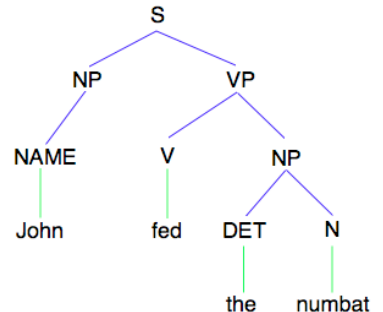



Grammars and Parsing

Aims	<ul style="list-style-type: none"> to describe types of formal grammar to introduce parse trees to study chart parsers to study parsing using Prolog
Reference	Allen, chapter 3.
Keywords	accepter, active arc, active chart, alphabet (in grammar), bottom-up parser, CFG, chart, chart parsing, Chomsky hierarchy, constituent, context-free, context-sensitive, CSG, derivation, distinguished non-terminal, generate, grammar rule, language generated by a grammar, left-to-right parsing, lexical category, lexical insertion rule, lexical symbol, lexicon, non-terminal, parse tree, parser, phrasal category, pre-terminal, production, regular, rewriting process, right-linear grammar, sentential form, start symbol, terminal, unrestricted grammar
Plan	<ul style="list-style-type: none"> parsers, parse trees grammars context-free grammars parsing in Prolog bottom-up chart parsing

Parser

- a *parser* is an algorithm for analysing sentences, given a grammar
- an *accepter* just answers yes or no to the question *Does this sentence conform to the given grammar?*
- a full parser also produces a structure description or **parse tree** for correct sentences:

 <pre> graph TD S --> NP1[NP] S --> VP[VP] NP1 --> NAME[NAME] NAME --> John[John] VP --> V[V] V --> fed[fed] VP --> NP2[NP] NP2 --> DET[DET] DET --> the[the] NP2 --> N[N] N --> numbat[numbat] </pre>	 <p>Numbats are ant-eating marsupials now confined to the south-west corner of Western Australia, though previously they were more widespread. Image from www.wilderness.org.au</p>	<p>In list notation ...</p> <pre> (S (NP (NAME John)) (VP (V fed) (NP (DET the)) (N numbat)))) </pre> <p>and in a Prolog notation ...</p> <pre> s(np(name(john)), vp(v(fed), np(det(the), n(numbat)))) </pre>
--	---	---

Grammar

- a (formal) grammar is a set of rules for describing a language
- a *syntactic* grammar describes the *form* of the language
- a *semantic* grammar builds in *meaning*-based restrictions
- we will be concerned with syntactic grammar

Describing Syntax: Context-Free Grammars (CFGs)

- | | |
|----------------------------|---------|
| 1. $S \rightarrow NP VP$ | |
| 2. $VP \rightarrow V NP$ | Grammar |
| 3. $NP \rightarrow NAME$ | Rules |
| 4. $NP \rightarrow DET N$ | |
| 5. $NAME \rightarrow John$ | |
| 6. $V \rightarrow fed$ | Lexical |
| 7. $DET \rightarrow the$ | Entries |
| 8. $N \rightarrow numbat$ | |

Context-Free Grammar Definition

A context-free grammar (CFG) is a 5-tuple (P, A, N, T, S) where:

- P is a set of **Context-free productions**, i.e. objects of the form $X \rightarrow \beta$, where X is a member of N , and β is a string over the alphabet A
- A is an **Alphabet** of grammar symbols;
- N is a set of **Non-terminal** symbols;
- T is a set of **Terminal** symbols (and $N \cup T = A$);
- S is a distinguished non-terminal called the **Start** symbol (think of it as "sentence" in NLP applications).

CFG Example

$P = \{S \rightarrow NP VP, VP \rightarrow V NP, NP \rightarrow NAME, NP \rightarrow DET N, NAME \rightarrow John, V \rightarrow fed, DET \rightarrow the, N \rightarrow numbat\}$.

$A = N \cup T$

$N = \{S, NP, VP, N, V, NAME, DET\}$

$T = \{fed, John, numbat, the\}$

$S = , well, S$

$N, V, NAME$ and DET are called **pre-terminal** or **lexical** symbols, because they only occur in productions of the form $X \rightarrow \text{some_word}$.

Programming Language Grammars

In programming language grammars, there would be context-free rules like:

WhileStatement \rightarrow **while** Condition { StatementList }
 StatementList \rightarrow Statement ;
 StatementList \rightarrow Statement StatementList

Optional Homework for Digression: Write grammar rules for the Prolog programming language. You will need grammar rules to describe:

- a Prolog rule
- the goal list in the body of the Prolog rule
- Prolog terms
- arithmetic expressions like $N * (N - 1)$ (several rules here: this part is a bit hard)
- relational expressions like $N = M$ (several rules here)
- ...

What would the lexicon be like?

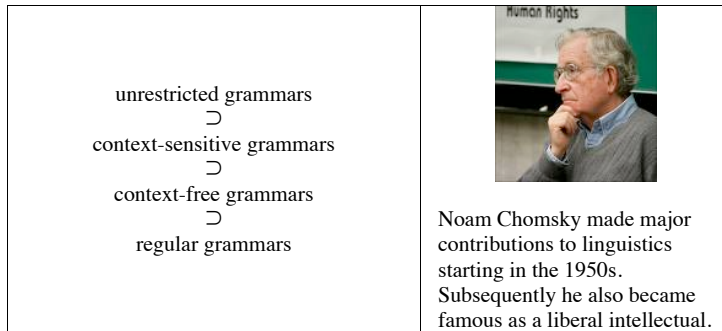
Definition of String

- A string over the alphabet A is a sequence of zero or more symbols from A .
- The string with zero symbols is often written as ϵ (*empty string*).
- The *length* of the string is the number of symbols in it.

Examples of Strings	Length
numbat S the	3
S	1
ϵ	0
NP VP	2
John fed the numbat	4
numbat the fed John	4
numbat the John S S John NP DET DET S S John fed	13

- A *non-empty string* means a string other than ϵ .
- The strings over the alphabet A are sometimes referred to as A^* . (This has nothing to do with the A^* in A^* search.)
- A^+ then refers to the non-empty strings over A .

The Chomsky Hierarchy of Grammars



With context-free grammars, these form the *Chomsky hierarchy* of grammars. The four types of grammar differ in the type of rewriting rule $\alpha \rightarrow \beta$ that is allowed.

Since the restrictions which define the grammar types apply to the *rules*, it makes sense to talk of unrestricted, context-sensitive, context-free, and regular rules.

Unrestricted Grammars

No restrictions on the form that the Rules $\alpha \rightarrow \beta$ can take.
 Not used much: the extreme power makes them difficult to work with.

Context Sensitive Grammars

- Also called a *transformational grammar*.
- Restriction is $\text{length}(\alpha) \leq \text{length}(\beta)$;
- Equivalently, all rules must be of the form $\lambda N \rho \rightarrow \lambda \alpha \rho$, where λ and ρ are any strings (including the empty string ϵ).
- λ and ρ are thought of as the *left and right context* in which the non-terminal symbol N can be rewritten as the non-null symbol-string α . Hence the term context-sensitive grammar.
- Context-sensitive production rules can be used for transforming an active sentence into the corresponding passive sentence.

Context-Free Grammar

- All rules must be of the form $N \rightarrow \alpha$, where N is a nonterminal symbol and α is any string.
- Mostly we will be considering context-free rules in this course.
- Any context-sensitive rule fits the definition of context-free (just set $\lambda = \rho = \epsilon$).
- Thus a rule's type is determined by the *strongest* condition that it meets.

Regular Grammar or Right Linear Grammar

- All rules take one of two forms:
 1. $N \rightarrow t$,
 2. $N \rightarrow tM$, where N and M are non-terminals and t is a terminal (N and M may be the same).
- Regular grammar rules are not powerful enough to conveniently describe natural languages (or even programming languages).
- They can sometimes be used to describe portions of languages, and have the advantage that they lead to fast parsing.

Grammars and Logic Programming

Section 3.8 in Allen

- Grammar rules can be encoded directly in Prolog:

$$S \rightarrow NP VP \quad \Rightarrow \quad s(P1, P3) \text{ :- } np(P1, P2), vp(P2, P3).$$
- That is, there is an S FROM sentence position P1 TO position P3 if
 - there is an NP FROM P1 TO P2 and
 - there is a VP FROM P2 TO P3.
- $VP \rightarrow V NP \quad \Rightarrow \quad vp(P1, P3) \text{ :- } v(P1, P2), np(P2, P3).$
- $NP \rightarrow NAME \quad \Rightarrow \quad np(P1, P2) \text{ :- } propernoun(P1, P2).$
- $NP \rightarrow DET N \quad \Rightarrow \quad np(P1, P3) \text{ :- } det(P1, P2), noun(P2, P3).$

Lexicon in Prolog

- The lexicon can be defined in Prolog like this:

```
NAME → John      isname(john).
V → fed          isverb(fed).
DET → the        isdet(the).
N → numbat      isnoun(numbat).
```

Rules to Build Lexical Constituents

- Predicates are needed, to link the lexicon to the grammar.
- For each lexical category, like DET, you define a predicate, like

```
det(From, To) :-
    word(Word, From, To),
    isdet(Word).
```

This predicate is true only if the word between the specified positions satisfies `isdet()`.

- Here `word(Word, From, To)` signifies that `Word` is there in the input sentence between positions `From` and `To`.

Words, Parsing

- To use the system as so far described, assert the words in their sentence positions:

```
word(john, 1, 2).
word(fed, 2, 3).
word(the, 3, 4).
word(numbat, 4, 5).
```

- This is equivalent to annotating the sentence as: `1 John 2 fed 3 the 4 numbat. 5`
- then use the Prolog query `?- s(1,5)`. This will result in Yes.
- A more elaborate parser would take a list of words as input, and generate word facts like those above.
- More significant is the fact that this is just an *accepter* – we want to compute the *structure* of the sentence.

Parsing for Structure

To get at the structure of the sentence, add extra arguments to pass the structure back across necks as it is built:

```
% if there is a noun phrase with structure NP, and a verb phrase
% with structure, VP, there there is a sentence with structure
% s(SynNP, SynVP):
s(P1, P3, s(SynNP, SynVP)) :-
    np(P1, P2, SynNP), vp(P2, P3, SynVP).

vp(P1, P3, vp(SynVerb, SynNP)) :-
    v(P1, P2, SynVerb), np(P2, P3, SynNP).

np(P1, P2, np(SynName)) :-
    proper(P1, P2, SynName).

np(P1, P3, np(SynDet, SynNoun)) :-
    det(P1, P2, SynDet), noun(P2, P3, SynNoun).
```

This sorts out the phrasal constituents' structures – what about the lexical constituents?

Structure Passing with Lexical Constituents

```
% if Word is a determiner, then there is a lexical constituent
% with structure Word
det(From, To, det(Word)) :-
    word(Word, From, To), isdet(Word).

noun(From, To, noun(Word)) :-
    word(Word, From, To), isnoun(Word).
v(From, To, v(Word)) :-
    word(Word, From, To), isverb(Word).
proper(From, To, name(Word)) :-
    word(Word, From, To), isname(Word).
```

Starting the Parser

- The query to start the parser now becomes

```
?- s(1,5,Parse).
Parse= s(np(name(john)), vp(v(fed), np(det(the), noun(numbat))))
```

- You can get a copy of this code, to experiment with, at <http://www.cse.unsw.edu.au/~billw/cs9414/notes/nlp/grampars/prolog-parser-3.pro>

Optional Homework

- How would you go about writing a parser for, say, a Prolog rule?

Backtracking in a Prolog Parser

- The simple Prolog parser just described only parses a few sentences – a more realistic one would need hundreds of rules.
- The parser operates by a process of *top-down*¹ depth-first search – if it is trying to parse a noun phrase, it picks the first rule for np and tries it. If it doesn't work, it backtracks and tries the next rule for np, and so on.
- This works well if it finds out quickly when it has tried the wrong rule. If this does not occur, considerable amounts of parsing time can be wasted – parsing time can be exponential in the length of the sentence in the worst case.
- The amount of wasted time can be reduced by recording structures built while using wrong rules. This may seem strange, but if a prepositional phrase is parsed while trying a rule that ultimately fails because of something after the prepositional phrase, this phrase will probably be parsed in the same way by the correct rule when it is found. If the PP parse is already known, this bit of work can be skipped, the second time around.

¹ It is called “top-down” because it builds the parse tree from the root down – starting with an S rule, and then say an NP rule, and so on down the hierarchy of phrasal and eventually lexical categories.

A Bottom-Up Chart Parser ...

- parses a sentence of length N within N^3 steps.
- does better than this (N^2 or N steps) with “well-behaved” grammars.
- constructs phrasal or lexical *constituents* of a sentence.
- We use the sentence *the green fly flies* as an example.
- We annotate the sentence with *positions*: ${}_0\text{the}_1\text{green}_2\text{fly}_3\text{flies}_4$.

Chart Parser 2

- The parsing process succeeds if an S (sentence) constituent is found covering positions 0 to 4.
- Operations (2) to (9) below do not completely specify the order in which parsing steps are carried out: one reasonable order is
 - scan a word (as in (3))
 - perform all possible parsing steps as specified in (4) - (7) before scanning another word.
 - Parsing is completed when the last word has been read and all possible subsequent parsing steps have been performed.

Chart Parser 3

- **Parser Inputs:** sentence, lexicon, grammar. In our example, the lexicon is:
the: DET *green*: ADJ *fly*: N, V *flies*: N, V
 and the grammar rules are $S \rightarrow NP VP$, $NP \rightarrow DET ADJ N$, and $VP \rightarrow V$

- **Parser Operations:**
 - 1 The algorithm operates on two data structures: the **active chart** - a collection of **active arcs** (see (4) below) and the constituents (see (3) and (6)). Both are initially empty.
 - 2 The grammar is considered to include **lexical insertion rules**: for example, if *fly* is a word in the lexicon/vocabulary being used, and if its lexical entry includes the fact that *fly* may be a N or a V, then rules of the form $N \rightarrow fly$ and $V \rightarrow fly$ are considered to be part of the grammar.

Chart Parser 4

- 3 **Creating lexical constituents:** As a word (like *fly*) is scanned, constituents corresponding to its lexical categories are created:
 N1: $N \rightarrow fly$ FROM 2 TO 3, and
 V1: $V \rightarrow fly$ FROM 2 TO 3
- 4 **Starting an active arc:** If the grammar contains a rule like $NP \rightarrow DET ADJ N$, and a constituent like $DET1: DET \rightarrow the$ FROM m TO n has been found, then an *active arc*
 ARC1: $NP \rightarrow DET1 \bullet ADJ N$ FROM m TO n

is added to the *active chart*. (In our example sentence, m would be 0 and n would be 1.) The dot "•" in an active arc marks the boundary between found constituents and constituents not (yet) found.

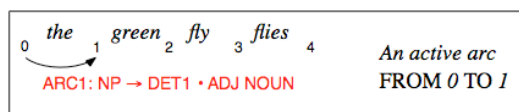


Chart Parser 5

- 5 **Advancing the "•":** If the active chart has an active arc like:
 ARC1: $NP \rightarrow DET1 \bullet ADJ N$ FROM m TO n
 and there is a constituent in the chart of type ADJ (i.e. the first item after the •), say
 ADJ1: $ADJ \rightarrow green$ FROM n TO p
 such that the FROM position in the constituent matches the TO position in the active arc, then the "•" can be advanced, creating a new active arc:
 ARC2: $NP \rightarrow DET1 ADJ1 \bullet N$ FROM m TO p .

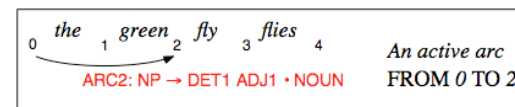


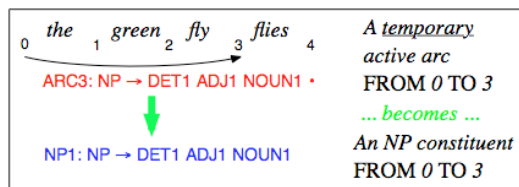
Chart Parser 6

- 6 **Conversion to Constituent:** If the process of advancing the "•" creates an active arc whose "•" is at the far right hand side of the rule: e.g.

ARC3: NP → DET1 ADJ1 N1• FROM 0 TO 3

then this arc is converted to a constituent (and is no longer an active arc).

NP1: NP → DET1 ADJ1 N1 FROM 0 TO 3.

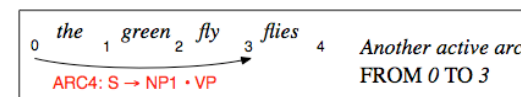


Not all active arcs are ever completed in this sense.

Chart Parser 7

- 7 Both lexical and phrasal constituents can be used in steps 3 and 4: e.g. if the grammar contains a rule $S \rightarrow NP VP$, then as soon as the constituent NP1 discussed in step 5 is created, it will be possible to make a new active arc

ARC4: S → NP1 • VP FROM 0 TO 3



- 8 When subsequent constituents are created, they would have names like NP2, NP3, ..., ADJ2, ADJ3, ... and so on.
- 9 The goal of parsing is to get phrasal constituents (normally of type S) whose FROM is 0 and whose TO is the length of the sentence. *There will be several such constituents if the sentence is structurally ambiguous.*

Final State of the Chart for "The green fly flies"

Constituents

DET1: DET → "the" FROM 0 TO 1

ADJ1: ADJ → "green" FROM 1 TO 2

N1: N → "fly" FROM 2 TO 3

V1: V → "fly" FROM 2 TO 3

N2: N → "flies" FROM 3 TO 4

V2: V → "flies" FROM 3 TO 4

NP1: NP → DET1 ADJ1 N1 FROM 0 TO 3

VP1: VP → V2 FROM 3 TO 4

S1: S → NP1 VP1 FROM 0 TO 4

Active Arcs

ARC1: NP → DET1 • ADJ N FROM 0 TO 1

ARC2: NP → DET1 ADJ1 • N FROM 0 TO 2

ARC4: S → NP1 • VP FROM 0 TO 3

Homework

Two longer worked examples are available, one in the exercise set, and one at <http://www.cse.unsw.edu.au/~billw/cs9414/notes/nlp/grampars/largecan.html>

Limitations of Syntax in NLP

- It is reasonable to ask for syntactically correct programs, but unrealistic to ask for syntactically correct NL. Written NL material is sometimes correct, but spoken utterances are often ungrammatical. NL systems must be **syntactically and semantically robust**. That is, they must be able to process sentences with grammatical errors and unknown words.
- Some approaches have sought to be semantics-driven, to avoid the problem of how to deal with syntactically ill-formed text. However, some syntax is essential - else how do we distinguish between *Brad loves Angelina* and *Angelina loves Brad*?

Summary

We have looked at

- parsing in general
- formal grammars
- the Chomsky hierarchy of grammar types
- context-free grammars
- top-down parsing in Prolog
- bottom-up chart parsing