *generalized categories* (or *extended* ones), which have a base category and an associated feature structure.

Adding features to rules

Example:
A feature structure that might be associated with phrases of category *NP* (noun phrases).

$$\mathit{NP}$$
$$\begin{bmatrix} \text{NUM}: & [\,] \\ \text{PERS}: & [\,] \end{bmatrix}$$

A third person singular noun phrase such as lamb may be associated with:

$$\mathit{NP}$$
$$\begin{bmatrix} \text{NUM}: & \mathit{sg} \\ \text{PERS}: & \mathit{third} \end{bmatrix}$$

Extended grammar rules

Example: Rules for imposing number agreement

(1) $\underset{[\text{NUM}: \ X]}{N} \rightarrow \underset{[\text{NUM}: \ X(sg)]}{lamb}$

(2) $\underset{[\text{NUM}: \ X]}{N} \rightarrow \underset{[\text{NUM}: \ X(pl)]}{lambs}$

(3) $S \rightarrow \underset{[\text{NUM}: \ X]}{NP} \ \underset{[\text{NUM}: \ X]}{VP}$

(4) $\underset{[\text{NUM}: \ X]}{NP} \rightarrow \underset{[\text{NUM}: \ X]}{D} \ \underset{[\text{NUM}: \ X]}{N}$

Scope of variables

The scope of a variable is the grammar rule in which it occurs. Reformulating rule (2) as

$$\underset{[\text{NUM}: \ Y]}{N} \rightarrow \underset{[\text{NUM}: \ Y(pl)]}{lambs}$$

has no effect.

No sharing is implied by occurrences of the same variables in different rules, for example the occurrences of X in rules (1) and (2) above.

Declarativity

Rule (4) stipulates that in order to form a noun phrase (NP) from the concatenation of a determiner (D) and a noun (N), the NUM features of the determiner and the noun must agree.

The NUM feature of the noun phrase thus constructed is equal to that of either daughter.

What the rule does not determine is an order of this value check.

The undirectional view of agreement is a typical view of unification-based grammar formalisms.

Extending AVMs

Multi-AVMs can be viewed as sequences of AVMs, with the important observation that some sub-structures can be shared among two or more AVMSs.

In other words, the scope of variables is extended from a single AVM to a multi-AVM: the same variable can be associated with two sub-structures of different AVMs.

The notion of well-formedness is extended to multi-AVMs.

The function val, associating a value with features (and paths) has to be extended, too.

If the value of the path π_1 leaving the i-th root of σ is reentrant with the value of the path π_2 leaving the j-th root, we write $(i, \pi_1) \overset{\sigma}{\leftrightsquigarrow} (j, \pi_2)$.

Extending AVMs

Example: Multi-AVM

Let σ be the multi-AVM: $\begin{bmatrix} \text{F}: & \begin{bmatrix} \text{G}: & a \\ \text{H}: & X \end{bmatrix} \end{bmatrix} \quad \begin{bmatrix} \text{G}: & Y \end{bmatrix} \quad \begin{bmatrix} \text{F}: & \begin{bmatrix} \text{H}: & b \\ \text{G}: & X \end{bmatrix} \\ \text{H}: & a \end{bmatrix}$

Then $val(\sigma, 1, \langle \text{F} \rangle)$ is: $\begin{bmatrix} \text{G}: & a \\ \text{H}: & [\,] \end{bmatrix}$

whereas $val(\sigma, 3, \langle \text{F} \rangle)$ is: $\begin{bmatrix} \text{H}: & b \\ \text{G}: & [\,] \end{bmatrix}$

In this example, $(1, \langle \text{F H} \rangle) \overset{\sigma}{\leftrightsquigarrow} (3, \langle \text{F G} \rangle)$.

Example: (continued)

A multi-AVM can have an empty feature structure as an element:

$$\begin{bmatrix} \text{F}: & \begin{bmatrix} \text{G}: & a \\ \text{H}: & X \end{bmatrix} \end{bmatrix} \quad [\,] \quad \begin{bmatrix} \text{F}: & \begin{bmatrix} \text{H}: & b \\ \text{G}: & X \end{bmatrix} \\ \text{H}: & a \end{bmatrix}$$

Extending AVMs

The following is a valid multi-AVM:

$$\begin{bmatrix} \text{F}: & a \\ \text{G}: & \boxed{2} \end{bmatrix} \quad \boxed{2} \begin{bmatrix} \text{H}: & b \end{bmatrix}$$

The only restriction is that the same variable cannot be associated with two different elements in the sequence. Thus, the following is not a multi-AVM:

$$\boxed{2} \begin{bmatrix} \text{H}: & b \end{bmatrix} \quad \begin{bmatrix} \text{F}: & a \\ \text{G}: & \boxed{2} \end{bmatrix} \quad \boxed{2}$$

Subsumption

The notion of subsumption can be naturally extended from AVMs to multi-AVMs: if σ and ρ are two multi-AVMs *of the same length*, n, then $\sigma \sqsubseteq \rho$ if the following conditions hold:

1. every element of σ subsumes the corresponding element of ρ: for every i, $1 \leq i \leq n$, $val(\sigma, i, \epsilon) \sqsubseteq val(\rho, i, \epsilon)$; and
2. if two paths are reentrant in σ, they are also reentrant in ρ: if $(i, \pi_1) \overset{\sigma}{\leftrightsquigarrow} (j, \pi_2)$ then $(i, \pi_1) \overset{\rho}{\leftrightsquigarrow} (j, \pi_2)$.

Subsumption

Example: Multi-AVM subsumption

Let σ be: $\begin{bmatrix} \text{F}: & \begin{bmatrix} \text{G}: & a \\ \text{H}: & X \end{bmatrix} \end{bmatrix} \quad \begin{bmatrix} \text{G}: & c \end{bmatrix} \quad \begin{bmatrix} \text{F}: & \begin{bmatrix} \text{H}: & b \\ \text{G}: & X(d) \end{bmatrix} \\ \text{H}: & a \end{bmatrix}$

and ρ be: $\begin{bmatrix} \text{F}: & \begin{bmatrix} \text{G}: & a \\ \text{H}: & d \end{bmatrix} \end{bmatrix} \quad \begin{bmatrix} \text{G}: & c \end{bmatrix} \quad \begin{bmatrix} \text{F}: & \begin{bmatrix} \text{H}: & b \\ \text{G}: & d \end{bmatrix} \\ \text{H}: & a \end{bmatrix}$

Then σ does not subsume ρ, but $\rho \sqsubseteq \sigma$.

Unification

In the same way, the notion of unification can be extended to multi-AVMs (of the same length): we say that ρ is the unification of σ_1 and σ_2 (and write $\rho = \sigma_1 \sqcup \sigma_2$) if σ_1, σ_2 and ρ are of the same length, and ρ is the most general multi-AVM that is more specific than both σ_1 and σ_2.

Rules and grammars

An extended context-free rule consists of two components: a context-free rule, and a multi-AVM of the same length.

A unification grammar consists of a set of extended context-free rules and an extended category that serves as the *start symbol*.

Unification grammars

Example: G_1, a unification grammar for E_0

(1) $S \quad \rightarrow \quad \overset{NP}{\begin{bmatrix} \text{NUM}: & X \end{bmatrix}} \quad \overset{VP}{\begin{bmatrix} \text{NUM}: & X \end{bmatrix}}$

(2) $\overset{NP}{\begin{bmatrix} \text{NUM}: & X \end{bmatrix}} \quad \rightarrow \quad \overset{D}{\begin{bmatrix} \text{NUM}: & X \end{bmatrix}} \quad \overset{N}{\begin{bmatrix} \text{NUM}: & X \end{bmatrix}}$

(3) $\overset{VP}{\begin{bmatrix} \text{NUM}: & X \end{bmatrix}} \quad \rightarrow \quad \overset{V}{\begin{bmatrix} \text{NUM}: & X \end{bmatrix}}$

Example: (continued)

(4) $\underset{[\text{NUM}: X]}{VP} \rightarrow \underset{[\text{NUM}: X]}{V} \quad \underset{[\text{NUM}: Y]}{NP}$

(5,6) $\underset{[\text{NUM}: X]}{N} \rightarrow \underset{[\text{NUM}: X(sg)]}{lamb} \mid \underset{[\text{NUM}: X]}{sheep} \mid .$

(7,8) $\underset{[\text{NUM}: X]}{V} \rightarrow \underset{[\text{NUM}: X(sg)]}{sleeps} \mid \underset{[\text{NUM}: X(pl)]}{sleep} \mid .$

(9,10) $\underset{[\text{NUM}: X]}{D} \rightarrow \underset{[\text{NUM}: X(sg)]}{a} \mid \underset{[\text{NUM}: X(pl)]}{two} \mid .$

Rule application

- Forms and sentential forms
- Derivations
- Derivation trees
- Language

Forms

Forms are generalized and are composed of "sequences" of generalized categories, that is, of a sequence of base categories or words, augmented by a multi-AVM of the same length.

We use Greek letters such as α, β as meta-variables over forms. For example, following is a form of length two:

$$\underset{[\text{NUM}: Y]}{NP} \quad \underset{[\text{NUM}: Y]}{VP}$$

Derivations

Derivation is a binary relation over generalized forms. Let α be a generalized form, and $B_0 \rightarrow B_1 B_2 \ldots B_k$ a grammar rule, where the B_i are all generalized categories, and where reentrancies might occur among elements of the form or the rule. Application of the rule to α consists of the following steps:

- Matching the rule's head with some element of the form that has the same base category;
- Replacing the selected element in the form with the body of the rule, producing a new form.

Derivations

Example: Matching
Suppose that

$$\alpha = \underset{[\text{NUM}: \ Y]}{NP} \quad \underset{[\text{NUM}: \ Y]}{VP}$$

is a (sentential) form and that

$$\rho = \underset{[\text{NUM}: \ X]}{VP} \rightarrow \underset{[\text{NUM}: \ X]}{V} \quad \underset{[\text{NUM}: \ W]}{NP}$$

is a rule. Let the selected element of α be its second element, namely the extended category

$$\underset{[\text{NUM}: \ Y]}{VP}$$

Example: (continued)
This extended category matches the head of the rule ρ, as the base categories are identical (*VP*) and the AVMs associated with them are unifiable (consistent). The result of the unification is the extended category

$$\underset{[\text{NUM}: \ Z]}{VP}$$

which is equivalent to

$$\underset{[\text{NUM}: \ [\,]]}{VP}$$

An additional effect of the unification is that the variables Y of the form and X of the rule are unified, too.

Replacement

The two feature structures (associated with the head of the rule and with the selected element) are unified in their respective contexts: the body of the rule and the form.

When some variable X in the form is unified with some variable Y in the rule, all occurrences of X in the form and of Y in the rule are modified: they are all set to the unified value.

The replacement operation inserts the modified rule body into the modified form, replacing the selected element of the form.

The variables of the resulting form are then systematically renamed.

Derivation

Example: Derivation step
Let

$$\alpha = \underset{[\text{NUM}: \ Y]}{NP} \quad \underset{[\text{NUM}: \ Y]}{VP}$$

$$\rho = \underset{[\text{NUM}: \ X]}{VP} \rightarrow \underset{[\text{NUM}: \ X]}{V} \quad \underset{[\text{NUM}: \ W]}{NP}$$

be a form and a rule, respectively. The unification of the rule's head with the second element of α succeeds, and identifies the values of X and Y . After replacement and variable renaming we obtain:

$$\beta = \underset{[\text{NUM}: \ X_1]}{NP} \quad \underset{[\text{NUM}: \ X_1]}{V} \quad \underset{[\text{NUM}: \ W_1]}{NP}$$

Example: (continued)
Now assume that the (terminal) rule

$$\overset{V}{\left[\text{NUM}: \ Y\right]} \rightarrow \overset{herds}{\left[\text{NUM}: \ Y(sg)\right]}$$

is to be applied to β. The value of the variable X_1 in the form is set, through unification, to sg, and the resulting form is:

$$\gamma = \overset{NP}{\left[\text{NUM}: \ X_2\right]} \ \overset{herds}{\left[\text{NUM}: \ X_2(sg)\right]} \ \overset{NP}{\left[\text{NUM}: \ W_2\right]}$$

Note that the first NP had its feature structure modified, even though it did not participate directly in the rule application.

Derivation

Example: Derivation step (continued)
Assume now that γ is expanded by applying to its first element the rule

$$\overset{NP}{\left[\text{NUM}: \ X\right]} \rightarrow \overset{D}{\left[\text{NUM}: \ X\right]} \ \overset{N}{\left[\text{NUM}: \ X\right]}$$

In this case, unification of the first element of γ with the head of the rule binds the value of X in the rule to *sg*:

$$\delta = \overset{D}{\left[\text{NUM}: \ X_3\right]} \ \overset{N}{\left[\text{NUM}: \ X_3\right]} \ \overset{herds}{\left[\text{NUM}: \ X_3(sg)\right]} \ \overset{NP}{\left[\text{NUM}: \ W_3\right]}$$

Example: (continued)
If we now tried to apply the (terminal) rule

$$\overset{D}{\left[\text{NUM}: \ Y\right]} \rightarrow \overset{two}{\left[\text{NUM}: \ Y(pl)\right]}$$

to the first element of δ, this attempt would have caused unification failure.

Derivation

Example: Derivation with ϵ-rules
Let

$$\alpha = \overset{A}{\left[\text{F}: \ X\right]} \ \overset{B}{\begin{bmatrix}\text{F}: & X\\ \text{G}: & Y\end{bmatrix}} \ \overset{C}{\left[\text{G}: \ Y\right]}, \quad \rho = \overset{B}{\begin{bmatrix}\text{F}: & Z\\ \text{G}: & Z\end{bmatrix}} \rightarrow \epsilon$$

be a form and a rule, respectively. Applying the rule to the second element of the form yields:

$$\overset{A}{\left[\text{F}: \ W\right]} \ \overset{C}{\left[\text{G}: \ W\right]}$$

Derivation

Example: Derivation can modify information
Let

$$\alpha = \underset{[\text{F}: \ a]}{A} \ \underset{[\text{G}: \ b]}{B}, \quad \rho = \underset{[\text{F}: \ a]}{A} \rightarrow \underset{[\text{F}: \ c]}{A}$$

be a form and a rule, respectively. Applying the rule to the first element of the form yields:

$$\underset{[\text{F}: \ c]}{A} \ \underset{[\text{G}: \ b]}{B}$$

Notice that in the result, the value of F in A was modified from a to c.

Derivation

The full derivation relation is, as usual, the reflexive-transitive closure of rule application.

A form is *sentential* if it is derivable from the start symbol.

Derivation

A derivation of the sentence two sheep sleep with the grammar G_1. After each rule is applied, the variables in the obtained form are renamed.

Example: Derivation
The derivation starts with the start symbol, which is the extended category S. Applying rule (1), one gets:

$$\underset{[\text{NUM}: \ X_1]}{NP} \ \underset{[\text{NUM}: \ X_1]}{VP}$$

Example: (continued)
It is now possible to select the leftmost element in the above sentential form and to apply rule (2). Renaming all occurrences of X in rule (2) to X_2, one gets the following sentential form:

$$\underset{[\text{NUM}: \ X_2]}{D} \ \underset{[\text{NUM}: \ X_2]}{N} \ \underset{[\text{NUM}: \ X_2]}{VP}$$

Now select the rightmost element in the above form and apply rule (3), renaming all occurrences of X_2 to X_3:

$$\underset{[\text{NUM}: \ X_3]}{D} \ \underset{[\text{NUM}: \ X_3]}{N} \ \underset{[\text{NUM}: \ X_3]}{V}$$

Example: (continued)
The leftmost element is selected, and (the terminal) rule (10) is applied, binding X_3 to pl:

$$\underset{\text{two}}{[num: X_3(pl)]} \quad \underset{N}{[\text{NUM}: X_3]} \quad \underset{V}{[\text{NUM}: X_3]}$$

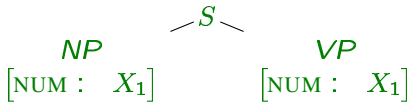
In the same way, rule (6) can be applied to the middle element in this form, and rule (8) to the rightmost, resulting in:

$$\underset{\text{two}}{[\text{NUM}: X_3(pl)]} \quad \underset{\text{sheep}}{[\text{NUM}: X_3(pl)]} \quad \underset{\text{sleep}}{[\text{NUM}: X_3(pl)]}$$

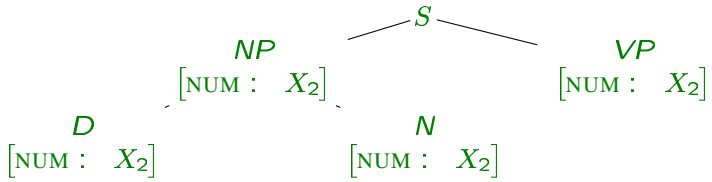
Thus the string two sheep sleep is indeed a sentence.

Derivation trees

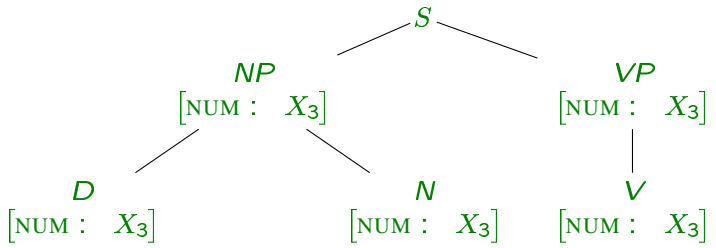
Example: Snapshots of a derivation sequence
We begin with the start symbol, the extended category S, which is expanded by applying rule (1), yielding (after renaming):



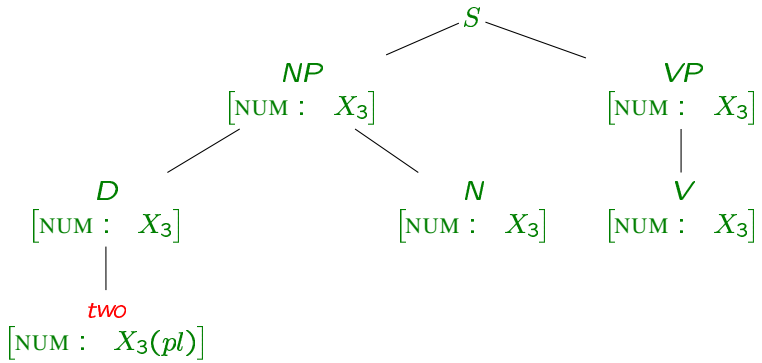
Example: (continued)
The next step is the application of rule (2) to the leftmost element in the frontier of the tree. Since this application results in binding X_1 with X_2, we rename all occurrences of X_1 in the tree to X_2, obtaining the following tree:



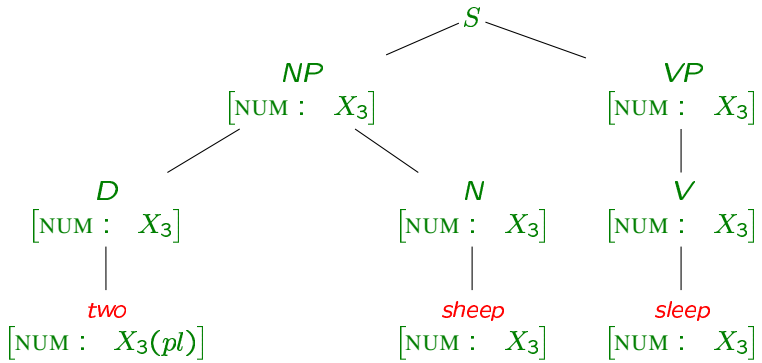
Example: (continued)
Now select the rightmost element in the frontier of the above tree and apply rule (3), renaming all occurrences of X_2 in the tree to X_3; the following tree is obtained:



Example: (continued)
Next, the leftmost element is selected, and (the terminal) rule (10) is applied, binding X_3 to pl:



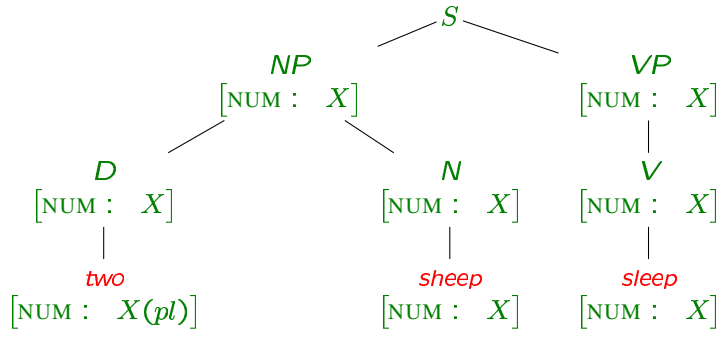
Example: (continued)
Similarly, rule (6) can be applied to the middle element in the frontier, and rule (8) to the rightmost, yielding:



Derivation trees

The final derivation tree for the same sentence:

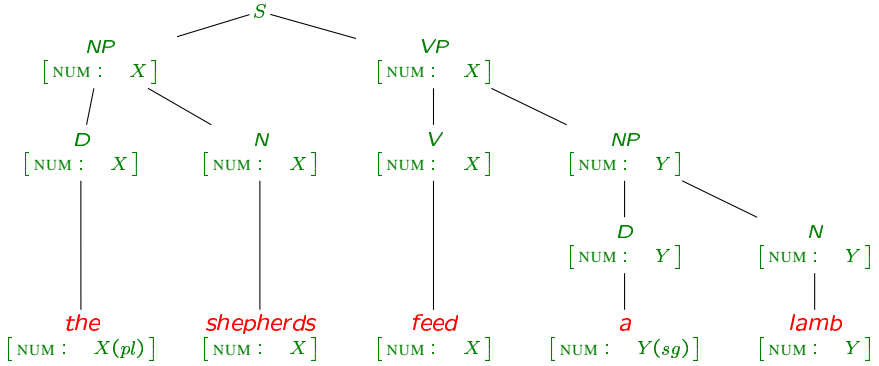
Example: Derivation tree



Derivation trees

The final derivation tree for the sentence the shepherds feed a lamb:

Example: Derivation tree



Languages

To determine whether a sequence of words, $w = a_1 \cdots a_n$, is in $L(G)$, consider a derivation in G whose first form consists of the start symbol (an extended category, viewed as an extended form of length 1), and whose last form is $\langle w, \sigma' \rangle$.

Let $\langle w, \sigma \rangle$ be an extended form obtained by concatenating $A_1, \ldots, A_n$, where each A_i is a lexical entry of the word a_1.

We say that $w \in L(G)$ if and only if σ' be a multi-AVM that is unifiable with σ: $\sigma \sqcup \sigma'$ does not fail.

Languages

Example: Language
Given this definition, observe, for example, that the string two sheep sleep is in the language generated by the example grammar G_1; we have seen a derivation sequence for this string. The first and the last elements of this sequence, namely the feature structures associated with the words two and sleep, are identical to lexical entries of G_1. However, the middle element, namely the feature structure associated with sheep, is more specific than (subsumed by) the lexical entry of sheep.

Languages

The language generated by the grammar G_1 is context free:

Example: A context-free grammar G'_1

$S \rightarrow S_{sg} \mid S_{pl}$

$S_{sg} \rightarrow NP_{sg}\ VP_{sg}$	$S_{pl} \rightarrow NP_{pl}\ VP_{pl}$
$NP_{sg} \rightarrow D_{sg}\ N_{sg}$	$NP_{pl} \rightarrow D_{pl}\ N_{pl}$
$VP_{sg} \rightarrow V_{sg}$	$VP_{pl} \rightarrow V_{pl}$
$VP_{sg} \rightarrow V_{sg}\ NP_{sg} \mid V_{sg}\ NP_{pl}$	$VP_{pl} \rightarrow V_{pl}\ NP_{sg} \mid V_{pl}\ NP_{pl}$
$D_{sg} \rightarrow$ *a*	$D_{pl} \rightarrow$ *two*
$N_{sg} \rightarrow$ *lamb* | *sheep* | $\cdots$	$N_{pl} \rightarrow$ *lambs* | *sheep* | $\cdots$
$V_{sg} \rightarrow$ *sleeps* | $\cdots$	$V_{pl} \rightarrow$ *sleep* | $\cdots$

Imposing case control

The extensions of the CFG formalism can be used for imposing various constraints on generated languages. Here we suggest a solution for the problem of controlling the case of a noun phrase.

First, add pronouns to the grammar:

(2.1) $\begin{bmatrix} NP \\ \text{NUM}: X \end{bmatrix} \rightarrow \begin{bmatrix} D \\ \text{NUM}: X \end{bmatrix} \quad \begin{bmatrix} N \\ \text{NUM}: X \end{bmatrix}$

(2.2) $\begin{bmatrix} NP \\ \text{NUM}: X \end{bmatrix} \rightarrow \begin{bmatrix} PropN \\ \text{NUM}: X \end{bmatrix}$

(2.3) $\begin{bmatrix} NP \\ \text{NUM}: X \end{bmatrix} \rightarrow \begin{bmatrix} Pron \\ \text{NUM}: X \end{bmatrix}$

Imposing case control

Additionally, the following terminal rules are needed:

$$\underset{\left[\text{NUM}: \; sg\right]}{\textit{PropN}} \quad \rightarrow \quad \text{Jacob} \mid \text{Rachel} \mid \ldots$$

$$\underset{\left[\text{NUM}: \; sg\right]}{\textit{Pron}} \quad \rightarrow \quad \text{she} \mid \text{her} \mid \ldots$$

Imposing case control

The additional rules allow sentences such as

She herds the sheep

Jacob loves her

but also non-sentences such as

*Her herds the sheep

*Jacob loves she

Imposing case control

We add a feature, CASE, to the feature structures associated with nominal categories: nouns, pronouns, proper names and noun phrases.

What should the values of the CASE feature be?

Imposing case control

(11) $\underset{\left[\begin{smallmatrix}\text{NUM}: & X \\ \text{CASE}: & Y\end{smallmatrix}\right]}{\textit{PropN}} \rightarrow \underset{\left[\begin{smallmatrix}\text{NUM}: & X(sg) \\ \text{CASE}: & Y\end{smallmatrix}\right]}{\textit{Rachel}}$

(12) $\underset{\left[\begin{smallmatrix}\text{NUM}: & X \\ \text{CASE}: & Y\end{smallmatrix}\right]}{\textit{PropN}} \rightarrow \underset{\left[\begin{smallmatrix}\text{NUM}: & X(sg) \\ \text{CASE}: & Y\end{smallmatrix}\right]}{\textit{Jacob}}$

(13) $\underset{\left[\begin{smallmatrix}\text{NUM}: & X \\ \text{CASE}: & Y\end{smallmatrix}\right]}{\textit{Pron}} \rightarrow \underset{\left[\begin{smallmatrix}\text{NUM}: & X(sg) \\ \text{CASE}: & Y(nom)\end{smallmatrix}\right]}{\textit{she}}$

(14) $\underset{\left[\begin{smallmatrix}\text{NUM}: & X \\ \text{CASE}: & Y\end{smallmatrix}\right]}{\textit{Pron}} \rightarrow \underset{\left[\begin{smallmatrix}\text{NUM}: & X(sg) \\ \text{CASE}: & Y(acc)\end{smallmatrix}\right]}{\textit{her}}$

Imposing case control

Percolating the value of the CASE feature from the lexical entries to the category NP:

(2.1) $\underset{\left[\begin{smallmatrix}\text{NUM}: & X\\ \text{CASE}: & Y\end{smallmatrix}\right]}{NP} \rightarrow \underset{[\text{NUM}: X]}{D} \quad \underset{\left[\begin{smallmatrix}\text{NUM}: & X\\ \text{CASE}: & Y\end{smallmatrix}\right]}{N}$

(2.2) $\underset{\left[\begin{smallmatrix}\text{NUM}: & X\\ \text{CASE}: & Y\end{smallmatrix}\right]}{NP} \rightarrow \underset{\left[\begin{smallmatrix}\text{NUM}: & X\\ \text{CASE}: & Y\end{smallmatrix}\right]}{PropN}$

(2.3) $\underset{\left[\begin{smallmatrix}\text{NUM}: & X\\ \text{CASE}: & Y\end{smallmatrix}\right]}{NP} \rightarrow \underset{\left[\begin{smallmatrix}\text{NUM}: & X\\ \text{CASE}: & Y\end{smallmatrix}\right]}{Pron}$

Imposing case control

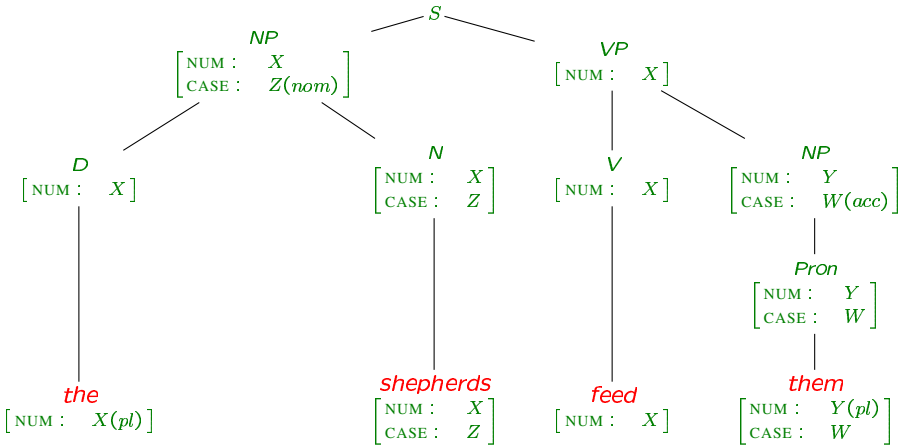
Imposing the constraint:

(1′) $S \rightarrow \underset{\left[\begin{smallmatrix}\text{NUM}: & X\\ \text{CASE}: & nom\end{smallmatrix}\right]}{NP} \quad \underset{[\text{NUM}: X]}{VP}$

(4′) $\underset{[\text{NUM}: X]}{VP} \rightarrow \underset{[\text{NUM}: X]}{V} \quad \underset{\left[\begin{smallmatrix}\text{NUM}: & Y\\ \text{CASE}: & acc\end{smallmatrix}\right]}{NP}$

Derivation tree with case control

Example: Derivation tree with case control



Example: (continued)
This tree represents a derivation which starts with the initial symbol, *S*, and ends with multi-AVM σ', where

$$\sigma' = \underset{\text{the}}{[\text{NUM}: X(pl)]} \quad \underset{\text{shepherds}}{\left[\begin{smallmatrix}\text{NUM}: & X\\ \text{CASE}: & Z\end{smallmatrix}\right]} \quad \underset{\text{feed}}{[\text{NUM}: X]} \quad \underset{\text{them}}{\left[\begin{smallmatrix}\text{NUM}: & Y(pl)\\ \text{CASE}: & W(acc)\end{smallmatrix}\right]}$$

This multi-AVM is unifiable with (but not identical to!) the sequence of lexical entries of the words in the sentence, which is:

$$\sigma = \underset{\text{the}}{[\text{NUM}: [\,]]} \quad \underset{\text{shepherds}}{\left[\begin{smallmatrix}\text{NUM}: & pl\\ \text{CASE}: & [\,]\end{smallmatrix}\right]} \quad \underset{\text{feed}}{[\text{NUM}: pl]} \quad \underset{\text{them}}{\left[\begin{smallmatrix}\text{NUM}: & pl\\ \text{CASE}: & acc\end{smallmatrix}\right]}$$

Imposing subcategorization constraints

We use the extended formalism for a naïve solution to the subcategorization problem; reminder:

intransitive verbs: sleep, walk, run, laugh, ...

transitive verbs (with a nominal object): feed, love, eat, ...

Imposing subcategorization constraints

First, the lexical entries of verbs are extended:

Example: Lexical entries for verbs

$$\overset{V}{\begin{bmatrix}\text{NUM}: & X\\ \text{SUBCAT}: & intrans\end{bmatrix}} \rightarrow \overset{sleeps}{\begin{bmatrix}\text{NUM}: & X(sg)\end{bmatrix}} \mid \overset{sleep}{\begin{bmatrix}\text{NUM}: & X(pl)\end{bmatrix}} \mid \ldots$$

$$\overset{V}{\begin{bmatrix}\text{NUM}: & X\\ \text{SUBCAT}: & trans\end{bmatrix}} \rightarrow \overset{feeds}{\begin{bmatrix}\text{NUM}: & X(sg)\end{bmatrix}} \mid \overset{feed}{\begin{bmatrix}\text{NUM}: & X(pl)\end{bmatrix}} \mid \ldots$$

Imposing subcategorization constraints

Second, the rules that involve verbs and verb phrases are extended:

Example: Modified rules for verb phrases

(4.1) $$\overset{VP}{\begin{bmatrix}\text{NUM}: & X\end{bmatrix}} \rightarrow \overset{V}{\begin{bmatrix}\text{NUM}: & X\\ \text{SUBCAT}: & intrans\end{bmatrix}}$$

(4.2) $$\overset{VP}{\begin{bmatrix}\text{NUM}: & X\end{bmatrix}} \rightarrow \overset{V}{\begin{bmatrix}\text{NUM}: & X\\ \text{SUBCAT}: & trans\end{bmatrix}} \quad \overset{NP}{\begin{bmatrix}\text{NUM}: & Y\end{bmatrix}}$$

Imposing subcategorization constraints

Example: Derivation of a shepherd feeds two sheep

$$S \overset{(1)}{\Rightarrow} \overset{NP}{\begin{bmatrix}\text{NUM}: & X\end{bmatrix}} \quad \overset{VP}{\begin{bmatrix}\text{NUM}: & X\end{bmatrix}}$$

$$\overset{(2)}{\Rightarrow} \overset{D}{\begin{bmatrix}\text{NUM}: & X\end{bmatrix}} \quad \overset{N}{\begin{bmatrix}\text{NUM}: & X\end{bmatrix}} \quad \overset{VP}{\begin{bmatrix}\text{NUM}: & X\end{bmatrix}}$$

$$\overset{(4.2)}{\Rightarrow} \overset{D}{\begin{bmatrix}\text{NUM}: & X\end{bmatrix}} \quad \overset{N}{\begin{bmatrix}\text{NUM}: & X\end{bmatrix}} \quad \overset{V}{\begin{bmatrix}\text{NUM}: & X\\ \text{SUBCAT}: & trans\end{bmatrix}} \quad \overset{NP}{\begin{bmatrix}\text{NUM}: & Y\end{bmatrix}}$$

Example: (continued)

$$\overset{(4.2)}{\Rightarrow} \quad \underset{[\text{NUM}: X]}{D} \quad \underset{[\text{NUM}: X]}{N} \quad \underset{\left[\begin{smallmatrix}\text{NUM}: X\\ \text{SUBCAT}: trans\end{smallmatrix}\right]}{V} \quad \underset{[\text{NUM}: Y]}{NP}$$

$$\overset{(1)}{\Rightarrow} \quad \underset{[\text{NUM}: X]}{D} \quad \underset{[\text{NUM}: X]}{N} \quad \underset{\left[\begin{smallmatrix}\text{NUM}: X\\ \text{SUBCAT}: trans\end{smallmatrix}\right]}{V} \quad \underset{[\text{NUM}: Y]}{D} \quad \underset{[\text{NUM}}{I}$$

$$\overset{*}{\Rightarrow} \quad \underset{[\text{NUM}: sg]}{a} \quad \underset{[\text{NUM}: sg]}{shepherd} \quad \underset{\left[\begin{smallmatrix}\text{NUM}: sg\\ \text{SUBCAT}: trans\end{smallmatrix}\right]}{feeds} \quad \underset{[\text{NUM}: pl]}{two} \quad \underset{[\text{NUM}}{sh}$$

G_2, a complete E_0-grammar

Example: G_2, a complete E_0-grammar

$$\begin{array}{lcll}
S & \to & \underset{\left[\begin{smallmatrix}\text{NUM}: X\\ \text{CASE}: nom\end{smallmatrix}\right]}{NP} & \underset{[\text{NUM}: X]}{VP} \\
\underset{\left[\begin{smallmatrix}\text{NUM}: X\\ \text{CASE}: Y\end{smallmatrix}\right]}{NP} & \to & \underset{[\text{NUM}: X]}{D} & \underset{\left[\begin{smallmatrix}\text{NUM}: X\\ \text{CASE}: Y\end{smallmatrix}\right]}{N} \\
\underset{\left[\begin{smallmatrix}\text{NUM}: X\\ \text{CASE}: Y\end{smallmatrix}\right]}{NP} & \to & \underset{\left[\begin{smallmatrix}\text{NUM}: X\\ \text{CASE}: Y\end{smallmatrix}\right]}{Pron} \mid & \underset{\left[\begin{smallmatrix}\text{NUM}: X\\ \text{CASE}: Y\end{smallmatrix}\right]}{PropN} \\
\underset{[\text{NUM}: X]}{VP} & \to & \underset{\left[\begin{smallmatrix}\text{NUM}: X\\ \text{SUBCAT}: intrans\end{smallmatrix}\right]}{V} & \\
\underset{[num: X]}{VP} & \to & \underset{\left[\begin{smallmatrix}\text{NUM}: X\\ \text{SUBCAT}: trans\end{smallmatrix}\right]}{V} & \underset{\left[\begin{smallmatrix}\text{NUM}: Y\\ \text{CASE}: acc\end{smallmatrix}\right]}{NP}
\end{array}$$

Example: (continued)

$$\begin{array}{lcll}
\underset{\left[\begin{smallmatrix}\text{NUM}: X\\ \text{CASE}: Y\end{smallmatrix}\right]}{N} & \to & \underset{\left[\begin{smallmatrix}\text{NUM}: X(sg)\\ \text{CASE}: Y\end{smallmatrix}\right]}{lamb} \mid & \underset{\left[\begin{smallmatrix}\text{NUM}: X(pl)\\ \text{CASE}: Y\end{smallmatrix}\right]}{lambs} \mid \ldots \\
\underset{\left[\begin{smallmatrix}\text{NUM}: X\\ \text{CASE}: Y\end{smallmatrix}\right]}{Pron} & \to & \underset{\left[\begin{smallmatrix}\text{NUM}: X(sg)\\ \text{CASE}: Y(nom)\end{smallmatrix}\right]}{she} \mid & \underset{\left[\begin{smallmatrix}\text{NUM}: X(sg)\\ \text{CASE}: Y(acc)\end{smallmatrix}\right]}{her} \mid \ldots \\
\underset{\left[\begin{smallmatrix}\text{NUM}: X\\ \text{CASE}: Y\end{smallmatrix}\right]}{PropN} & \to & \underset{\left[\begin{smallmatrix}\text{NUM}: X(sg)\\ \text{CASE}: Y\end{smallmatrix}\right]}{Rachel} \mid & \underset{\left[\begin{smallmatrix}\text{NUM}: X(sg)\\ \text{CASE}: Y\end{smallmatrix}\right]}{Jacob} \mid \ldots \\
\underset{\left[\begin{smallmatrix}\text{NUM}: X\\ \text{SUBCAT}: intrans\end{smallmatrix}\right]}{V} & \to & \underset{[\text{NUM}: X(sg)]}{sleeps} \mid & \underset{[\text{NUM}: X(pl)]}{sleep} \mid \ldots \\
\underset{\left[\begin{smallmatrix}\text{NUM}: X\\ \text{SUBCAT}: trans\end{smallmatrix}\right]}{V} & \to & \underset{[\text{NUM}: X(sg)]}{feeds} \mid & \underset{[\text{NUM}: X(pl)]}{feed} \mid \ldots \\
\underset{[\text{NUM}: X]}{D} & \to & \underset{[\text{NUM}: X(sg)]}{a} \mid & \underset{[\text{NUM}: X(pl)]}{two} \mid \ldots
\end{array}$$

Internalizing categories

The grammars we have seen so far had an explicit context-free backbone (or skeleton), obtained by considering the (context-free) grammar induced by the base categories.

This is not imposed by the formalism; rather, the base categories might be internalized into the feature structures themselves.

Internalizing categories

For example, the rule

$$\underset{\left[\text{NUM}: \ X\right]}{NP} \rightarrow \underset{\left[\text{NUM}: \ X\right]}{D} \ \underset{\left[\text{NUM}: \ X\right]}{N}$$

can be re-written as

$$\begin{bmatrix}\text{CAT}: & np\\ \text{NUM}: & X\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & d\\ \text{NUM}: & X\end{bmatrix} \begin{bmatrix}\text{CAT}: & n\\ \text{NUM}: & X\end{bmatrix}$$

Internalizing categories

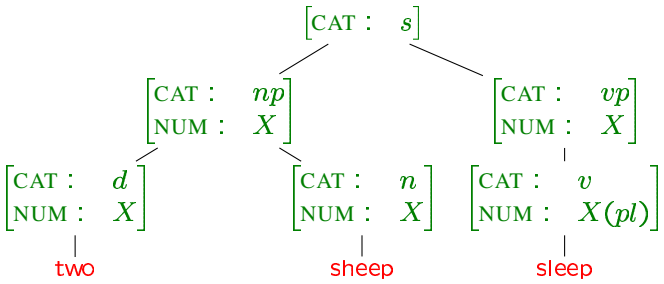
In the new presentation of grammars, productions are essentially multi-AVMs.

Derivations, derivation trees, languages...

Special features and the signature.

Internalizing categories

Example: Derivation tree



Internalizing categories

Once the base category of a phrase is admitted as the value of one of the features in the feature structure associated with that phrase, it does not have to be represented as an atomic value.

Internalizing categories

For example, the Chomskian representation of categories:

nouns: $\begin{bmatrix} \text{N}: & + \\ \text{V}: & - \end{bmatrix}$

verbs: $\begin{bmatrix} \text{N}: & - \\ \text{V}: & + \end{bmatrix}$

adjectives: $\begin{bmatrix} \text{N}: & + \\ \text{V}: & + \end{bmatrix}$

prepositions: $\begin{bmatrix} \text{N}: & - \\ \text{V}: & - \end{bmatrix}$

Internalizing categories

Internalization of the category results in additional expressive power.

It now becomes possible to consider feature structures in which the value of the CAT feature is underspecified, or even unrestricted.

For example, one might describe a phrase in singular using the feature structure

$$\begin{bmatrix} \text{CAT}: & [\,] \\ \text{NUM}: & sg \end{bmatrix}$$

Internalizing categories

Once information about the category of a phrase is embedded within the feature structure, it can be manipulated in more ways than simply encoding the category of a phrase.

Internalized categories will be used to:

- represent information about the subcategories of verbs
- list information about constituents that are "moved", or "transformed", using the *slash* notation
- account for coordination.

Subcategorization lists

Motivation: to account for the subcategorization data in a more general, elegant way, extending the coverage of our grammar from the smallest fragment E_0 to the fragment E_1.

In E_1 different verbs subcategorize for different kinds of complements: noun phrases, infinitival verb phrases, sentences etc. Also, some verbs require more than one complement.

The idea behind the solution is to store in the lexical entry of each verb not an atomic feature indicating its subcategory, but rather a list of atomic categories, indicating the appropriate complements of the verb.

Subcategorization lists

Example: Lexical entries of verbs using subcategorization lists

sleep $\begin{bmatrix} \text{CAT}: & v \\ \text{SUBCAT}: & elist \\ \text{NUM}: & pl \end{bmatrix}$

give $\begin{bmatrix} \text{CAT}: & v \\ \text{SUBCAT}: & \langle [\text{CAT}: \ np], [\text{CAT}: \ np] \rangle \\ \text{NUM}: & pl \end{bmatrix}$

love $\begin{bmatrix} \text{CAT}: & v \\ \text{SUBCAT}: & \langle [\text{CAT}: \ np] \rangle \\ \text{NUM}: & pl \end{bmatrix}$

tell $\begin{bmatrix} \text{CAT}: & v \\ \text{SUBCAT}: & \langle [\text{CAT}: \ np], [\text{CAT}: \ s] \rangle \\ \text{NUM}: & pl \end{bmatrix}$

Subcategorization lists

The grammar rules must be modified to reflect the additional wealth of information in the lexical entries:

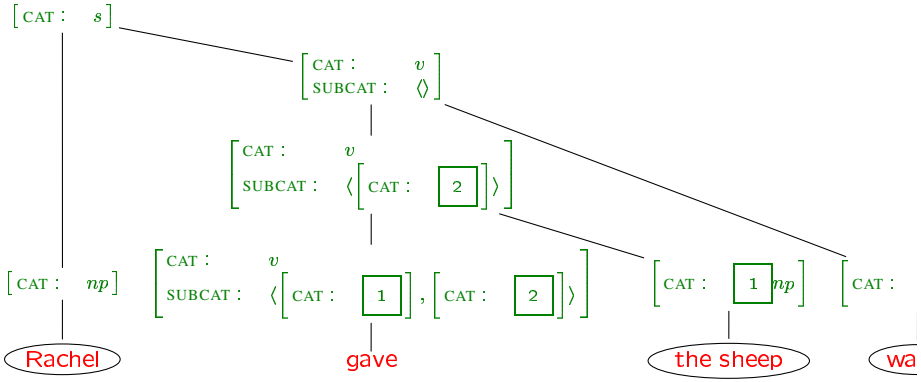
Example: VP rules using subcategorization lists

$$[\text{CAT}: \ s] \rightarrow [\text{CAT}: \ np] \begin{bmatrix} \text{CAT}: & v \\ \text{SUBCAT}: & elist \end{bmatrix}$$

$$\begin{bmatrix} \text{CAT}: & v \\ \text{SUBCAT}: & Y \end{bmatrix} \rightarrow \begin{bmatrix} \text{CAT}: & v \\ \text{SUBCAT}: & \begin{bmatrix} \text{FIRST}: & [\text{CAT}: \ X] \\ \text{REST}: & Y \end{bmatrix} \end{bmatrix} [\text{CAT}: \ X]$$

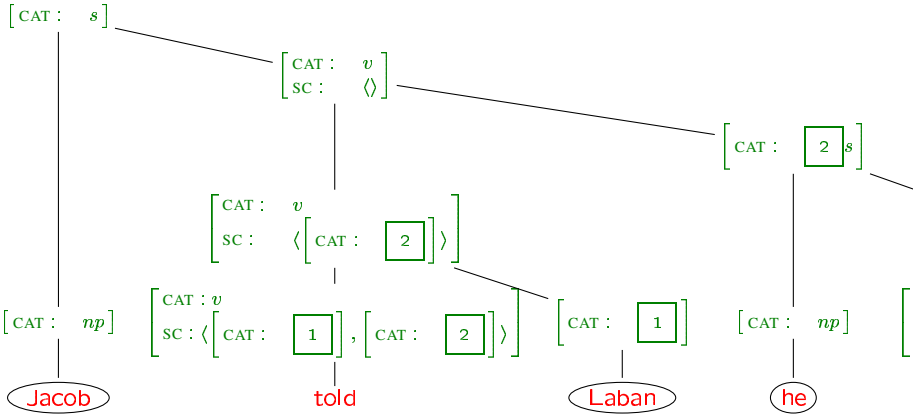
Subcategorization lists

Example: A derivation tree



Subcategorization lists

Example: A derivation tree



Subcategorization lists

In the above grammar, categories on subcategorization lists are represented as an atomic symbol.

The method outlined here can be used with more complex encodings of categories. In other words, the specification of categories in a subcategorization list can include all the constraints that the verb imposes on its complements

Subcategorization lists

Example: Subcategorization imposes case constraints

	Ich	gebe	dem	Hund	den	Knochen
	I	*give*	*the(dat)*	*dog*	*the(acc)*	*bone*
	I give the dog the bone					
*	Ich	gebe	den	Hund	den	Knochen
	I	*give*	*the(acc)*	*dog*	*the(acc)*	*bone*
*	Ich	gebe	dem	Hund	dem	Knochen
	I	*give*	*the(dat)*	*dog*	*the(dat)*	*bone*

Subcategorization lists

The lexical entry of gebe, then, could be:

$$\begin{bmatrix} \text{CAT}: & v \\ \text{SUBCAT}: & \left\langle \begin{bmatrix} \text{CAT}: & np \\ \text{CASE}: & dat \end{bmatrix}, \begin{bmatrix} \text{CAT}: & np \\ \text{CASE}: & acc \end{bmatrix} \right\rangle \\ \text{NUM}: & sg \end{bmatrix}$$

The VP rule has to be slightly modified:

$$\begin{bmatrix} \text{CAT}: & v \\ \text{SUBCAT}: & Y \end{bmatrix} \rightarrow \begin{bmatrix} \text{CAT}: & v \\ \text{SUBCAT}: & \begin{bmatrix} \text{FIRST}: & X \\ \text{REST}: & Y \end{bmatrix} \end{bmatrix} X([\,])$$

G_3, a complete E_1-grammar

Example: G_3, a complete E_1-grammar

$$\begin{bmatrix} \text{CAT}: & s \end{bmatrix} \rightarrow \begin{bmatrix} \text{CAT}: & np \\ \text{NUM}: & X \\ \text{CASE}: & nom \end{bmatrix} \begin{bmatrix} \text{CAT}: & v \\ \text{NUM}: & X \\ \text{SUBCAT}: & elist \end{bmatrix}$$

$$\begin{bmatrix} \text{CAT}: & v \\ \text{NUM}: & X \\ \text{SUBCAT}: & Y \end{bmatrix} \rightarrow \begin{bmatrix} \text{CAT}: & v \\ \text{NUM}: & X \\ \text{SUBCAT}: & \begin{bmatrix} \text{FIRST}: & Z \\ \text{REST}: & Y \end{bmatrix} \end{bmatrix} Z([\,])$$

$$\begin{bmatrix} \text{CAT}: & np \\ \text{NUM}: & X \\ \text{CASE}: & Y \end{bmatrix} \rightarrow \begin{bmatrix} \text{CAT}: & d \\ \text{NUM}: & X \end{bmatrix} \begin{bmatrix} \text{CAT}: & n \\ \text{NUM}: & X \\ \text{CASE}: & Y \end{bmatrix}$$

$$\begin{bmatrix} \text{CAT}: & np \\ \text{NUM}: & X \\ \text{CASE}: & Y \end{bmatrix} \rightarrow \begin{bmatrix} \text{CAT}: & pron \\ \text{NUM}: & X \\ \text{CASE}: & Y \end{bmatrix} \mid \begin{bmatrix} \text{CAT}: & propn \\ \text{NUM}: & X \\ \text{CASE}: & Y \end{bmatrix}$$

Example: (continued)

sleep → $\begin{bmatrix} \text{CAT}: & v \\ \text{SUBCAT}: & elist \\ \text{NUM}: & pl \end{bmatrix}$

love → $\begin{bmatrix} \text{CAT}: & v \\ \text{SUBCAT}: & \langle \begin{bmatrix} \text{CAT}: & np \\ \text{CASE}: & acc \end{bmatrix} \rangle \\ \text{NUM}: & pl \end{bmatrix}$

lamb → $\begin{bmatrix} \text{CAT}: & n \\ \text{NUM}: & sg \\ \text{CASE}: & Y \end{bmatrix}$

give → $\begin{bmatrix} \text{CAT}: & v \\ \text{SUBCAT}: & \langle \begin{bmatrix} \text{CAT}: & np \\ \text{CASE}: & acc \end{bmatrix}, [\text{CAT}: \\ \text{NUM}: & pl \end{bmatrix}$

tell → $\begin{bmatrix} \text{CAT}: & v \\ \text{SUBCAT}: & \langle \begin{bmatrix} \text{CAT}: & np \\ \text{CASE}: & acc \end{bmatrix}, [\text{CAT}: \\ \text{NUM}: & pl \end{bmatrix}$

lambs → $\begin{bmatrix} \text{CAT}: & n \\ \text{NUM}: & pl \\ \text{CASE}: & Y \end{bmatrix}$

Example: (continued)

she → $\begin{bmatrix} \text{CAT}: & pron \\ \text{NUM}: & sg \\ \text{CASE}: & nom \end{bmatrix}$

her → $\begin{bmatrix} \text{CAT}: & pron \\ \text{NUM}: & pl \\ \text{CASE}: & acc \end{bmatrix}$

Rachel → $\begin{bmatrix} \text{CAT}: & propn \\ \text{NUM}: & sg \end{bmatrix}$

Jacob → $\begin{bmatrix} \text{CAT}: & propn \\ \text{NUM}: & sg \end{bmatrix}$

a → $\begin{bmatrix} \text{CAT}: & d \\ \text{NUM}: & sg \end{bmatrix}$

two → $\begin{bmatrix} \text{CAT}: & d \\ \text{NUM}: & pl \end{bmatrix}$

Long distance dependencies

Internalized categories are very useful in the treatment of unbounded dependencies, which are included in the grammar fragment E_3.

Such phenomena involve a "missing" constituent that is realized outside the clause from which it is missing, as in:

(1) The shepherd wondered whom Jacob loved ⌙.

(2) The shepherd wondered whom Laban thought Jacob loved ⌙.

(3) The shepherd wondered whom Laban thought Rachel claimed Jacob loved ⌙.

Long distance dependencies

An attempt to replace the gap with an explicit noun phrase results in ungrammaticality:

(4) *The shepherd wondered who Jacob loved Rachel.

Long distance dependencies

The gap need not be in the object position:

(5) Jacob wondered who ⌙ loved Leah

(6) Jacob wondered who Laban believed ⌙ loved Leah

Again, an explicit noun phrase filling the gap results in ungrammaticality:

(7) Jacob wondered who the shepherd loved Leah

Long distance dependencies

More than one gap may be present in a sentence (and, hence, more than one filler):

(8a) This is the well which Jacob is likely to ⌙ draw water from ⌙

(8b) It was Leah that Jacob worked for ⌙ without loving ⌙

In some languages (e.g., Norwegian) there is no (principled) bound on the number of gaps that can occur in a single clause.

Long distance dependencies

There are other fragments of English in which long distance dependencies are manifested in other forms. *Topicalization*:

(9) Rachel, Jacob loved ⌙

(10) Rachel, every shepherd knew Jacob loved ⌙

Another example is *interrogative sentences*:

(11) Who did Jacob love ⌙?

(12) Who did Laban believe Jacob loved ⌙?

We do not account for such phenomena here.

Long distance dependencies

Phrases such as whom Jacob loved ⌙ or who ⌙ loved Rachel are instances of a category that we haven't discussed yet.

They are basically *sentences*, with a constituent which is "moved" from its default position and realized as a wh-pronoun in front of the phrase.

We will represent such phrases by using the same category, *s*, which we used for sentences; but to distinguish them from declarative sentences we will add a feature, QUE, to the category. The value of QUE will be '+' in sentences with an interrogative pronoun realizing a transposed constituent.

Long distance dependencies

We add a lexical entry for the pronoun whom:

$$\text{whom} \rightarrow \begin{bmatrix} \text{CAT}: & pron \\ \text{CASE}: & acc \\ \text{QUE}: & + \end{bmatrix}$$

and update the rule that derives pronouns:

$$\begin{bmatrix} \text{CAT}: & np \\ \text{NUM}: & X \\ \text{CASE}: & Y \\ \text{QUE}: & Q \end{bmatrix} \rightarrow \begin{bmatrix} \text{CAT}: & pron \\ \text{NUM}: & X \\ \text{CASE}: & Y \\ \text{QUE}: & Q \end{bmatrix}$$

Long distance dependencies

We now propose an extension of G_3 that can handle long distance dependencies.

The idea is to allow partial phrases, such as Jacob loved ␣, to be derived from a category that is similar to the category of the full phrase, in this case Jacob loved Rachel; but to signal in some way that a constituent, in this case a noun phrase, is missing.

We extend G_3 with two additional rules, based on the first two rules of G_3.

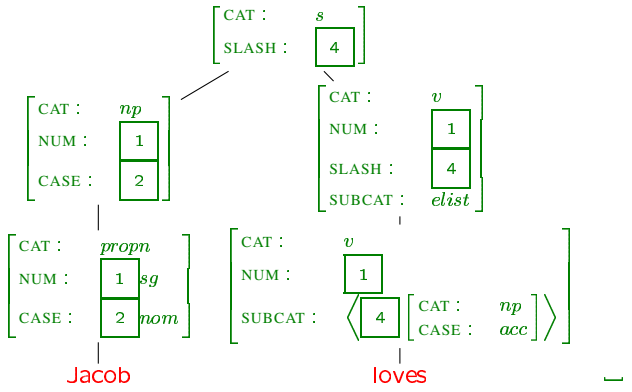
Long distance dependencies

(3) $$\begin{bmatrix} \text{CAT}: & s \\ \text{SLASH}: & Z \end{bmatrix} \rightarrow \begin{bmatrix} \text{CAT}: & np \\ \text{NUM}: & X \\ \text{CASE}: & nom \end{bmatrix} \begin{bmatrix} \text{CAT}: & v \\ \text{NUM}: & X \\ \text{SUBCAT}: & elist \\ \text{SLASH}: & Z \end{bmatrix}$$

(4) $$\begin{bmatrix} \text{CAT}: & v \\ \text{NUM}: & X \\ \text{SUBCAT}: & Y \\ \text{SLASH}: & Z \end{bmatrix} \rightarrow \begin{bmatrix} \text{CAT}: & v \\ \text{NUM}: & X \\ \text{SUBCAT}: & \begin{bmatrix} \text{FIRST}: & Z \\ \text{REST}: & Y \end{bmatrix} \end{bmatrix}$$

Long distance dependencies

Example: A derivation tree for Jacob loves ␣



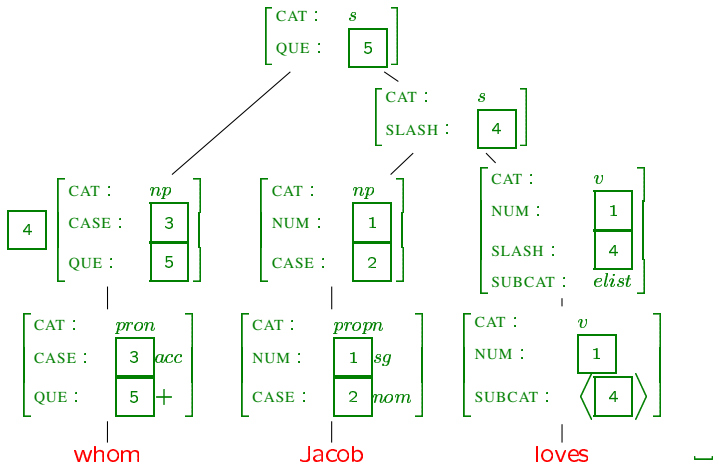
Long distance dependencies

Now that partial phrases can be derived, with a record of their "missing" constituent, all that is needed is a rule for creating "complete" sentences by combining the missing category with a "slashed" sentence:

$$(5)\quad \begin{bmatrix}\text{CAT}: & s\\ \text{QUE}: & Q\end{bmatrix} \rightarrow Z(\begin{bmatrix}\text{QUE}: & Q(+)\end{bmatrix}) \begin{bmatrix}\text{CAT}: & s\\ \text{SLASH}: & Z\end{bmatrix}$$

Long distance dependencies

Example: A derivation tree for whom Jacob loves ␣



Long distance dependencies

Unbounded dependencies can hold across several clause boundaries:

- The shepherd wondered whom Jacob loved ␣.
- The shepherd wondered whom Laban thought Jacob loved ␣.
- The shepherd wondered whom Laban thought Leah claimed Jacob loved ␣.

Also, the dislocated constituent does not have to be an object:

- The shepherd wondered who ␣ loved Rachel.
- The shepherd wondered who Laban thought ␣ loved Rachel.
- The shepherd wondered who Laban thought Leah claimed ␣ loved Rachel.

Long distance dependencies

The solution we proposed for the simple case of unbounded dependencies can be easily extended to the more complex examples:

- a slash introduction rule;
- slash propagation rules;
- and a gap filler rule.

In order to account for filler-gap relations that hold across several clauses, all that needs to be done is to add more slash propagation rules.

Long distance dependencies

For example, in

The shepherd wondered whom Laban thought Jacob loved ␣.

the slash is introduced by the verb phrase loved ␣, and is propagated to the sentence Jacob loved ␣ by rule (3).

A rule that propagates the value of SLASH from a sentential object to the verb phrase of which it is an object:

(6) $$\begin{bmatrix}\text{CAT}: & v\\ \text{NUM}: & X\\ \text{SUBCAT}: & Y\\ \text{SLASH}: & Z\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & v\\ \text{NUM}: & X\\ \text{SUBCAT}: & \begin{bmatrix}\text{FIRST}: & W\\ \text{REST}: & Y\end{bmatrix}\end{bmatrix} W(\begin{bmatrix}\text{SLASH}: & Z\end{bmatrix})$$

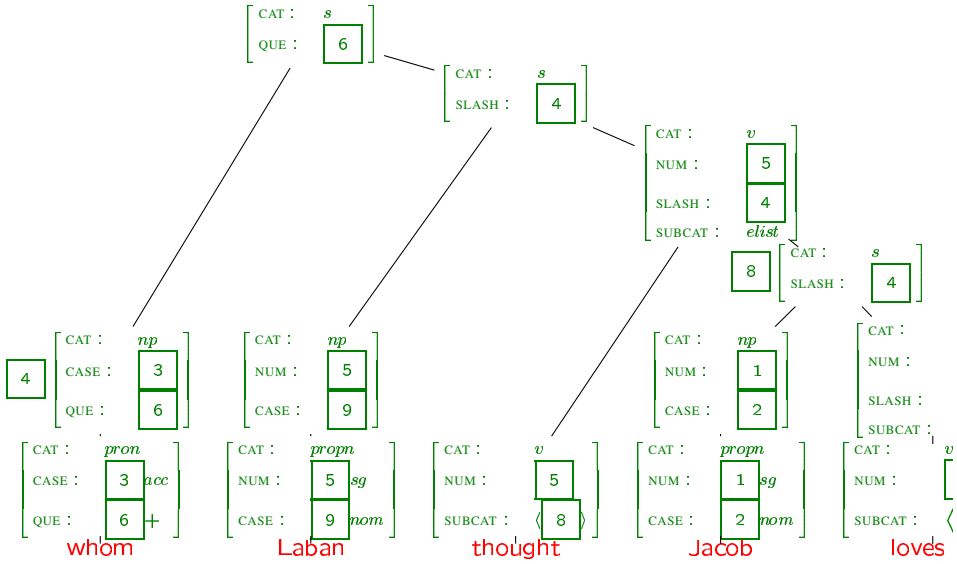
Long distance dependencies

Then, the slash is propagated from the verb phrase thought Jacob loved ␣ to the sentence Laban thought Jacob loved ␣:

(7) $$\begin{bmatrix}\text{CAT}: & s\\ \text{SLASH}: & Z\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & np\\ \text{NUM}: & X\\ \text{CASE}: & nom\end{bmatrix} \begin{bmatrix}\text{CAT}: & v\\ \text{NUM}: & X\\ \text{SUBCAT}: & elist\\ \text{SLASH}: & Z\end{bmatrix}$$

Long distance dependencies

Example: A derivation tree for whom Laban thought Jacob loves ␣



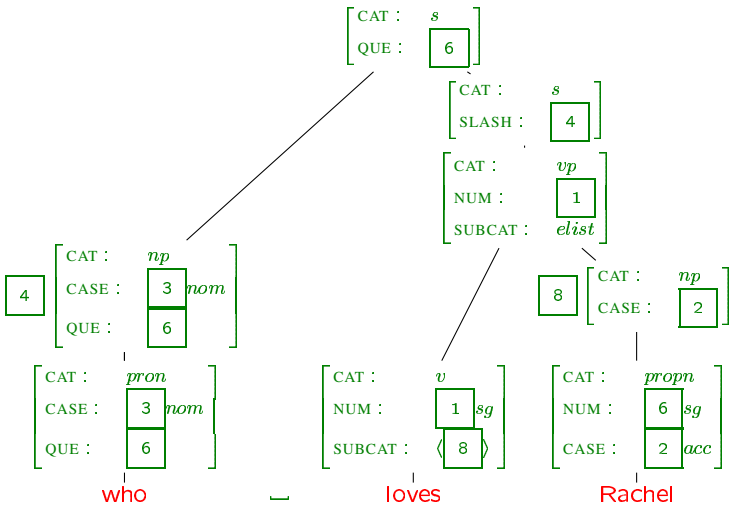
Long distance dependencies

Finally, to account for gaps in the subject position, all that is needed is an additional slash introduction rule:

$$(8)\quad \begin{bmatrix} \text{CAT}: & s \\ \text{SLASH}: & \begin{bmatrix} \text{CAT}: & np \\ \text{NUM}: & X \\ \text{CASE}: & nom \end{bmatrix} \end{bmatrix} \rightarrow \begin{bmatrix} \text{CAT}: & v \\ \text{NUM}: & X \\ \text{SUBCAT}: & elist \end{bmatrix}$$

Long distance dependencies

Example: A derivation tree for who ␣ loves Rachel



Subject and object control

Subject and object control phenomena capture the differences between the 'understood' subjects of the infinitive verb phrase to work seven years in the following sentences:

Jacob promised Laban to work seven years

Laban persuaded Jacob to work seven years

Subject and object control

The key observation in the solution is that the differences between the two examples stem from differences in the matrix verbs:

- promise is a *subject control* verb
- persuade is *object control*.

Subject and object control

Our departure point is the grammar G_3. We modify it by adding a SUBJ feature to verb phrases, whose value is a feature structure associated with the phrase that serves as the verb's subject.

Subject and object control

Example: G_4: explicit SUBJ values

$$\begin{bmatrix}\text{CAT}: & s\end{bmatrix} \rightarrow \boxed{1}\begin{bmatrix}\text{CAT}: & np\\ \text{CASE}: & nom\\ \text{NUM}: & \boxed{7}\end{bmatrix}\begin{bmatrix}\text{CAT}: & v\\ \text{NUM}: & \boxed{7}\\ \text{SUBCAT}: & elist\\ \text{SUBJ}: & \boxed{1}\end{bmatrix}$$

$$\begin{bmatrix}\text{CAT}: & v\\ \text{NUM}: & \boxed{7}\\ \text{SUBCAT}: & \boxed{4}\\ \text{SUBJ}: & \boxed{1}\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & v\\ \text{NUM}: & \boxed{7}\\ \text{SUBCAT}: & \begin{bmatrix}\text{FIRST}: & \boxed{2}\\ \text{REST}: & \boxed{4}\end{bmatrix}\\ \text{SUBJ}: & \boxed{1}\end{bmatrix}\boxed{2}[\,]$$

Example: (continued)

$$\begin{bmatrix}\text{CAT}: & np\\ \text{NUM}: & \boxed{7}\\ \text{CASE}: & \boxed{6}\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & d\\ \text{NUM}: & \boxed{7}\end{bmatrix}\begin{bmatrix}\text{CAT}: & n\\ \text{NUM}: & \boxed{7}\\ \text{CASE}: & \boxed{6}\end{bmatrix}$$

$$\begin{bmatrix}\text{CAT}: & np\\ \text{NUM}: & \boxed{7}\\ \text{CASE}: & \boxed{6}\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & pron\\ \text{NUM}: & \boxed{7}\\ \text{CASE}: & \boxed{6}\end{bmatrix} \mid \begin{bmatrix}\text{CAT}: & propn\\ \text{NUM}: & \boxed{7}\\ \text{CASE}: & \boxed{6}\end{bmatrix}$$

Example: (continued)

sleep →
$$\begin{bmatrix}\text{CAT}: & v\\ \text{SUBCAT}: & elist\\ \text{SUBJ}: & \begin{bmatrix}\text{CAT}: & np\\ \text{CASE}: & nom\end{bmatrix}\\ \text{NUM}: & pl\end{bmatrix}$$

love →
$$\begin{bmatrix}\text{CAT}: & v\\ \text{SUBCAT}: & \langle\begin{bmatrix}\text{CAT}: & np\\ \text{CASE}: & acc\end{bmatrix}\rangle\\ \text{SUBJ}: & \begin{bmatrix}\text{CAT}: & np\\ \text{CASE}: & nom\end{bmatrix}\\ \text{NUM}: & pl\end{bmatrix}$$

give →
$$\begin{bmatrix}\text{CAT}: & v\\ \text{SUBCAT}: & \langle\begin{bmatrix}\text{CAT}: & np\\ \text{CASE}: & acc\end{bmatrix},[\,\text{CAT}: \quad np\,]\rangle\\ \text{SUBJ}: & \begin{bmatrix}\text{CAT}: & np\\ \text{CASE}: & nom\end{bmatrix}\\ \text{NUM}: & pl\end{bmatrix}$$

Example: (continued)

lamb → $\begin{bmatrix}\text{CAT}: & n\\ \text{NUM}: & sg\\ \text{CASE}: & \boxed{6}\end{bmatrix}$ lambs → $\begin{bmatrix}\text{CAT}: & n\\ \text{NUM}: & pl\\ \text{CASE}: & \boxed{6}\end{bmatrix}$

she → $\begin{bmatrix}\text{CAT}: & pron\\ \text{NUM}: & sg\\ \text{CASE}: & nom\end{bmatrix}$ her → $\begin{bmatrix}\text{CAT}: & pron\\ \text{NUM}: & pl\\ \text{CASE}: & acc\end{bmatrix}$

Rachel → $\begin{bmatrix}\text{CAT}: & propn\\ \text{NUM}: & sg\end{bmatrix}$ Jacob → $\begin{bmatrix}\text{CAT}: & propn\\ \text{NUM}: & sg\end{bmatrix}$

a → $\begin{bmatrix}\text{CAT}: & d\\ \text{NUM}: & sg\end{bmatrix}$ two → $\begin{bmatrix}\text{CAT}: & d\\ \text{NUM}: & pl\end{bmatrix}$

Subject and object control

Accounting for infinitival verb phrases:

$$\text{to work} \rightarrow \begin{bmatrix} \text{CAT}: & v \\ \text{VFORM}: & inf \\ \text{SUBCAT}: & elist \\ \text{SUBJ}: & \begin{bmatrix} \text{CAT}: & np \end{bmatrix} \end{bmatrix}$$

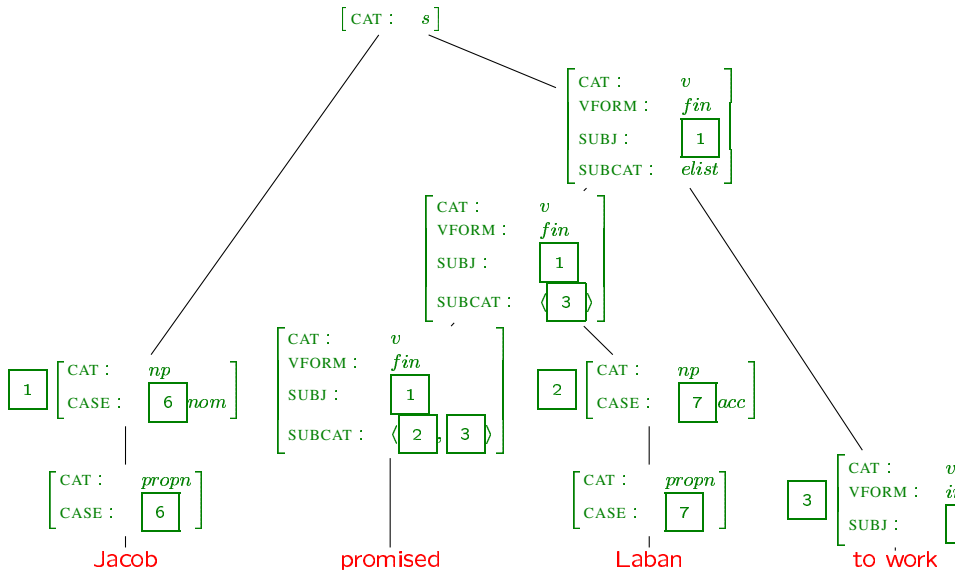
Subject and object control

The lexical entries of verbs such as promise or persuade:

$$\text{promised} \rightarrow \begin{bmatrix} \text{CAT}: & v \\ \text{VFORM}: & fin \\ \text{SUBCAT}: & \left\langle \begin{bmatrix} \text{CAT}: & np \\ \text{CASE}: & acc \end{bmatrix}, \begin{bmatrix} \text{CAT}: & v \\ \text{VFORM}: & inf \\ \text{SUBJ}: & \boxed{1} \end{bmatrix} \right\rangle \\ \text{SUBJ}: & \boxed{1} \begin{bmatrix} \text{CAT}: & np \\ \text{CASE}: & nom \end{bmatrix} \\ \text{NUM}: & [\,] \end{bmatrix}$$

Subject and object control

Example: A derivation tree for Jacob promised Laban to work



Subject and object control

The only difference between the lexical entries of promised and persuaded is that in the latter, the value of the SUBJ list of the infinitival verb phrase is reentrant with the first element on the SUBCAT list of the matrix verb, rather than with its SUBJ value:

$$\text{persuaded} \rightarrow \begin{bmatrix} \text{CAT} : & v \\ \text{VFORM} : & fin \\ \text{SUBCAT} : & \langle \boxed{1} \begin{bmatrix} \text{CAT} : & np \\ \text{CASE} : & acc \end{bmatrix}, \begin{bmatrix} \text{CAT} : & v \\ \text{VFORM} : & inf \\ \text{SUBJ} : & \boxed{1} \end{bmatrix} \rangle \\ \text{SUBJ} : & \begin{bmatrix} \text{CAT} : & np \\ \text{CASE} : & nom \end{bmatrix} \\ \text{NUM} : & [\,] \end{bmatrix}$$

Constituent coordination

Many languages exhibit a phenomenon by which constituents of the same category can be conjoined to form a constituent of this category.

Constituent coordination

N: no man lift up his [hand] or [foot] in all the land of Egypt

NP: Jacob saw [Rachel] and [the sheep of Laban]

VP: Jacob [went on his journey] and [came to the land of the people of the east]

VP: Jacob [went near], and [rolled the stone from the well's mouth], and [watered

ADJ: every [speckled] and [spotted] sheep

ADJP: Leah was [tender eyed] but [not beautiful]

S: [Leah had four sons], but [Rachel was barren]

S: she said to Jacob, "[Give me children], or [I shall die]!"

Constituent coordination

We extend the grammar fragment to cover coordination, referring to it as E_4.

The lexicon of a grammar for E_4 is extended by a closed class of conjunction words; categorized under Conj, this class includes the words and, or, but and perhaps a few others (E_4 contains only these three).

We assume that in E_4, every category of E_0 can be conjoined.

We also assume – simplifying a little – that the same conjunctions are possible for all the categories.

Constituent coordination

A context-free grammar for coordination:

$$\begin{array}{lcl} S & \rightarrow & S\ Conj\ S \\ NP & \rightarrow & NP\ Conj\ NP \\ VP & \rightarrow & VP\ Conj\ VP \\ \vdots & & \\ Conj & \rightarrow & \text{and, or, but, } \ldots \end{array}$$

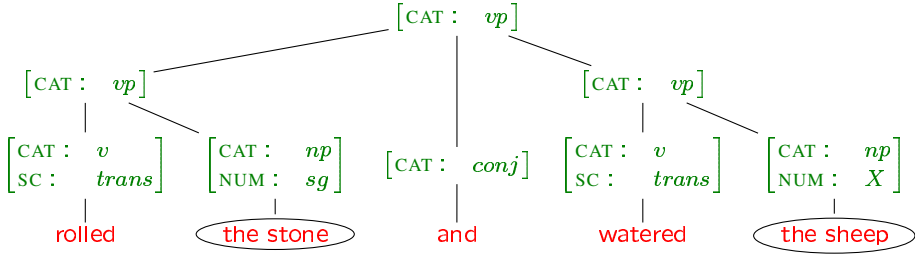
Constituent coordination

With generalized categories, a single production is sufficient:

$$\left[\text{CAT}: \quad X\right] \rightarrow \left[\text{CAT}: \quad X\right] \left[\text{CAT}: \quad conj\right] \left[\text{CAT}: \quad X\right]$$

Constituent coordination

Example: Coordination



Constituent coordination

The above solution is over-simplifying:

- it allows coordination of E_0 categories, but also of E_4 categories;
- it does not handle the properties of coordinated phrases properly;
- it does not permit conjunction of unlikes and of non-constituents.

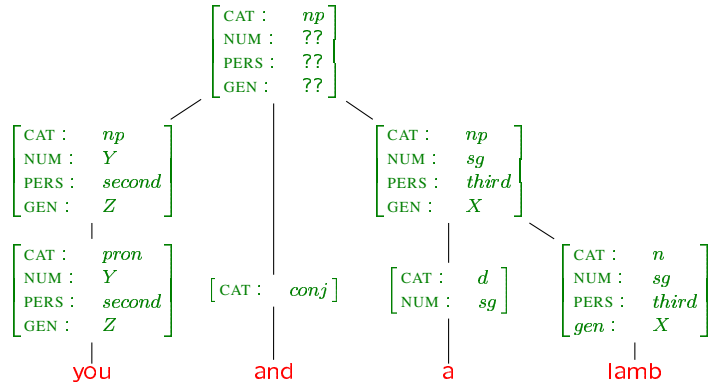
Constituent coordination

Not every category can be coordinated: for example, in English conjunctions cannot themselves be conjoined (in most cases):

$$\begin{bmatrix}\text{CAT}: & X\\ \text{CONJOINABLE}: & -\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & X\\ \text{CONJOINABLE}: & +\end{bmatrix} \begin{bmatrix}\text{CAT}: & conj\end{bmatrix} \begin{bmatrix}\text{CAT}: & X\\ \text{CONJOINABLE}: & +\end{bmatrix}$$

Properties of conjoined constituents

Example: NP coordination



Coordination of unlikes

Consider the following English data:

Joseph became wealthy

Joseph became a minister

Joseph became [wealthy and a minister]

Joseph grew wealthy

*Joseph grew a minister

*Joseph grew [wealthy and a minister]

Coordination of unlikes

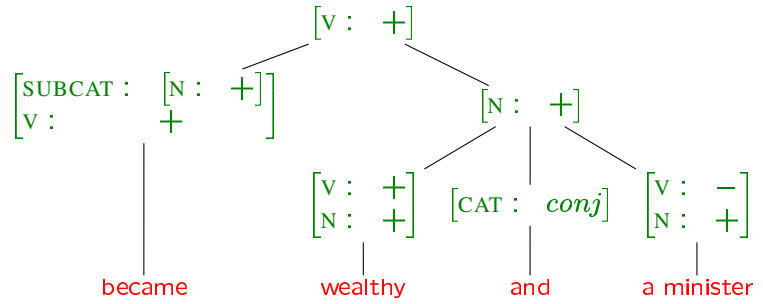
These data are easy to account for in a unification-based framework with a possibility of specifying generalization instead of unification in certain cases:

$$\boxed{1} \sqcap \boxed{2} \rightarrow \boxed{1}\begin{bmatrix}\text{CAT}: & X\end{bmatrix}\begin{bmatrix}\text{CAT}: & conj\end{bmatrix}\boxed{2}\begin{bmatrix}\text{CAT}: & Y\end{bmatrix}$$

where '$\sqcap$' is the generalization operator.

Coordination of unlikes

Example:



Coordination of unlikes

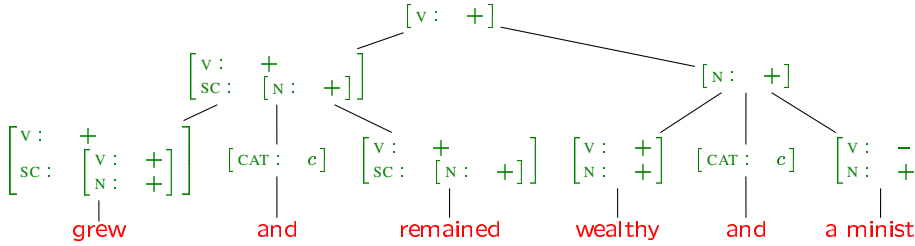
The situation becomes more complicated when verbs, too, are conjoined:

*Joseph [grew and remained] [wealthy and a minister]

While this example is ungrammatical, it is obtainable by the same grammar

Coordination of unlikes

Example:



Non-constituent coordination

Rachel gave the sheep [grass] and [water]

Rachel gave [the sheep grass] and [the lambs water]

Rachel [kissed] and Jacob [hugged] Binyamin

Expressiveness of unification grammars

Just how expressive are unification grammars?

What is the class of languages generated by unification grammars?

Trans-context-free languages

A grammar, G_{abc}, for the language $L = \{a^n b^n c^n \mid n > 0\}$.

Feature structures will have two features: CAT, which stands for category, and T, which "counts" the length of sequences of a-s, b-s and c-s.

The "category" is ap for strings of a-s, bp for b-s and cp for c-s. The categories *at*, *bt* and *ct* are pre-terminal categories of the words a, b and c, respectively.

"Counting" is done in unary base: a string of length n is derived by an AVM (that is, an multi-AVM of length 1) whose depth is n.

For example, the string bbb is derived by the following AVM:

$$\begin{bmatrix} \text{CAT}: & bp \\ \text{T}: & \begin{bmatrix} \text{T}: & \begin{bmatrix} \text{T}: & end \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Trans-context-free languages

Example: A unification grammar G_{abc} for the language $\{a^n b^n c^n \mid n > 0\}$

The signature of the grammar consists in the features CAT and T and the atoms *s*, *ap*, *bp*, *cp*, *at*, *bt*, *ct* and *end*. The terminal symbols are, of course, *a*, *b* and *c*. The start symbol is the left-hand side of the first rule.

$$\rho_1: \quad \begin{bmatrix} \text{CAT}: & s \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} \text{CAT}: & ap \\ \text{T}: & X \end{bmatrix} \quad \begin{bmatrix} \text{CAT}: & bp \\ \text{T}: & X \end{bmatrix} \quad \begin{bmatrix} \text{CAT}: & cp \\ \text{T}: & X \end{bmatrix}$$

Example: (continued)

$$\rho_2: \begin{bmatrix}\text{CAT}: & ap \\ \text{T}: & \begin{bmatrix}\text{T}: & X\end{bmatrix}\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & at\end{bmatrix} \begin{bmatrix}\text{CAT}: & ap \\ \text{T}: & X\end{bmatrix}$$

$$\rho_3: \begin{bmatrix}\text{CAT}: & ap \\ \text{T}: & end\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & at\end{bmatrix}$$

$$\rho_4: \begin{bmatrix}\text{CAT}: & bp \\ \text{T}: & \begin{bmatrix}\text{T}: & X\end{bmatrix}\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & bt\end{bmatrix} \begin{bmatrix}\text{CAT}: & bp \\ \text{T}: & X\end{bmatrix}$$

$$\rho_5: \begin{bmatrix}\text{CAT}: & bp \\ \text{T}: & end\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & bt\end{bmatrix}$$

$$\rho_6: \begin{bmatrix}\text{CAT}: & cp \\ \text{T}: & \begin{bmatrix}\text{T}: & X\end{bmatrix}\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & ct\end{bmatrix} \begin{bmatrix}\text{CAT}: & cp \\ \text{T}: & X\end{bmatrix}$$

$$\rho_7: \begin{bmatrix}\text{CAT}: & cp \\ \text{T}: & end\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & ct\end{bmatrix}$$

Example: (continued)

$$\begin{bmatrix}\text{CAT}: & at\end{bmatrix} \rightarrow a$$

$$\begin{bmatrix}\text{CAT}: & bt\end{bmatrix} \rightarrow b$$

$$\begin{bmatrix}\text{CAT}: & ct\end{bmatrix} \rightarrow c$$

Trans-context-free languages

Example: Derivation sequence of $a^2b^2c^2$
Start with a form that consists of the start symbol,

$$\sigma_0 = \begin{bmatrix}\text{CAT}: & s\end{bmatrix}.$$

Only one rule, ρ_1, can be applied to the single element of the multi-AVM in σ_0, yielding:

$$\sigma_1 = \begin{bmatrix}\text{CAT}: & ap \\ \text{T}: & X\end{bmatrix} \begin{bmatrix}\text{CAT}: & bp \\ \text{T}: & X\end{bmatrix} \begin{bmatrix}\text{CAT}: & cp \\ \text{T}: & X\end{bmatrix}$$

Example: (continued)
Applying ρ_2, to the first element of σ_1:

$$\sigma_2 = \begin{bmatrix}\text{CAT}: & at\end{bmatrix} \begin{bmatrix}\text{CAT}: & ap \\ \text{T}: & X\end{bmatrix} \begin{bmatrix}\text{CAT}: & bp \\ \text{T}: & \begin{bmatrix}\text{T}: & X\end{bmatrix}\end{bmatrix} \begin{bmatrix}\text{CAT}: & cp \\ \text{T}: & \begin{bmatrix}\text{T}: & X\end{bmatrix}\end{bmatrix}$$

We can now choose the third element in σ_2 and apply the rule ρ_4:

$$\sigma_3 = \begin{bmatrix}\text{CAT}: & at\end{bmatrix} \begin{bmatrix}\text{CAT}: & ap \\ \text{T}: & X\end{bmatrix} \begin{bmatrix}\text{CAT}: & bt\end{bmatrix} \begin{bmatrix}\text{CAT}: & bp \\ \text{T}: & X\end{bmatrix} \begin{bmatrix}\text{CAT}: & cp \\ \text{T}: & \begin{bmatrix}\text{T}: & X\end{bmatrix}\end{bmatrix}$$

Applying ρ_6 to the fifth element of $\sigma 3$, we get:

$$\sigma_4 = \begin{bmatrix}\text{CAT}: & at\end{bmatrix} \begin{bmatrix}\text{CAT}: & ap \\ \text{T}: & X\end{bmatrix} \begin{bmatrix}\text{CAT}: & bt\end{bmatrix} \begin{bmatrix}\text{CAT}: & bp \\ \text{T}: & X\end{bmatrix} \begin{bmatrix}\text{CAT}: & ct\end{bmatrix} \begin{bmatrix}\text{CAT}: & cp \\ \text{T}: & X\end{bmatrix}$$

Example: (continued)
The second element of σ_4 is unifiable with the heads of both ρ_2 and ρ_3. We choose to apply ρ_3:

$$\sigma_5 = \begin{bmatrix}\text{CAT}: & at\end{bmatrix}\ \begin{bmatrix}\text{CAT}: & at\end{bmatrix}\ \begin{bmatrix}\text{CAT}: & bt\end{bmatrix}\ \begin{bmatrix}\text{CAT}: & bp\\ \text{T}: & end\end{bmatrix}\ \begin{bmatrix}\text{CAT}: & ct\end{bmatrix}\ \begin{bmatrix}\text{CAT}: & cp\\ \text{T}: & end\end{bmatrix}$$

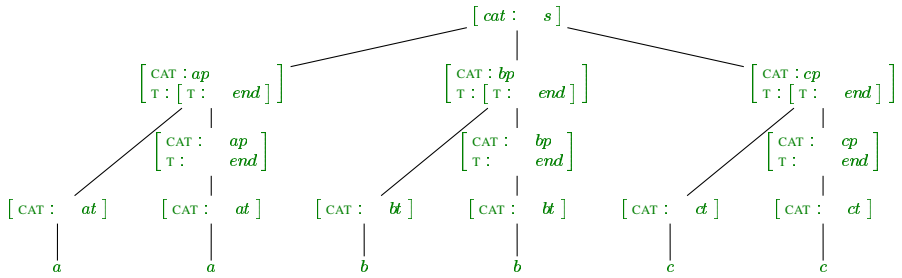
In the same way we can now apply ρ_5 and ρ_7 and obtain, eventually,

$$\sigma_7 = \begin{bmatrix}\text{CAT}: & at\end{bmatrix}\ \begin{bmatrix}\text{CAT}: & at\end{bmatrix}\ \begin{bmatrix}\text{CAT}: & bt\end{bmatrix}\ \begin{bmatrix}\text{CAT}: & bt\end{bmatrix}\ \begin{bmatrix}\text{CAT}: & ct\end{bmatrix}\ \begin{bmatrix}\text{CAT}: & ct\end{bmatrix}$$

Now, let $w = aabbcc$; then σ_7 is a member of $PT_w(1,6)$; in fact, it is the only member of the preterminal set. Therefore, $w \in L(G_{abc})$.

Trans-context-free languages

Example: Derivation tree of $a^2b^2c^2$



The repetition language

Example: A unification grammar for the language $\{ww \mid w \in \{a,b\}^+\}$
The signature of the grammar consists in the features CAT, FIRST and REST and the atoms *s*, *ap*, *bp*, *at*, *bt* and *elist*. The terminal symbols are *a* and *b*. The start symbol is the left-hand side of the first rule.

Example: (continued)

$$\begin{bmatrix}\text{CAT}: & s\end{bmatrix} \rightarrow \begin{bmatrix}\text{FIRST}: & X\\ \text{REST}: & Y\end{bmatrix}\ \begin{bmatrix}\text{FIRST}: & X\\ \text{REST}: & Y\end{bmatrix}$$

$$\begin{bmatrix}\text{FIRST}: & ap\\ \text{REST}: & \begin{bmatrix}\text{FIRST}: & X\\ \text{REST}: & Y\end{bmatrix}\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & at\end{bmatrix}\ \begin{bmatrix}\text{FIRST}: & X\\ \text{REST}: & Y\end{bmatrix}$$

$$\begin{bmatrix}\text{FIRST}: & bp\\ \text{REST}: & \begin{bmatrix}\text{FIRST}: & X\\ \text{REST}: & Y\end{bmatrix}\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & bt\end{bmatrix}\ \begin{bmatrix}\text{FIRST}: & X\\ \text{REST}: & Y\end{bmatrix}$$

$$\begin{bmatrix}\text{FIRST}: & ap\\ \text{REST}: & elist\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & at\end{bmatrix}$$

$$\begin{bmatrix}\text{FIRST}: & bp\\ \text{REST}: & elist\end{bmatrix} \rightarrow \begin{bmatrix}\text{CAT}: & bt\end{bmatrix}$$

$$\begin{bmatrix}\text{CAT}: & at\end{bmatrix} \rightarrow a$$

$$\begin{bmatrix}\text{CAT}: & bt\end{bmatrix} \rightarrow b$$

Unification grammars and Turing machines

Unification grammars can simulate the operation of Turing machines.

The membership problem for unification grammars is as hard as the halting problem.

Unification grammars and Turing machines

A (deterministic) **Turing machine** $(Q, \Sigma, \flat, \delta, s, h)$ is a tuple such that:

- Q is a finite set of states
- Σ is an alphabet, not containing the symbols L, R and $elist$
- $\flat \in \Sigma$ is the blank symbol
- $s \in Q$ is the initial state
- $h \in Q$ is the final state
- $\delta : (Q \setminus \{h\}) \times \Sigma \to Q \times (\Sigma \cup \{L, R\})$ is a total function specifying transitions.

Unification grammars and Turing machines

A **configuration** of a Turing machine consists of the state, the contents of the tape and the position of the head on the tape.

A configuration is depicted as a quadruple (q, w_l, σ, w_r) where $q \in Q$, $w_l, w_r \in \Sigma^*$ and $\sigma \in \Sigma$; in this case, the contents of the tape is $\flat^\omega \cdot w_l \cdot \sigma \cdot w_r \cdot \flat^\omega$, and the head is positioned on the σ symbol.

A given configuration yields a next configuration, determined by the transition function δ, the current state and the character on the tape that the head points to.

Unification grammars and Turing machines

Let

$$\text{first}(\sigma_1 \cdots \sigma_n) = \begin{cases} \sigma_1 & n > 0 \\ \flat & n = 0 \end{cases} \qquad \text{but-first}(\sigma_1 \cdots \sigma_n) = \begin{cases} \sigma_2 \cdots \sigma_n & n > 1 \\ \epsilon & n \leq 1 \end{cases}$$

$$\text{last}(\sigma_1 \cdots \sigma_n) = \begin{cases} \sigma_n & n > 0 \\ \flat & n = 0 \end{cases} \qquad \text{but-last}(\sigma_1 \cdots \sigma_n) = \begin{cases} \sigma_1 \cdots \sigma_{n-1} & n > 1 \\ \epsilon & n \leq 1 \end{cases}$$

Unification grammars and Turing machines

Then the next configuration of a configuration (q, w_l, σ, w_r) is defined iff $q \neq h$, in which case it is:

(p, w_l, σ', w_r)	if $\delta(q, \sigma) = (p, \sigma')$ where $\sigma' \in \Sigma$
$(p, w_l\sigma, \text{first}(w_r), \text{but-first}(w_r))$	if $\delta(q, \sigma) = (p, R)$
$(p, \text{but-last}(w_l), \text{last}(w_l), \sigma w_r)$	if $\delta(q, \sigma) = (p, L)$

Unification grammars and Turing machines

A next configuration is only defined for configurations in which the state is not the final state, h.

Since δ is a total function, there always exists a unique next configuration for every given configuration.

We say that a configuration c_1 yields the configuration c_2, denoted $c_1 \vdash c_2$, iff c_2 is the next configuration of c_1.

Unification grammars and Turing machines

Program:

- define a unification grammar G_M for every Turing machine M such that the grammar generates the word halt if and only if the machine accepts the empty input string:

$$L(G_M) = \begin{cases} \{\text{halt}\} & \text{if } M \text{ terminates for the empty input} \\ \emptyset & \text{if } M \text{ does not terminate on the empty input} \end{cases}$$

- if there were a decision procedure to determine whether $w \in L(G)$ for an arbitrary unification grammar G, then in particular such a procedure could determine membership in the language of G_M, simulating the Turing machine M.

- the procedure for deciding whether $w \in L(G)$, when applied to

the problem halt$\in L(G_M)$, determines whether M terminates for the empty input, which is known to be undecidable.

Unification grammars and Turing machines

Feature structures will have three features: curr, representing the character under the head; right, representing the tape contents to the right of the head (as a list); and left, representing the tape contents to the left of the head, in a reversed order.

All the rules in the grammar are unit rules; and the only terminal symbol is halt. Therefore, the language generated by the grammar is necessarily either the singleton {halt} or the empty set.

Unification grammars and Turing machines: signature

Let $M = (Q, \Sigma, \flat, \delta, s, h)$ be a Turing machine. Define a unification grammar G_M as follows:

- $\textsc{Feats} = \{\textsc{cat}, \textsc{left}, \textsc{right}, \textsc{curr}, \textsc{first}, \textsc{rest}\}$
- $\textsc{Atoms} = \Sigma \cup \{\mathit{start}, \mathit{elist}\}$.
- The start symbol is $\begin{bmatrix}\textsc{cat}: & \mathit{start}\end{bmatrix}$.
- the only terminal symbol is halt.

Unification grammars and Turing machines: rules

Two rules are defined for every Turing machine:

$$\begin{bmatrix}\textsc{cat}: & \mathit{start}\end{bmatrix} \rightarrow \begin{bmatrix}\textsc{cat}: & s\\ \textsc{curr}: & \flat\\ \textsc{right}: & \mathit{elist}\\ \textsc{left}: & \mathit{elist}\end{bmatrix}$$

$$h \rightarrow \text{halt}$$

Unification grammars and Turing machines: rules

For every q, σ such that $\delta(q, \sigma) = (p, \sigma')$ and $\sigma' \in \Sigma$, the following rule is defined:

$$\begin{bmatrix}\textsc{cat}: & q\\ \textsc{curr}: & \sigma\\ \textsc{right}: & X\\ \textsc{left}: & Y\end{bmatrix} \rightarrow \begin{bmatrix}\textsc{cat}: & p\\ \textsc{curr}: & \sigma'\\ \textsc{right}: & X\\ \textsc{left}: & Y\end{bmatrix}$$

Unification grammars and Turing machines: rules

For every q, σ such that $\delta(q, \sigma) = (p, R)$ we define two rules:

$$\begin{bmatrix} \text{CAT}: & q \\ \text{CURR}: & \sigma \\ \text{RIGHT}: & elist \\ \text{LEFT}: & X \end{bmatrix} \rightarrow \begin{bmatrix} \text{CAT}: & p \\ \text{CURR}: & \flat \\ \text{RIGHT}: & elist \\ \text{LEFT}: & \begin{bmatrix} \text{FIRST}: & \sigma \\ \text{REST}: & X \end{bmatrix} \end{bmatrix}$$

$$\begin{bmatrix} \text{CAT}: & q \\ \text{CURR}: & \sigma \\ \text{RIGHT}: & \begin{bmatrix} \text{FIRST}: & X \\ \text{REST}: & Y \end{bmatrix} \\ \text{LEFT}: & W \end{bmatrix} \rightarrow \begin{bmatrix} \text{CAT}: & p \\ \text{CURR}: & X \\ \text{RIGHT}: & Y \\ \text{LEFT}: & \begin{bmatrix} \text{FIRST}: & \sigma \\ \text{REST}: & W \end{bmatrix} \end{bmatrix}$$

Unification grammars and Turing machines: rules

For every q, σ such that $\delta(q, \sigma) = (p, L)$ we define two rules:

$$\begin{bmatrix} \text{CAT}: & q \\ \text{CURR}: & \sigma \\ \text{RIGHT}: & X \\ \text{LEFT}: & elist \end{bmatrix} \rightarrow \begin{bmatrix} \text{CAT}: & p \\ \text{CURR}: & \flat \\ \text{RIGHT}: & \begin{bmatrix} \text{FIRST}: & \sigma \\ \text{REST}: & X \end{bmatrix} \\ \text{LEFT}: & elist \end{bmatrix}$$

$$\begin{bmatrix} \text{CAT}: & q \\ \text{CURR}: & \sigma \\ \text{RIGHT}: & X \\ \text{LEFT}: & \begin{bmatrix} \text{FIRST}: & Y \\ \text{REST}: & W \end{bmatrix} \end{bmatrix} \rightarrow \begin{bmatrix} \text{CAT}: & p \\ \text{CURR}: & Y \\ \text{RIGHT}: & \begin{bmatrix} \text{FIRST}: & \sigma \\ \text{REST}: & X \end{bmatrix} \\ \text{LEFT}: & W \end{bmatrix}$$

Unification grammars and Turing machines: results

Lemma 1. *Let c_1, c_2 be configurations of a Turing machine M, and A_1, A_2 be AVMs encoding these configurations, viewed as multi-AVMs of length 1. Then $c_1 \vdash c_2$ iff $A_1 \Rightarrow A_2$ in G_m.*

Theorem 2. *A Turing machine M halts for the empty input iff halt $\in L(G_M)$.*

Corollary 3. *The universal recognition problem for unification grammars is undecidable.*