2 Simple Grammars and Parsers

2.3 Context-free phrase structure grammar

2.3.2 The grammar

$$G = \langle T, N, S, P \rangle$$

where

1. T is a finite set of **terminal symbols**,
2. N is a finite set of **nonterminal symbols**,
3. S is a **start symbol**, and
4. P is a finite set of **phrase-structure rules**

Terminal symbols: *"Toby"*, *"drinks"*, *"scotch"*, *"on"*, *"ice"*

Nonterminal symbols: S, NP, PName, VP, V, PP

Start symbol: S

Phrase-structure rules:

$S \rightarrow NP\ VP$	$VP \rightarrow V\ NP$
$NP \rightarrow PName$	$VP \rightarrow V$
$NP \rightarrow$ *scotch*	$VP \rightarrow VP\ PP$
$NP \rightarrow$ *ice*	$PName \rightarrow$ *Toby*
$NP \rightarrow$ *drinks*	$V \rightarrow$ *drinks*
$NP \rightarrow NP\ PP$	$PP \rightarrow$ *on* NP

Grammar 2.1: Example CF-PSG

The set of nonterminal symbols is disjoint from the set of terminal symbols; formally, we write that their intersection is empty:

$$T \cap N = \emptyset$$

In grammar 2.1, the nonterminal symbols are S, NP, PName, VP, V and PP.

The start symbol is one of the nonterminal symbols:

$$S \in N$$

It is a distinguished nonterminal used in defining the language generated by the grammar. In grammar 2.1 (and in most grammars), the start symbol is the nonterminal S.

If Σ is a finite set of symbols, then a **string** is a *finite* sequence of those symbols.

For the set $\{$*Toby, drinks*$\}$, we can form strings such as

Toby
Toby drinks
drinks Toby
drinks drinks drinks drinks

We also allow one string that has no symbols in it at all. This string is called the **empty string**.

It is conventional to use the Greek letter epsilon

$$\epsilon$$

to denote the empty string. (ϵ is not a member of the set of symbols from which strings are being formed, it is just a convention for denoting the empty string.)

For a set of symbols Σ, we write Σ^* to denote the set of all strings that can be formed from symbols in Σ.

Σ^* includes the empty string, which clearly has length zero, and it includes all possible strings of length one, length two, length three, and so on. Σ^* is thus an infinite set.

For the set $\{$*Toby, drinks*$\}$, $\{$*Toby, drinks*$\}^*$ is the set some of whose members are shown here:

$\{$ ϵ, *Toby, drinks, Toby Toby, Toby drinks, drinks Toby, drinks drinks, Toby Toby Toby, Toby Toby drinks, Toby drinks Toby,. . .* $\}$

We can now formally define the set P of context-free phrase-structure rules as

$$P \subset N \times (N \cup T)^*$$

- $(N \cup T)$ is the union of the nonterminal symbols and the terminal symbols to form a single set of all grammar symbols.
- $(N \cup T)^*$ is the set of all strings of grammar symbols, i.e. all strings that can be formed from the nonterminals and/or terminals. These are the possible rule *right-hand sides*.

Notational conventions:

- $A, B, C, \ldots$ as the nonterminal symbols of the language;
- $a, b, c, \ldots$ as the terminal symbols of the language;
- $X, Y, Z, \ldots$ as grammar symbols (either terminals or nonterminals);
- $w, x, y, z, \ldots$ as possibly empty sequences of terminals; and
- $\alpha, \beta, \gamma, \ldots$ as possibly empty sequences of terminals and/or nonterminals. (These are *meta*-variables ranging over non-terminals, terminals, etc.)

Derivations

Given a rule

$$A \rightarrow \beta$$

if we have a string

$$\alpha A \gamma$$

then we can rewrite this as

$$\alpha \beta \gamma$$

using the rule.

We say $\alpha A \gamma$ **directly derives** $\alpha \beta \gamma$ using $A \rightarrow \beta$.

This is written:

$$\alpha A \gamma \quad \Rightarrow \quad \alpha \beta \gamma$$

A concrete example using the rule $\mathrm{VP} \rightarrow \mathrm{V\ NP}$:

$$\mathrm{NP}\ \underline{\mathrm{VP}} \quad \Rightarrow \quad \mathrm{NP\ V\ NP}$$

We can 'summarize' the following sequence of rewrites (which uses the rules $\text{NP} \rightarrow \text{NP PP}$, $\text{PP} \rightarrow \textit{on}\ \text{NP}$ and $\text{NP} \rightarrow \text{PName}$ in that order):

$$\begin{aligned} \textit{drinks}\ \underline{\text{NP}} &\Rightarrow \textit{drinks}\ \text{NP}\ \underline{\text{PP}} \\ &\Rightarrow \textit{drinks}\ \underline{\text{NP}}\ \textit{on}\ \text{NP} \\ &\Rightarrow \textit{drinks}\ \text{PName}\ \textit{on}\ \text{NP} \end{aligned}$$

using the symbol $\stackrel{*}{\Rightarrow}$:

$$\textit{drinks}\ \text{NP} \stackrel{*}{\Rightarrow} \textit{drinks}\ \text{PName}\ \textit{on}\ \text{NP}$$

which says that *drinks* NP derives (in zero or more steps) *drinks* PName *on* NP. More simply we just say that *drinks* NP **derives** *drinks* PName *on* NP. (We also use $\stackrel{+}{\Rightarrow}$ to mean 'derives in one or more steps'.)

- $\stackrel{*}{\Rightarrow}$ is the reflexive, transitive closure of $\Rightarrow$
- $\stackrel{+}{\Rightarrow}$ is just the transitive closure of $\Rightarrow$.

Of most interest in defining languages is the case where a string containing only terminal symbols is derived. Since it contains no nonterminals it cannot be further ‘expanded’. For example:

$$
\begin{aligned}
\text{V } \underline{\text{NP}} &\Rightarrow \text{V } \underline{\text{PName}} \\
&\Rightarrow \underline{\text{V}}\ \textit{Toby} \\
&\Rightarrow \textit{drinks Toby}
\end{aligned}
$$

And of more interest still is where such a derivation begins with the start symbol of the grammar, as this is how we define the language generated by a grammar.

The language for some grammar $G = \langle T, N, S, P \rangle$, written $\mathcal{L}(G)$, is the set of all strings of terminal symbols, w, that can be derived from the grammar's start symbol, S:

$$\mathcal{L}(G) = \{w \in T^* \mid S \stackrel{*}{\Rightarrow} w\}$$

For a grammar to generate an infinite language it must contain some nonterminal, A, such that

- A is a **useful** nonterminal, where a nonterminal is useful if it can be deployed in generating strings of terminal symbols from the start symbol, i.e. if

  $$S \quad \stackrel{*}{\Rightarrow} \quad \alpha A \beta \quad \stackrel{*}{\Rightarrow} \quad w$$

  and

- in one or more steps, A can derive a string that contains A, i.e.

  $$A \quad \stackrel{+}{\Rightarrow} \quad \alpha A \beta$$

One trivial way in which some nonterminal can derive a string that contains another instance of that nonterminal ($A \overset{+}{\Rightarrow} \alpha A \beta$) is if the grammar contains one or more **recursive rules**.

A rule is recursive if the nonterminal which appears on the left-hand side of the arrow also appears on the right-hand side.

In grammar 2.1 an example is the rule $\mathrm{NP} \rightarrow \mathrm{NP\ PP}$. With this rule, grammar 2.1 can generate an infinite set of sentences: any NP in a derivation can be expanded using $\mathrm{NP} \rightarrow \mathrm{NP\ PP}$, which introduces another NP, so this too can be expanded using $\mathrm{NP} \rightarrow \mathrm{NP\ PP}$, and so on.

Thus, we can generate, e.g. *"Toby drinks scotch"*, *"Toby drinks scotch on ice"*, *Toby drinks scotch on ice on ice"*, and so on, i.e. sentences comprising *"Toby drinks scotch"* followed by zero, one or more instances of *"on ice"*.

A grammar that contained rules of the form $A \to aB$ and $B \to Ab$, or more complicated collections of rules along the same lines, would generate an infinite language (provided again that A is a useful nonterminal).

$\underline{\text{S}}$	$\Rightarrow$	$\underline{\text{NP}}$ VP	(S $\rightarrow$ NP VP)	(1)
	$\Rightarrow$	$\underline{\text{PName}}$ VP	(NP $\rightarrow$ PName)	(2)
	$\Rightarrow$	*Toby* $\underline{\text{VP}}$	(PName $\rightarrow$ *Toby*)	(3)
	$\Rightarrow$	*Toby* $\underline{\text{V}}$ NP	(VP $\rightarrow$ V NP)	(4)
	$\Rightarrow$	*Toby drinks* $\underline{\text{NP}}$	(V $\rightarrow$ *drinks*)	(5)
	$\Rightarrow$	*Toby drinks scotch*	(NP $\rightarrow$ *scotch*)	(6)

Figure 6: Leftmost derivation of *"Toby drinks scotch"*

$$
\begin{array}{lll}
\underline{\text{S}} & (\text{S} \rightarrow \text{NP VP}) & (1) \\
\Rightarrow \text{NP } \underline{\text{VP}} & (\text{VP} \rightarrow \text{V NP}) & (2) \\
\Rightarrow \text{NP } \underline{\text{V}} \textit{ scotch} & (\text{NP} \rightarrow \textit{scotch}) & (3) \\
\Rightarrow \underline{\text{NP}} \textit{ drinks scotch} & (\text{V} \rightarrow \textit{drinks}) & (4) \\
\Rightarrow \underline{\text{PName}} \textit{ drinks scotch} & (\text{NP} \rightarrow \text{PName}) & (5) \\
\Rightarrow \textit{Toby drinks scotch} & (\text{PName} \rightarrow \textit{Toby}) & (6)
\end{array}
$$

Figure 9: Rightmost derivation of *"Toby drinks scotch"*

2.3.3 The lexicon

The set of syntactic categories (nonterminals) is split into two *disjoint* sets:

- the **phrasal categories**, PC, and
- the **lexical categories** (or **preterminal** symbols), LC:

$$N = PC \cup LC$$

$$PC \cap LC = \emptyset$$

The phrase-structure rules are similarly split into two.

- those that relate phrasal categories to strings of phrasal or lexical categories; the grammar rules, GR
- those that relate lexical categories to words (terminal symbols); the lexical entries, LE, that form the lexicon.

$$GR \subset PC \times (PC \cup LC)^*$$

$$LE \subseteq LC \times T$$

This means that rules such as $\mathrm{PP} \rightarrow$ *on* NP are no longer allowed. Instead, the word *"on"* needs a lexical category, e.g. P, so that rules of GR can mention this category, and its relationship to the word *"on"* can be confined to the lexicon, LE.

We often use a different notation for lexical rules, e.g.:

$$\textit{drinks} : \mathrm{V}$$

Grammar	Lexicon
$S \rightarrow NP\ VP$	*Toby* : PName
$NP \rightarrow PName$	*scotch* : N
$NP \rightarrow N$	*ice* : N
$NP \rightarrow NP\ PP$	*drinks* : N
$VP \rightarrow V\ NP$	*drinks* : V
$VP \rightarrow V$	*on* : P
$VP \rightarrow VP\ PP$	
$PP \rightarrow P\ NP$	

Grammar 2.2: Grammar 2.1 reformulated to use a lexicon

2.4 Evaluation of Grammars

2.4.1 Observational adequacy

1. At a bare minimum, a grammar must describe the well-formed expressions of the language. The extent to which the strings that a grammar generates tally with those that human informants would judge to be grammatical is known as the **observational adequacy** of the grammar.
2. If there are expressions which humans would judge to be grammatical but which the grammar *cannot* generate, then the grammar is said to **undergenerate**.
3. If there are strings that humans would judge to be ungrammatical but which the grammar *does* generate, then the grammar is said to **overgenerate**.

Generative capacity

The power of a formalism is referred to as its **generative capacity**.

There are then other languages (though they may not be *natural* languages) that the CF-PSG formalism is not powerful enough to characterize precisely.

If we attempt to devise a CF-PSG to generate such languages, the grammars we come up with are doomed to undergenerate and/or overgenerate.

Type 3: The **regular grammars** contain rules of the form

$$A \rightarrow aB$$

$$A \rightarrow a$$

or

$$A \rightarrow \epsilon$$

These languages are referred to as **regular languages**.

Type 2: The **context-free grammars**, what we have been calling the 'context-free phrase-structure grammars' contain rules of the form

$$A \rightarrow \alpha$$

These grammars can describe a set of languages that is larger than but includes the regular languages, the **context-free languages**.

Type 1: The **context-sensitive grammars** contain rules are of the form

$$\alpha A \gamma \rightarrow \alpha \beta \gamma$$

where β is not empty.

They describe a set of languages that is larger than but includes the context-free languages, the **context-sensitive languages**.

Type 0: The **unrestricted grammars** place no restrictions on the format of the rules

$$\alpha \rightarrow \beta$$

They describe the **recursively enumerable languages**, which is a set of languages that is larger than but includes the context-sensitive languages.

Regular grammars cannot generate the set of strings of the form

$$a^n b^n$$

i.e. n copies of some terminal symbol a followed by the same number of copies of some terminal symbol b.

$$\begin{array}{ll} \mathrm{A} \rightarrow \epsilon & \mathrm{B} \rightarrow \epsilon \\ \mathrm{A} \rightarrow a\mathrm{A} & \mathrm{B} \rightarrow b\mathrm{B} \\ \mathrm{A} \rightarrow a\mathrm{B} & \end{array}$$

Grammar 2.3: A regular grammar for $a^n b^m$, $n, m \geq 0$

A derivation using grammar 2.3 of a string of the form $a^n b^n$, where $n = 3$:

$$
\begin{aligned}
\mathrm{A} &\Rightarrow a\mathrm{A} \\
&\Rightarrow aa\mathrm{A} \\
&\Rightarrow aaa\mathrm{B} \\
&\Rightarrow aaab\mathrm{B} \\
&\Rightarrow aaabb\mathrm{B} \\
&\Rightarrow aaabbb\mathrm{B} \\
&\Rightarrow aaabbb
\end{aligned}
$$

(this derivation uses two instances of $\mathrm{A} \rightarrow a\,\mathrm{A}$, one instance of $\mathrm{A} \rightarrow a\,\mathrm{B}$, three instances of $\mathrm{B} \rightarrow b\,\mathrm{B}$, and then one instance of $\mathrm{B} \rightarrow \epsilon$).

This seems to contradict the statement that such grammars cannot generate strings of the form $a^n b^n$. However, the grammar can also generate strings in which the number of *a*s and *b*s are not equal, e.g.

$$
\begin{aligned}
\mathrm{A} &\Rightarrow a\mathrm{A} \\
&\Rightarrow aa\mathrm{A} \\
&\Rightarrow aaa\mathrm{B} \\
&\Rightarrow aaab\mathrm{B} \\
&\Rightarrow aaab
\end{aligned}
$$

The grammar overgenerates.

There is no *regular* grammar that we could write that would both generate all strings of the form $a^n b^n$ and would exclude all strings of the form $a^m b^n$ where $m \neq n$. Regular grammars are doomed to overgenerate or undergenerate in this instance.

$$\mathrm{A} \longrightarrow \mathrm{aAb}$$

$$\mathrm{A} \longrightarrow \epsilon$$

Grammar 2.4: A CF-PSG for $a^n b^n$, $n, m \geq 0$

Is English Context-free?

(38) *"The terrorist (that the x)^n (y)^n escaped."*

The terrorist (that the policemen)^1 (are looking for)^1 escaped

*The terrorist (that the policemen)^1 escaped

The terrorist (that the policemen)^1 (that the court)^2 (is prosecuting)^1 (are looking for)^2 escaped

*The terrorist (that the policemen)^1 (are prosecuting)^1 (are looking for)^2 escaped

*The terrorist (that the policemen)^1 (that the court)^2 (are looking for)^1 escaped

Is English Context-sensitive?

Cross-serial (or **intersecting**) dependencies.

Constructions in which a string can be partitioned into two non-overlapping substrings, and where there is a dependency between each item in the first substring and its counterpart in the second.

(39) *"Toby, Andrew and Maria like beer, wine and scotch, respectively."*

The interpretation required is that Toby drinks beer, Andrew drinks wine and Maria drinks scotch.

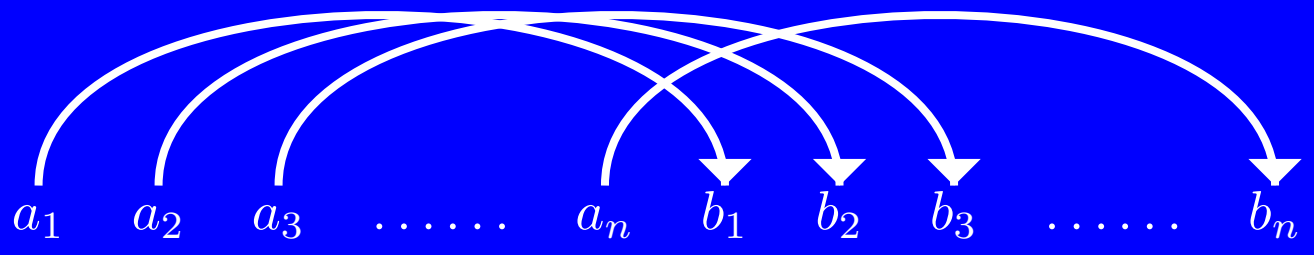


Figure 10: Intersecting dependencies

where a_1, a_2, and a_3 correspond to *"Toby"*, *"Andrew"* and *"Maria"*, and b_1, b_2, and b_3 correspond to *"beer"*, *"wine"* and *"scotch"*.

This argument is empirically false:

it is not necessary for there to be matching elements in the two halves of the construction.

Suppose Toby is reading a book and he exclaims:

(40) *"Here is a coincidence, all the words on this page are five letters long, except for the last three, which are one, two and three words long, respectively".*

The phrase *"the last three"* picks out the candidates for the matching, but without enumerating them; so we have *one* item in the first half of the construction and *three* in the second half.

Dutch

(41) *“. . . dat Jan [Marie Pieter Arabisch laat zien*
‘. . . that Jan Marie Pieter Arabic let see
schrijven].”
write’
“. . . *that Jan let Marie see Pieter write Arabic.*”

From a purely *syntactic* point of view, the only requirement in this construction is that the *number* of noun phrases appearing at the beginning of the bracketed sequence in (41) should be indentical to the number of verbs at its end. (I.e. it is an instance of $a^n b^n$.)

Which verb matches up semantically with which verb phrase is irrelevant syntactically because there is in Dutch (as in English) no overt syntactic marker to indicate which NP goes with which verb.

Swiss German

(42) *“. . . mer d’chind em Hans es*
‘. . . we the children-ACC the Hans-DAT the
huus lönd hälfe aastriiche.”
house-ACC let help paint’
“. . . *we let the children help Hans paint the house.*”

Each verb requires that its NP arguments be morphologically marked with an appropriate case. The consequence of this is that Swiss German exhibits a genuine cross-serial dependency (the dependency is *syntactic*).

2.4.2 Descriptive adequacy

Observational adequacy: the grammar generates all and only the expressions (i.e. strings of words) that a native speaker would judge to be grammatical.

A grammar is **descriptively adequate** if it is observationally adequate *and* assigns to the strings that it defines structural descriptions which correctly capture the linguistically significant generalisations about the language specified by the grammar.

Equivalence

A grammar G_1 is *weakly* equivalent to a grammar G_2 if and only if the language generated by G_1 is equal to the language generated by G_2.

A grammar G_1 is *strongly* equivalent to a grammar G_2 if and only if they are weakly equivalent *and* the phrase-structures assigned by G_1 to the strings it generates are, in some sense, the 'same' as the phrase-structures assigned by G_2 to the same strings.

Grammar a	Grammar b
NP → Det Nbar	NP → DetP N
Nbar → Adj Nbar	DetP → DetP Adj
Nbar → N	DetP → Det
Det → *that*	Det → *that*
Adj → *tipsy*	Adj → *tipsy*
N → *knight*	N → *knight*

Grammar 2.5: weakly equivalent grammars

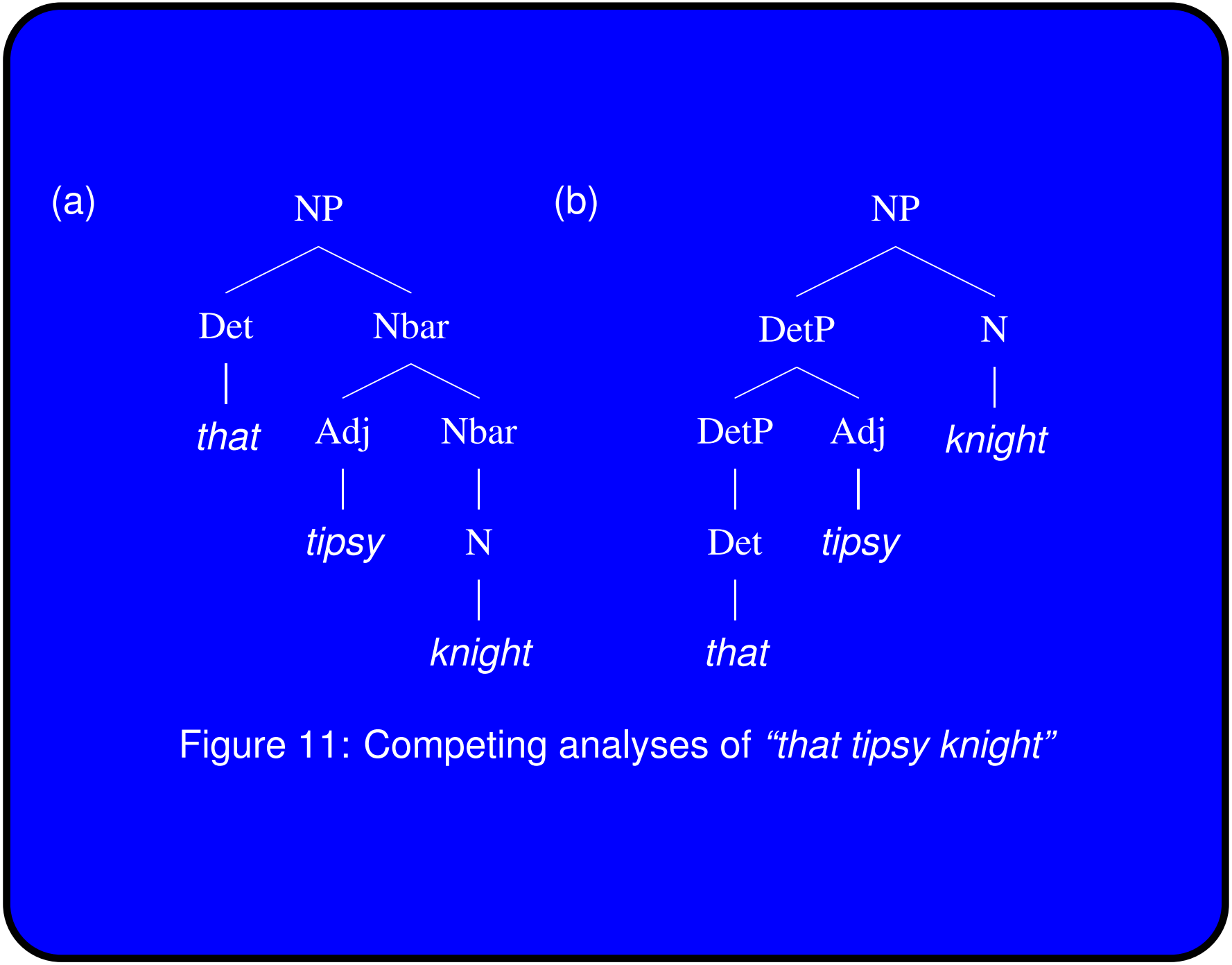


Figure 11: Competing analyses of *"that tipsy knight"*

- Applied to two grammars expressed in the same formalism, e.g. two CF-PSGs, strong equivalence is virtually a vacuous concept. Applied to two CF-PSGs, G_1 and G_2, for example, it comes close to requiring that G_1 *is* G_2. The extent to which CF-PSGs G_1 and G_2 can differ and yet remain strongly equivalent is severely limited: since the names of the syntactic categories are arbitrary, we can say that G_1 and G_2 are strongly equivalent if G_2 uses a consistent renaming of the nonterminals of G_1.
- Strong equivalence is a more challenging notion if the grammars being compared are expressed in different formalisms, as then the question of whether the two grammars assign the 'same phrase-structures' to corresponding strings will not be simply a question of whether the two grammars comprise structurally similar phrase-structure rules.

- Miller, Philip (2001) *Strong Generative Capacity: The Semantics of Linguistic Formalism.* CSLI Publications, Stanford CA.

2.4.3 Explanatory adequacy

Explanatory adequacy is something that theories of language, rather than descriptions of individual languages, should aspire to. An explanatorily adequate theory should correctly characterise the notion ‘possible human language’.

3 An Introduction to Parsing

$S \rightarrow NP\ VP$	*Toby* : PName
$NP \rightarrow PName$	*scotch* : N
$NP \rightarrow N$	*ice* : N
$NP \rightarrow NP\ PP$	*drinks* : N
$VP \rightarrow V\ NP$	*drinks* : V
$VP \rightarrow V$	*on* : P
$VP \rightarrow VP\ PP$	
$PP \rightarrow P\ NP$	

Grammar 3.1: Illustrative context-free phrase-structure grammar

3.1 Top-down and bottom-up parsing

- **recognizer**: a program which, given an input string, determines whether or not the string is a well-formed expression of the language described by the grammar. Such a program returns simply a “yes” or “no” answer.
- **parser**: recognizer which additionally, when the string *is* well-formed, it does not simply return a “yes” answer; rather, it will have computed one or more phrase-structure trees and it returns these as its answer.

Two parsing strategies:

- In **top-down** parsing (**hypothesis-driven**, **goal-driven** or **expectation-driven** parsing), the parser tries to build phrase-structure trees starting from their root nodes, adding branches and nodes until it reaches the leaves. Hence, such parsers try to find a derivation from the grammar's start symbol to the given input string.
- **Bottom-up** (or **data-driven**) parsers build phrase-structure trees from the leaves, adding branches and nodes until they reach the root. Hence, such parsers search for derivations in the opposite direction to that used by top-down parsers, beginning with the words themselves and trying to find a sequence of rewrites that lead to a single category that can form the root of the tree.

3.2 Working from left to right

S	$\Rightarrow$	NP VP	(S $\rightarrow$ NP VP)	(1)
	$\Rightarrow$	PName VP	(NP $\rightarrow$ PName)	(2)
	$\Rightarrow$	*Toby* VP	(*Toby* : PName)	(3)
	$\Rightarrow$	*Toby* V NP	(VP $\rightarrow$ V NP)	(4)
	$\Rightarrow$	*Toby drinks* NP	(*drinks* : V)	(5)
	$\Rightarrow$	*Toby drinks* NP PP	(NP $\rightarrow$ NP PP)	(6)
	$\Rightarrow$	*Toby drinks* N PP	(NP $\rightarrow$ N)	(7)
	$\Rightarrow$	*Toby drinks scotch* PP	(*scotch* : N)	(8)
	$\Rightarrow$	*Toby drinks scotch* P NP	(PP $\rightarrow$ P NP)	(9)
	$\Rightarrow$	*Toby drinks scotch on* NP	(*on* : P)	(10)
	$\Rightarrow$	*Toby drinks scotch on* N	(NP $\rightarrow$ N)	(11)
	$\Rightarrow$	*Toby drinks scotch on ice*	(*ice* : N)	(12)

Figure 12: One top-down, left-to-right parse of *"Toby drinks scotch on ice"*

Toby drinks scotch on ice	$\Leftarrow$ PName *drinks scotch on ice*	(*Toby* : PName)	(1)
	$\Leftarrow$ NP *drinks scotch on ice*	(NP $\rightarrow$ PName)	(2)
	$\Leftarrow$ NP V *scotch on ice*	(*drinks* : V)	(3)
	$\Leftarrow$ NP V N *on ice*	(*scotch* : N)	(4)
	$\Leftarrow$ NP V NP *on ice*	(NP $\rightarrow$ N)	(5)
	$\Leftarrow$ NP V NP P *ice*	(*on* : P)	(6)
	$\Leftarrow$ NP V NP P N	(*ice* : N)	(7)
	$\Leftarrow$ NP V NP P NP	(NP $\rightarrow$ N)	(8)
	$\Leftarrow$ NP V NP PP	(PP $\rightarrow$ P NP)	(9)
	$\Leftarrow$ NP V NP	(NP $\rightarrow$ NP PP)	(10)
	$\Leftarrow$ NP VP	(VP $\rightarrow$ V NP)	(11)
	$\Leftarrow$ S	(S $\rightarrow$ NP VP)	(12)

Figure 13: One bottom-up, left-to-right parse of *"Toby drinks scotch on ice"*

3.2.1 Working from right to left

$\underline{\text{S}}$	$\Rightarrow$	NP $\underline{\text{VP}}$	(S $\rightarrow$ NP VP)	(1)
	$\Rightarrow$	NP V $\underline{\text{NP}}$	(VP $\rightarrow$ V NP)	(2)
	$\Rightarrow$	NP V NP $\underline{\text{PP}}$	(NP $\rightarrow$ NP PP)	(3)
	$\Rightarrow$	NP V NP P $\underline{\text{NP}}$	(PP $\rightarrow$ P NP)	(4)
	$\Rightarrow$	NP V NP P $\underline{\text{N}}$	(NP $\rightarrow$ N)	(5)
	$\Rightarrow$	NP V NP $\underline{\text{P}}$ *ice*	(*ice* : N)	(6)
	$\Rightarrow$	NP V $\underline{\text{NP}}$ *on ice*	(*on* : P)	(7)
	$\Rightarrow$	NP V $\underline{\text{N}}$ *on ice*	(NP $\rightarrow$ N)	(8)
	$\Rightarrow$	NP $\underline{\text{V}}$ *scotch on ice*	(*scotch* : N)	(9)
	$\Rightarrow$	$\underline{\text{NP}}$ *drinks scotch on ice*	(*drinks* : V)	(10)
	$\Rightarrow$	$\underline{\text{PName}}$ *drinks scotch on ice*	(NP $\rightarrow$ PName)	(11)
	$\Rightarrow$	*Toby drinks scotch on ice*	(*Toby* : PName)	(12)

Figure 14: One top-down, right-to-left parse of *"Toby drinks scotch on ice"*

	Left-to-right	**Right-to-left**
Top-down	finds leftmost derivations	finds rightmost derivations
Bottom-up	finds rightmost derivations (in reverse)	finds leftmost derivations (in reverse)

Table 1: Four types of parser

3.3 Search in parsing

3.3.1 The need for search

$$
\begin{array}{lcll}
\underline{\text{S}} & \Rightarrow & \underline{\text{NP}}\ \text{VP} & (\text{S} \rightarrow \text{NP VP}) \quad (1) \\
 & \Rightarrow & \underline{\text{PName}}\ \text{VP} & (\text{NP} \rightarrow \text{PName}) \quad (2) \\
 & \Rightarrow & \textit{Toby}\ \underline{\text{VP}} & (\textit{Toby} : \text{PName}) \quad (3)
\end{array}
$$

The VP is to be rewritten. But there are in fact three rules in the grammar that say how to expand a VP:

$$\text{VP} \rightarrow \text{V NP}$$

$$\text{VP} \rightarrow \text{V}$$

$$\text{VP} \rightarrow \text{VP PP}$$

$\underline{\text{S}}$	$\Rightarrow$	$\underline{\text{NP}}$ VP	(S $\rightarrow$ NP VP)	(1)
	$\Rightarrow$	$\underline{\text{PName}}$ VP	(NP $\rightarrow$ PName)	(2)
	$\Rightarrow$	*Toby* $\underline{\text{VP}}$	(*Toby* : PName)	(3)
	$\Rightarrow$	*Toby* $\underline{\text{VP}}$ PP	(VP $\rightarrow$ VP PP)	(4)
	$\Rightarrow$	*Toby* $\underline{\text{V}}$ NP PP	(VP $\rightarrow$ V NP)	(5)
	$\Rightarrow$	*Toby drinks* $\underline{\text{NP}}$ PP	(*drinks* : V)	(6)
	$\Rightarrow$	*Toby drinks* $\underline{\text{N}}$ PP	(NP $\rightarrow$ N)	(7)
	$\Rightarrow$	*Toby drinks scotch* $\underline{\text{PP}}$	(*scotch* : N)	(8)
	$\Rightarrow$	*Toby drinks scotch* $\underline{\text{P}}$ NP	(PP $\rightarrow$ P NP)	(9)
	$\Rightarrow$	*Toby drinks scotch on* $\underline{\text{NP}}$	(*on* : P)	(10)
	$\Rightarrow$	*Toby drinks scotch on* $\underline{\text{N}}$	(NP $\rightarrow$ N)	(11)
	$\Rightarrow$	*Toby drinks scotch on ice*	(*ice* : N)	(12)

Figure 15: Another top-down, left-to-right parse of *"Toby drinks scotch on ice"*

$$
\begin{array}{lcll}
\underline{\text{S}} & \Rightarrow & \underline{\text{NP}}\ \text{VP} & (\text{S} \rightarrow \text{NP VP}) \quad (1) \\
 & \Rightarrow & \underline{\text{PName}}\ \text{VP} & (\text{NP} \rightarrow \text{PName}) \quad (2) \\
 & \Rightarrow & \textit{Toby}\ \underline{\text{VP}} & (\textit{Toby} : \text{PName}) \quad (3) \\
 & \Rightarrow & \textit{Toby}\ \underline{\text{V}} & (\text{VP} \rightarrow \text{V}) \quad (4) \\
 & \Rightarrow & \textit{Toby drinks} & (\textit{drinks} : \text{V}) \quad (5)
\end{array}
$$

Figure 16: Unsuccessful top-down, left-to-right parse of *"Toby drinks scotch on ice"*

$$\underline{\textit{Toby}}\ \textit{drinks scotch on ice} \Rightarrow \underline{\text{PName}}\ \textit{drinks scotch on ice} \quad (\textit{Toby} : \text{PName}) \quad (1)$$

$$\Rightarrow \text{NP}\ \underline{\textit{drinks}}\ \textit{scotch in ice} \quad (\text{NP} \rightarrow \text{PName}) \quad (2)$$

Here *"drinks"* can be rewritten as a V or as an N, (using *drinks* : V and *drinks* : N respectively).

There is also a choice in the left-to-right, bottom-up parser of whether to rewrite the leftmost terminal or to rewrite a sequence of nonterminals, as at the end of step (3):

$$\underline{\textit{Toby}}\ \textit{drinks scotch on ice} \Rightarrow \underline{\text{PName}}\ \textit{drinks scotch on ice} \quad (\textit{Toby} : \text{PName}) \quad (1)$$

$$\Rightarrow \text{NP}\ \underline{\textit{drinks}}\ \textit{scotch in ice} \quad (\text{NP} \rightarrow \text{PName}) \quad (2)$$

$$\Rightarrow \text{NP V}\ \textit{scotch on ice} \quad (\textit{drinks} : \text{V}) \quad (3)$$

Here *"scotch"*, as the leftmost terminal, can be rewritten as an N (using *scotch* : N) or V can be rewritten as a VP (using VP $\rightarrow$ V).

2 reasons why natural language parsers must search:

- Some strings will be structurally ambiguous. Assuming that in these cases we wish to find all parses of the string, then, having found a parse, the only way of discovering whether there are other parses is by trying some of the unexplored rewrites.
- Although some choices do not lead to successful derivations, a parser of a natural language cannot in general know at the time it is faced with these choices which will be successful and which will not. The parser must explore all of them, and if one of them reaches a dead-end, it must have a way of returning to some of the unexplored options in case these prove fruitful.

Search trees

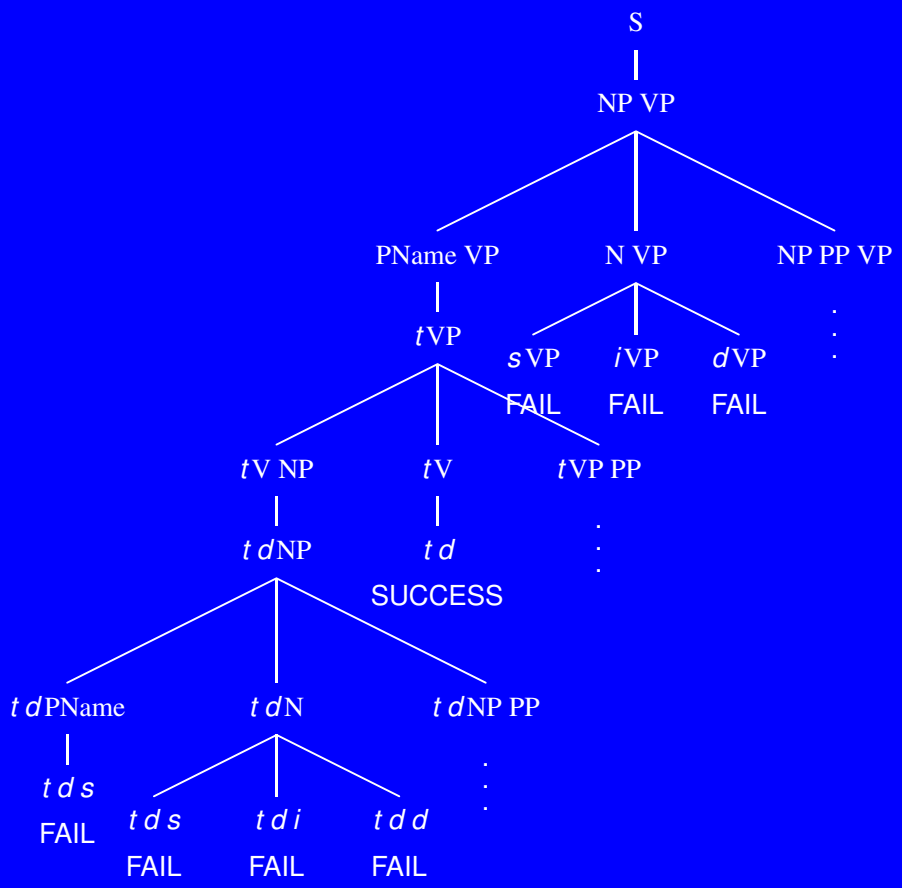


Figure 17: A fragment of the search tree for simple left-to-right, top-down parsing of *"Toby drinks"*

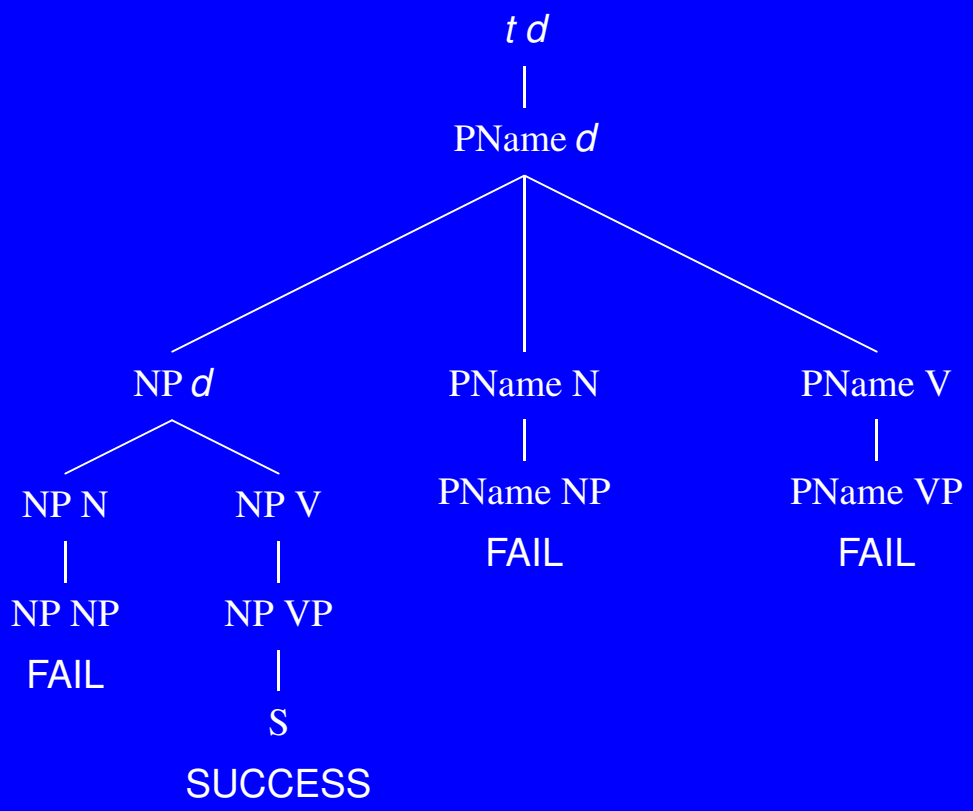
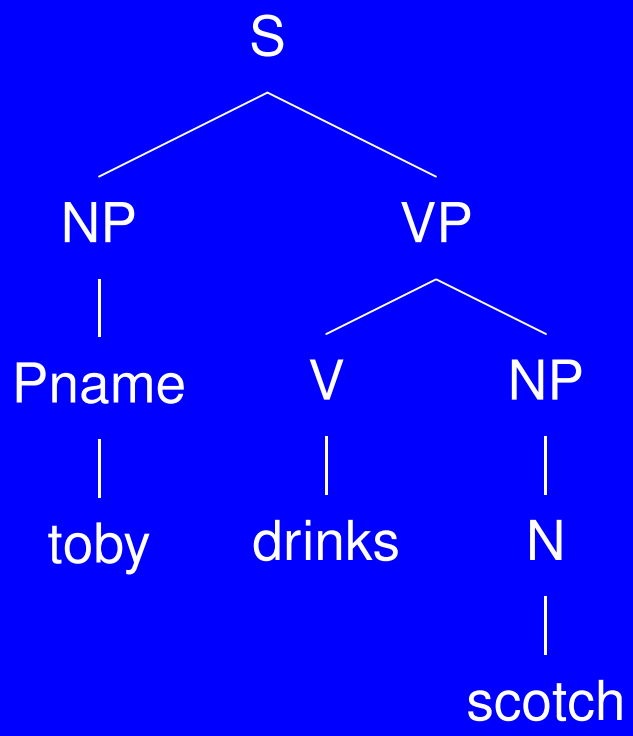


Figure 18: A fragment of the search tree for shift-reduce (left-to-right, bottom-up) parsing of *"Toby drinks"*

Here a path fails if it comprises only nonterminals, they cannot be rewritten further, and yet a single nonterminal (S) has not been attained.

3.3.2 Search strategies

- Depth first: the tree is explored path by path
- Breadth first; the tree is explored level by level



If the search tree of figure 18 is searched breadth-first, then *t d* would be rewritten as PName *d*, and then all this string's immediate 'descendants' would be computed: NP *d*, PName N and PName V. Each of these is then taken in turn and all of its immediate 'descendants' are computed: NP N, NP V, PName NP, and PName VP. This would go on, level by level, until the whole tree had been traversed.

3.3.3 Infinite search

Left recursion

$$A \stackrel{+}{\Rightarrow} A\alpha$$

A left-recursive grammar gives rise to infinite paths in the search trees of *left-to-right, top-down* parsers.

$$\begin{array}{lcll} \underline{\text{S}} & \Rightarrow & \underline{\text{NP}}\ \text{VP} & (\text{S} \rightarrow \text{NP VP}) \quad (1) \\ & \Rightarrow & \underline{\text{NP}}\ \text{PP VP} & (\text{NP} \rightarrow \text{NP PP}) \quad (2) \\ & \Rightarrow & \underline{\text{NP}}\ \text{PP PP VP} & (\text{NP} \rightarrow \text{NP PP}) \quad (3) \\ & \Rightarrow & \underline{\text{NP}}\ \text{PP PP PP VP} & (\text{NP} \rightarrow \text{NP PP}) \quad (4) \end{array}$$

Figure 19: The initial rewrites in an infinite left-to-right, top-down parse

In this figure, NPs can be indefinitely rewritten as NPs followed by PPs.

A right-recursive grammar gives rise to infinite paths in the search trees of *right-to-right, top-down* parsers.

Solutions:

- use a different grammar that recognizes the same language but which is not left-recursive.
- for CF-PSGs, there are algorithms that can be implemented as computer programs for automatically converting left-recursive CF-PSGs into weakly equivalent non-left-recursive CF-PSGs.
- changing the parser.
- left- (and right-) recursion do *not* create infinite paths in the search tree for bottom-up parsers: since the parser makes immediate use of the input string, the number of rewrites it carries out with left- (or right-) recursive rules is constrained. components).

3.3.4 ϵ-productions

$$\text{NP} \rightarrow \epsilon$$

An attempted bottom-up parse of *"Toby drinks scotch"* for a grammar that contains this ϵ-production.

$$
\begin{array}{lll}
\_\ \textit{Toby drinks scotch} & \Leftarrow \_\ \text{NP}\ \textit{Toby drinks scotch} & (\text{NP} \rightarrow \epsilon) \quad (1) \\
& \Leftarrow \_\ \text{NP NP}\ \textit{Toby drinks scotch} & (\text{NP} \rightarrow \epsilon) \quad (2) \\
& \Leftarrow \_\ \text{NP NP NP}\ \textit{Toby drinks scotch} & (\text{NP} \rightarrow \epsilon) \quad (3)
\end{array}
$$

Figure 20: The initial rewrites in an infinite left-to-right bottom-up parse

The naïve bottom-up parser can use an infinite number of applications of the ϵ-production at the front of the string, between each pair of words in the string, and at the end of the string.

- We can abandon exhaustive search, with the consequential possibility of missing parses.
- We can convert the grammar that contains ϵ-productions into a weakly equivalent grammar that does not contain ϵ-productions, or use a grammar formalism (such as Head-driven Phrase Structure Grammar) that does not utilize ϵ-productions.
- ϵ-productions do not create an infinite search tree for top-down parsing: the top-down approach will propose only a finite number of empty constituents. Thus, we could use a top-down parser (or a parser that mixes bottom-up and top-down processing) instead.

3.4 Computational Complexity

Resources: the amount of time and/or memory space, needed to solve certain computational problems.

- Measures of running time are expressed as functions of the size of the input to the problem, showing how the running time varies as the size of the input varies.
- Computational complexity focuses on *worst-case* running times.

Tractable problems: those whose 'best' algorithms have worst-case running times that are characterized by **polynomial** functions.

A polynomial function is one of the form:

$$a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$$

where k is a non-negative integer, $a_0, a_1, \ldots, a_k$, the so-called *coefficients*, are real numbers, and $a_k \neq 0$, and n is the variable in terms of which the function is being defined.

Examples of polynomial functions are:

$$5n^3 + 2n^2 + 6n + 4$$

$$4n^2 + 2$$

$$6n^3$$

$$n^2$$

In some of these, certain coefficients are either one or zero, allowing some simplification (e.g. the polynomial n^2 is a simplification of the polynomial $1 \times n^2 + 0 \times n + 0$).

Intractable problems: those which we believe admit only solutions whose worst-case running times are characterized by **exponential** functions.

An exponential function is one that contains a term such as a^n, where a is a non-zero real number and n is the variable in terms of which the function is being defined.

	n	0	1	2	3	4	5	6	7	8	9	10
A polynomial function	n^2	0	1	4	9	16	25	36	49	64	81	100
An exponential function	2^n	1	2	4	8	16	32	64	128	256	512	1024

Table 2: Relative growth of polynomial and exponential functions

For low values of n, the value of 2^n is lower than that of n^2, also the crossover point after which the exponential will be bigger than the polynomial is quite a low value here ($n = 4$). However, the main point is that the exponential *grows* much more quickly than the polynomial, and this, according to computational complexity theory, which focuses on worst cases as the input gets ever bigger, is what makes problems whose solutions take exponential time intractable.

Membership of a list

- If this search is implemented as a serial search, testing each item of the list in turn from the start to the end, then in the *worst case* every item will have to be checked.
- For n items in the list, the worst-case execution time is therefore kn, where k is some constant saying how much time each operation takes.
- the algorithm might also carry out certain other operations such as steps of initialization or termination. Therefore, the running time is actually computed by a formula such as $kn + c$, where c is a constant characterizing the amount of time these extra operations take but showing that this extra time is not dependent on the length of the list.
- This function is a polynomial function. (It is of the form $a_1 n + a_0$.)

- In complexity theory it is conventional to ignore the terms of the polynomial other than the one being raised to the highest power.
- Thus we would say that the serial search algorithm's worst case time is kn, ignoring the term c, because for large values of n, the contribution to the overall running time made by the extra term c can be regarded as negligible.
- We also suppress the value k, since it is only the highest power that is of relevance to the growth of the function.
- the growth of the function is often shown using what is called 'big oh notation', where we would write that our search algorithm takes time $O(n)$ (of order n).

- Linear search is only one algorithm for searching a list.
- As it turns out, all other ‘sensible’ algorithms also take polynomial time.
- there is an algorithm called binary search, which takes $k \log_2 n + c$ time, or $O(\log_2 n)$, which is also a polynomial function.
- Complexity theory therefore designates search of a list to be a tractable computational *problem*.

- The problem is to determine whether arbitrary formulae of propositional logic are satisfiable
- to determine whether some formula such as

$$\neg(P \rightarrow Q) \wedge (Q \vee (R \rightarrow \neg P))$$

  has an assignment of truth values to the propositional variables P, Q and R that makes the whole formula true.
- A simple solution would be to try out every possible assignment of true and false to the propositional variables, looking for an assignment that makes the whole formula true.
- Effectively then, the task is one of constructing rows of the truth-table for this formula, each one being an assignment. In the worst case, *all* rows would need to be constructed.

- For n variables, there are 2^n possible assignments and so 2^n rows would need to be constructed.
- Therefore, this algorithm's worst case time is proportional to 2^n, where n is the number of variables, making this an exponential solution.
- Since no polynomial time solution is known, we hypothesize that this problem is intractable.

- The time complexity of CF-PSG *recognition* is polynomial.
- We know this because there exist programs whose worst case time on a string of n words is proportional to n^3;
- one such algorithm is called 'Earley's algorithm' and can be embodied in a program called a *chart parser*

- The time complexity of CF-PSG *parsing* can be exponentially related to the number of words in the string.

We illustrate this by looking at compound nominals. To allow nominal compounding, our CF-PSG might include a rule such as

$$\mathrm{N} \rightarrow \mathrm{N}\ \mathrm{N}$$

- the compound nominal *“tin can”* has a single parse (making use of the rule once);
- the compound nominal *“tin can opener”* has two parses (both using the rule twice),

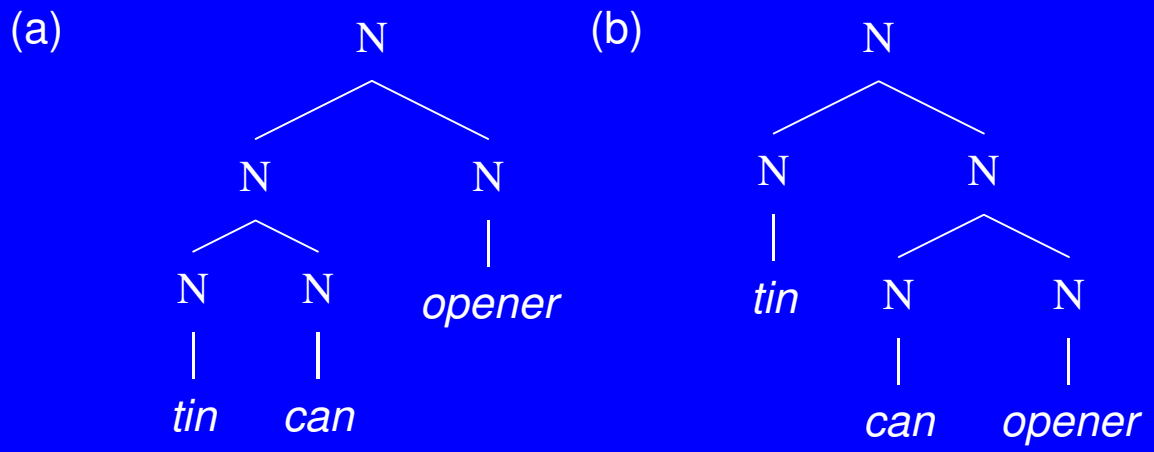


Figure 21: Two phrase-structure trees for *“tin can opener”*

- Compound nominals comprising three nouns have five parses (one left-branching, one right-branching and three in between);
- those with four nouns have six parses;
- those with five nouns have fourteen parses;
- those with six have forty-two;
- and those with seven, have 132. Such as the infamous:

(43) *“Long-term car park courtesy vehicle pick-up point.”*

The number of parses grows exponentially according to the Catalan series, the first ten of whose terms are as follows:

$$1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, \ldots$$

The Catalan series shows how many ways a string of length n can be 'parenthesized', and is defined by

$$Cat(n) = \left(\begin{array}{c} 2n \\ n \end{array} \right) - \left(\begin{array}{c} 2n \\ n-1 \end{array} \right)$$

where $\left(\begin{array}{c} n \\ r \end{array} \right)$ is the number of ways to pick r objects out of a set of n objects:

$$\left(\begin{array}{c} n \\ r \end{array} \right) = \frac{n!}{r!(n-r)!}$$

where $n!$ is n factorial, i.e. $n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$.

- Given that an input string might have a number of parses that is exponentially related to its length, it follows that the CF-PSG parsing problem is worst-case exponential.
- Thus the CF-PSG parsing problem is intractable.

- Complexity theory, as we have seen, is concerned with worst cases.
- when one focuses on normal or average cases rather than worst cases, results are less gloomy.
- in practice, n, the number of words in the input string, is not going to get very large: typical lengths will be ten to thirty words and strings longer than fifty words will be very rare.
- In this case, although CF-PSG parsing is exponential in the length of the input string, this may not be too problematic.
- An algorithm taking $k2^n$ time only takes longer than one taking kn^3 time when $n = 10$ (although this assumes that k is the same in both cases).
- If the polynomial algorithm actually takes $30n^3$ and the exponential takes 3×2^n, then the exponential algorithm is better than the polynomial one up to and including $n = 15$.

- the factors that complexity theory normally *ignores* (such as the lower order terms in the functions and the values of the constants) come into play when we look at normal or average case behaviour.

- One of the most significant of these will be the time spent accessing the grammar, and this is likely to be something that we can express as a function of the number of rules in the grammar, usually written as |G|.
- Suppose, as is the case for some of the n^3 recognizers mentioned above that recognition takes time proportional to |G|^2, i.e. the worst case recognition time is k |G|^2 n^3 or O(|G|^2 n^3).
- In practical grammars, |G| could be quite large.
- if |G| is 750 and we ignore k, then for sentences of fifteen words, the execution time is 750^2 $(= 562500)$ grammar access operations multiplied by 15^3 $(= 3375)$.
- It is conceivable that grammar access operations could dominate the processing.

4 Parsing in Prolog

4.1 Simple top-down parsing in Prolog

$S \rightarrow NP\ VP$	*Toby* : NP
$VP \rightarrow V$	*scotch* : NP
$VP \rightarrow V\ NP$	*drinks* : NP, V

Grammar 4.1: Simple grammar for top-down parsing

```
s  ---> [np, vp].
vp ---> [v].
vp ---> [v, np].

np ===> [toby].
np ===> [scotch].
np ===> [drinks].
v  ===> [drinks].
```

Grammar 4.2: A grammar written in BH-GRN

```
:- op(1200, xfx, --->).
:- op(1200, xfx, ===>).
```

4.1.1 A top-down, depth-first parser in Prolog

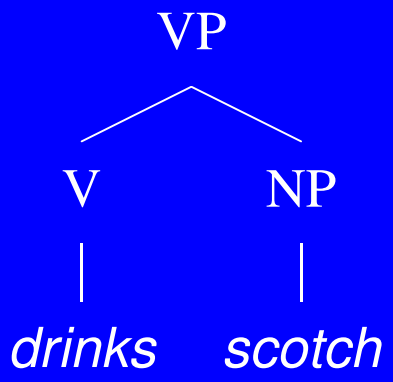
```
% progtddf.pl
:- op(1200, xfx, --->).
:- op(1200, xfx, ===>).
td_parse(Cat,[Word|RestofString],RestofString) :-
    (Cat===> [Word]).
td_parse(Mother, S0,S) :-
     (Mother ---> Daughters),
    td_parse_dtrs(Daughters, S0, S).
td_parse_dtrs([],S, S).
td_parse_dtrs([Cat|Cats], S0, S) :-
    td_parse(Cat, S0, S1),
    td_parse_dtrs(Cats, S1, S).
```

Program 4.1: A top-down, depth-first Prolog parser

```
?- td_parse(s, [toby, drinks], []).
```

4.1.2 Building phrase-structure trees

Representing trees



Trees as lists:

```
[vp, [v, drinks], [np, scotch]]
```

[VP [V *drinks*] [NP *scotch*]]

Trees as terms, with the function symbol corresponding to the label of the parent node and the arguments corresponding to the children:

```
vp(v(drinks), np(scotch))
```

```
% progtddfparse.pl
:- op(1200, xfx, --->).
:- op(1200, xfx, ===>).
td_parse(Cat,[Cat, Word], [Word|Rest],Rest) :-
    (Cat===> [Word]).
td_parse(Mother, [Mother | Dtrs], S0,S) :-
     (Mother ---> Daughters),
     td_parse_dtrs(Daughters, Dtrs,  S0,S).
td_parse_dtrs([], [], S, S).
td_parse_dtrs([Cat | Cats],[Dtr | Dtrs], S0,S) :-
    td_parse(Cat, Dtr, S0,S1),
    td_parse_dtrs(Cats, Dtrs, S1,S).
```

Program 4.2: A top-down, depth-first Prolog parser that builds trees

```
?- td_parse(s,  Tree, [toby, drinks, scotch], []).
Tree = [s,[np,toby],[vp,[v,drinks],[np,scotch]]]
 yes
```

4.1.3 Depth-bounded parsing

```
% progdbtdparser.pl
:- op(1200, xfx, --->).
:- op(1200, xfx, ===>).
td_parse(Cat, Depth, [Word|RestofString],RestofString) :-
    (Cat===> [Word]).
td_parse(Mother, Depth, S0,S) :-
     Depth > 0, NewDepth is Depth - 1,
     (Mother ---> Daughters),
    td_parse_dtrs(Daughters, NewDepth, S0,S).
td_parse_dtrs([], _, S, S).
td_parse_dtrs([Cat | Cats], Depth, S0,S) :-
    td_parse(Cat, Depth, S0,S1),
    td_parse_dtrs(Cats, Depth, S1,S).
```

Program 4.3: Depth-bounded top-down Prolog parser

A suitable maximum depth will be the length of the input string multiplied by the number of rules in the grammar whose right-hand sides are of length one or zero: this will never be too small but for some sentences it may be very generous.

```
s ---> [np, vp].
vp ---> [v].
vp ---> [v, np].
vp ---> [vp, pp].
pp ---> [p, np].
np ---> [np, pp].

np ===> [toby].
np ===> [scotch].
np ===> [drinks].
np ===> [ice].
v ===> [drinks].
p ===> [on].
```

Grammar 4.3: A grammar with left-recursion

4.2 Simple bottom-up parsing in Prolog

$$\begin{aligned} \text{S} &\rightarrow \text{NP VP} & \text{NP} &\rightarrow \textit{Toby} \\ \text{VP} &\rightarrow \text{V NP} & \text{NP} &\rightarrow \textit{scotch} \\ \text{VP} &\rightarrow \text{V} & \text{NP} &\rightarrow \textit{drinks} \\ & & \text{V} &\rightarrow \textit{drinks} \end{aligned}$$

Grammar 4.4: A simple CF-PSG without an explicit lexicon

4.2.1 A shift-reduce parser

A shift-reduce parser employs two data structures (other than the grammar):

- a **buffer** that contains the unprocessed part of the input string, and
- a **stack** (or last-in-first-out store) on which a parse is gradually assembled.

The state of these two data structures at any point in the algorithm is referred to as the parser's **configuration**.

At any step, the parser can carry out one of two operations. The two operations are:

Shift: the next word of the input string is moved from the buffer to the top of the stack: if the next word of the input string is some a and the stack is presently

$[\ldots X]$,

where X is the top of the stack, then the stack becomes

$[\ldots X\ a]$.

Reduce: the parser finds a rule whose right-hand side matches the constituents on the stack, one by one, starting from the top and replaces these constituents by the category on the left-hand side of the rule: if there is a rule

$A \rightarrow \alpha$

and the stack contains

$[\ldots\ \alpha]$,

where α is the sequence of symbols on the top of the stack, the categories α on the stack are popped (deleted) and the left-hand side of the rule is pushed (inserted onto the stack at its top), and so the stack becomes

$[\ldots\ A]$.

Here are some of the steps involved in finding the parse of *"Toby drinks scotch"* using grammar 4.4.

The parser's initial configuration is one in which the stack is empty and the input buffer contains the whole input string:

[ ]	⟨*Toby drinks scotch*⟩
Stack	Input buffer

Shift *"Toby"*, onto the stack:

[*Toby*]	⟨*drinks scotch*⟩

Reduce with rule NP → *Toby*, so the word *"Toby"* can be replaced by the left-hand side of the rule, NP:

[NP]	⟨*drinks scotch*⟩

Shift the next word in the input buffer follows:

[NP *drinks*]	⟨*scotch*⟩

Reduce using $\mathrm{V} \rightarrow$ *drinks*; the top item on the stack is replaced by V:

$$[\mathrm{NP\ V}] \qquad \langle \textit{scotch} \rangle$$

Carry out another shift:

$$[\mathrm{NP\ V}\ \textit{scotch}] \qquad \langle\ \rangle$$

"scotch" can be reduced to NP using $\mathrm{NP} \rightarrow$ *scotch*:

$$[\mathrm{NP\ V\ NP}] \qquad \langle\ \rangle$$

The top two items of the stack match the right-hand side of the rule $\mathrm{VP} \rightarrow \mathrm{V\ NP}$. Reduce by popping these two items and pushing the left-hand side of the rule:

$$[\mathrm{NP\ VP}] \qquad \langle\ \rangle$$

The top items of the stack match the right-hand side of the rule $\mathrm{S} \rightarrow \mathrm{NP\ VP}$, so reduce again:

$$[\mathrm{S}] \qquad \langle\ \rangle$$

Shift-reduce parsing is

- bottom-up, working from the words to the root.
- left-to-right, since the words of the buffer are processed left-to-right.

At some steps the parser has a *choice* of operations and thus needs to search.

These decision points are referred to as **conflicts** and can be subcategorized as follows:

Shift-reduce conflicts: both shift and reduce are applicable and at least one reduce is also applicable. In the configuration

$$[\textit{Toby}] \qquad\qquad \langle \textit{drinks scotch} \rangle$$

either the word *"drinks"* could be shifted onto the stack or,

given that the top of the stack matches the right-hand side of the rule $\mathrm{NP} \rightarrow \textit{Toby}$, the *"Toby"* on the top of the stack can be reduced to NP.

Reduce-reduce conflicts: More than one reduction is applicable. In the configuration

$$[\mathrm{NP}\ \textit{drinks}] \qquad\qquad \langle \textit{scotch} \rangle$$

there are two applicable reductions,

one using $\mathrm{NP} \rightarrow$ *drinks* and

the other using $\mathrm{V} \rightarrow$ *drinks*.

(A shift of *"scotch"* is also possible so there is a shift-reduce conflict in this configuration too.)

Faced with a conflict, a simple shift-reduce parser will have first to try one of the applicable operations and eventually return and try the others: some may lead to parses, others may prove fruitless.

4.2.2 A Prolog implementation of shift-reduce parsing

Shift-reduce data structures How to represent the stack, the input buffer, the grammar and the lexicon.

- The stack will be represented as a list. Items in the list will be words and categories.

  The *first* item of the list will be the top of the stack, i.e. the top of the stack is the head of the list. `[scotch, v, np]`.

- The input buffer will be a list of the words of the input string.

- We will use BH-GRN as our grammar representation.

The shift operation Shift is an operation which has four arguments: it is given a current stack and a current input buffer and produces a new stack and a new input buffer.

A shift operation can only be successful if the current input buffer has at least one word on it (`[Word | Words]`). All that shift does is to make the first word of the input the head of the list representing the stack (this being its top):

```
shift(Stack, [Word | Stack], [Word | Words], Words).
```

Reduce is a four-place predicate that relates a current stack and input buffer to a new stack and input buffer.

Reduce finds a rule either in the grammar (`LHS ---> RHS`) or in the lexicon (`LHS ===> RHS`) whose body (`RHS`) when reversed (to `RRHS`) matches the top items of the stack.

```
reduce(Stack, [LHS | Rest], Input, Input) :-
    ((LHS ---> RHS) ;
     (LHS ===> RHS)),
    reverse(RHS, RRHS),
    append(RRHS, Rest, Stack).
```

```
reduce(Stack, [LHS | Rest], Input, Input) :-
    ((LHS ---> RHS) ;
     (LHS ===> RHS)),
    reverse(RHS, RRHS),
    append(RRHS, Rest, Stack).
```

This match is implemented by calling
`append(RRHS, Rest, Stack).`

The second argument is the bottom of the stack (`Rest`), and the first argument is not only the top of the stack but is also constrained to be equal to the (reverse of the) body of the rule or lexical entry (`RRHS`).

If this succeeds, the new stack (the second argument of `reduce`) has the left-hand side of the rule or lexical entry (`LHS`) as its top and the old stack's bottom (`Rest`) as the rest of the stack.

The input buffer is unchanged.

A more efficient version of reduce

```
reduce(Stack, NewStack, Input, Input) :-
    reduce_aux(Stack, [], NewStack).

reduce_aux(Rest, RHS, [LHS | Rest]) :-
    ((LHS ---> RHS) ;
     (LHS ===> RHS)).

reduce_aux([Cat | Cats], RevStack, NewStack) :-
    reduce_aux(Cats, [Cat | RevStack], NewStack.
```

The complete program

```
shift(Stack, [Word | Stack], [Word | Words], Words).

reduce(Stack, NewStack, Input, Input) :-
    reduce_aux(Stack, [], NewStack).

reduce_aux(Rest, RHS, [LHS | Rest]) :-
    ((LHS ---> RHS) ;
     (LHS ===> RHS)).

reduce_aux([Cat | Cats], RevStack, NewStack) :-
    reduce_aux(Cats, [Cat | RevStack], NewStack).
```

```
sr_parse([Stack], [Stack], Input, Input).

sr_parse(Stack, ResultStack, Input, ResultInput) :-
    shift(Stack, NewStack, Input, NewInput),
    sr_parse(NewStack, ResultStack, NewInput, ResultInput).

sr_parse(Stack, ResultStack, Input, ResultInput) :-
    reduce(Stack, NewStack, Input, NewInput),
    sr_parse(NewStack, ResultStack, NewInput, ResultInput).

parse(Input, Start) :-
    sr_parse([], [Start], Input, []).
```

Program 4.4: Shift-reduce parser

Typical user goals:

```
?- sr_parse([], [s], [toby, drinks, scotch], []).
```

```
?- parse([toby, drinks, scotch], X).
```

4.3 Compilers and interpreters

Compilers and interpreters are **metaprograms**; that is, they are programs that treat other programs as their data.

- An **interpreter** is a program that takes another program and executes it.
- A **compiler** by contrast does not execute programs. It is a program that takes in other programs and converts them into another form. We say that compilers take in source programs and translate them into object programs. Compilers are thus preprocessors for interpreters.

```
parent(henry_II, john).
parent(john, henry_III).
parent(isabella, henry_III).
parent(henry_III, edward_I).

grandparent(A, B) :-
     parent(A, C), parent(C, B).

greatgrandparent(D, E) :-
     grandparent(D, F), parent(F, E).
```

Program 4.5: Family relationships program

```
parent(henry_II, john).
parent(john, henry_III).
parent(isabella, henry_III).
parent(henry_III, edward_I).

grandparent(A, B) :-
     parent(A, C), parent(C, B).

greatgrandparent(D, E) :-
     parent(D, C), parent(C, F), parent(F, E).
```

Program 4.6: Another family relationships program

```
parent(henry_II, john).
parent(john, henry_III).
parent(isabella, henry_III).
parent(henry_III, edward_I).

grandparent(henry_II, henry_III).
grandparent(john, edward_I).
grandparent(isabella, edward_I).

greatgrandparent(henry_II, edward_I).
```

Program 4.7: A third family relationships program

In the logic programming literature this form of compilation, in which predictable steps are executed in a preprocessing step, is referred to as **partial execution** or **partial evaluation**. The computer science literature also makes reference to **unfolding** in this context, see e.g. Burstall and Darlington (1977). We will use the term 'partial execution' to subsume this.

We can apply these ideas to natural language parsing.

- We can think of a parser as an interpreter for a grammar.
- The parser program retrieves rules of the grammar and initiates actions on the basis of the rules retrieved,
- much as an interpreter, such as the Prolog interpreter, retrieves statements of a program and initiates actions on the basis of the statements retrieved.
- This gives us an example of the use of interpreters in computational linguistics.

Programs which convert grammars, i.e. convert from some ‘source’ grammar to some ‘object’ grammar

Converting to a new grammar formalism: The source grammar might be expressed in an elegant formalism, convenient for linguists but inconvenient for parsing. If the source grammar is weakly (or, even, strongly) equivalent to some CF-PSG, say, then we could convert the source grammar to that equivalent CF-PSG for possibly more efficient parsing.

Eliminating certain constructions: In the last chapter, we noted that programs exist for removing left- and right-recursion and ϵ-productions from some source grammars to give weakly equivalent grammars that are more amenable to simple parsing (although we also noted that this is a dispreferred solution as it gives rise to grammars that assign incorrect phrase-structures to the well-formed expressions). These programs are therefore also compilers of sorts.

Obtaining control information: In chapter **??** we present a parser in which grammar rule invocation is constrained during parsing with reference to an ‘oracle’ that contains control information. The ‘oracle’ is constructed from the grammar in a preprocessing step.

Producing a directly executable program: In the above examples, the compiler takes a grammar and produces a new grammar. This new grammar can still only be used in conjunction with a parser, acting as interpreter over the new grammar. Another possibility is to convert the source grammar into a *grammar-specific parsing program* that can be executed directly. Examples of this are given later.

Suppose we have

- a parser, written in Prolog, and
- a separate grammar to be used with this parser
- The parser is an interpreter for the grammar, and
- the Prolog interpreter is an interpreter for the parser.

There are therefore *two* interpretation processes going on:

- the Prolog interpreter is picking up and making use of the clauses in the Prolog parsing program, and
- these are picking up and making use of the rules of the grammar.

We could have a compiler which

- partially executes the parser on the grammar, i.e. it modifies the clauses of the parser (in much the same way as we showed with the `greatgrandparent` example) by making them refer directly to the rules of the grammar.
- There would no longer be a separate parser and grammar, just the single ‘expanded’ program.
- The program would thus be a parser with an in-built grammar.
- The disadvantage is that this program would be grammar-specific, but
- this may be offset by the efficiency gains of reducing the two interpretation processes above to a single interpretation, by the Prolog interpreter, of the ‘expanded’ program.

Disadvantages of compilation

- the compilation process can be very slow. Provided that it is a one-off process, this is not a problem, but in an environment in which grammars are regularly being changed it is problematic.
- the intention that lies behind compilation sometimes seems to be to 'absorb' the computational complexity of the parsing process into the compilation step, resulting in a tractable parsing step. This may not be an achievable objective. Grammars needed for computational linguistics can, in the worst case, assign a number of parses to a string that is exponentially related to the length of the string. No matter what compilation has taken place, finding all the parses will still take worst case exponential time.

- compilation often results in object grammars that are larger than the source grammars. If grammar size can dominate natural language parsing time, so increasing the size of the grammar may not be a good thing.
- it is not always the case that compilers exist or can work on all grammars of the class of source grammars they are supposed to work on (it is not unusual for some restriction to be placed on source grammars to guarantee termination of a compilation process, for example).

4.3.1 Compiling the shift-reduce parser

The explicit calls to `shift/4` and `reduce/4` can be 'expanded' into the definition of `sr_parse/4` itself.

Reduce, using $\mathrm{S} \rightarrow \mathrm{NP\ VP}$**:**

```
sr_parse([vp, np | Cats], RS, I, RI) :-
   sr_parse([s | Cats], RS, I, RI).
```

Shift of `toby` **and simultaneous reduction of this word using** $\mathrm{NP} \rightarrow$ ***Toby***:

```
sr_parse(Cats, RS, [toby|X], RI) :-
   sr_parse([np | Cats], RS, X, RI).
```

```
sr_parse([Stack], [Stack], Input, Input).

sr_parse([vp, np | Cats], RS, I, RI) :-
    sr_parse([s | Cats], RS, I, RI).

sr_parse([np, v | Cats], RS, I, RI) :-
    sr_parse([vp | Cats], RS, I, RI).

sr_parse([v | Cats], RS, I, RI) :-
    sr_parse([vp | Cats], RS, I, RI).
```

```
sr_parse(Cats, RS, [toby|X], RI) :-
    sr_parse([np | Cats], RS, X, RI).

sr_parse(Cats, RS, [scotch|X], RI) :-
    sr_parse([np | Cats], RS, X, RI).

sr_parse(Cats, RS, [drinks|X], RI) :-
    sr_parse([np | Cats], RS, X, RI).

sr_parse(Cats, RS, [drinks|X], RI) :-
    sr_parse([v | Cats], RS, X, RI).

parse(Input, Start) :-
    sr_parse([], [Start], Input, []).
```

Program 4.8: Compiled version of the shift-reduce parser

- In executing the original program, there would have been *two* interpretation processes:
  1. Prolog interpreting the parser, and
  2. the parser interpreting the grammar.
- With program 4.8, there will be only the one interpretation process:
  1. Prolog interpretation of the grammar-specific parser.

4.3.2 Compiling a top-down interpreter

Whenever `td_parse/3` succeeds with a call to a rule from the lexicon, the result is like this:

```
td_parse(np, [toby  drinks, scotch], [drinks,scotch]]).
```

or like this:

```
td_parse(v,[drinks,scotch],[scotch]).
```

Each of these clauses has the form

```
td_parse(Category,[Word | Words], Words).
```

This means that for each lexical entry in the grammar, such as `np ===> [toby]`, the first clause of `td_parse/3` can be replaced by a clause such as:

```
td_parse(np, [toby | Words], Words]).
```

which has the same effect but obviates the separate lexical lookup.

For grammar rules, we have something like the following:

```
1  1  Call: td_parse(s,[toby,drinks,scotch],[]) ?
2  2  Call: s--->_371 ?
2  2  Exit: s--->[np,vp] ?
3  2  Call: td_parse_dtrs([np,vp],[toby,drinks,scotch],[
```

Eliminating the reference to a specific list of words to be parsed and the lookup of the grammar rule gives us the partially executed form:

```
td_parse(s, Input, RemainingInput):-
      td_parse_dtrs([np,vp],Input, RemainingInput).
```

This means that for each rule of the grammar of the form `Mother ---> Daughters`, the second clause of `td_parse/3` can be replaced by a clause of the form:

```
td_parse(Mother, Input, RemainingInput):-
      td_parse_dtrs(Daughters,Input, RemainingInput).
```

```
td_parse(np, [toby | Words], Words).
td_parse(np, [scotch | Words], Words).
td_parse(np, [drinks | Words], Words).
td_parse(v, [drinks | Words], Words).

td_parse(s, Input, RemainingInput):-
      td_parse_dtrs([np, vp] ,Input, RemainingInput).
td_parse(vp, Input, RemainingInput):-
      td_parse_dtrs([v],Input, RemainingInput).
td_parse(vp, Input, RemainingInput):-
      td_parse_dtrs([v, np],Input, RemainingInput).
```

Program 4.9: Partial execution of program 4.1 over grammar 4.2.

But, even more steps are predictable and can be carried out now rather than at execution time. For a clause such as:

```
td_parse(s, Input, RemainingInput):-
        td_parse_dtrs([np, vp] ,Input, RemainingInput).
```

we know what `td_parse_dtrs/3` will try to do. It will try to parse some prefix of the input string as an `np` using `td_parse/3`, and the rest of the string as the remaining 'goal' `[vp]` using `td_parse_dtrs/3`. But this call of `td_parse_dtrs/3` on 'goals' `[vp]` will invoke another call to `td_parse/3` on `vp` and a call to `td_parse_dtrs/3` on `[ ]`, which does nothing. So we can replace the above clause by:

```
td_parse(s, Input, ResultInput):-
        td_parse(np, Input, RemainingInput),
        td_parse(vp, RemainingInput, ResultInput).
```

Doing this throughout program 4.9 gives the following program:

```
td_parse(np, [toby | Words], Words).
td_parse(np, [scotch | Words], Words).
td_parse(np, [drinks | Words], Words).
td_parse(v, [drinks | Words], Words).

td_parse(s, Input, ResultInput):-
        td_parse(np, Input, RemainingInput),
        td_parse(vp, RemainingInput, ResultInput).
td_parse(vp, Input, ResultInput):-
        td_parse(v, Input, ResultInput).
td_parse(s, Input, ResultInput):-
        td_parse(np, Input, RemainingInput),
        td_parse(vp, RemainingInput, ResultInput).
```

Program 4.10: Partial execution of program 4.9

Further optimization: We can remove the call to `td_parse/3` altogether, by, for example, changing

```
td_parse(s, Input, ResultInput):-
        td_parse(np, Input, RemainingInput),
        td_parse(vp, RemainingInput, ResultInput).
```

into

```
s(Input, ResultInput):-
        np(Input, RemainingInput),
        vp(RemainingInput, ResultInput).
```

Replace calls to `td_parse/3` by calls to predicates corresponding directly to the categories used in the rules of the grammar.

Lexical rules:

replace

```
td_parse(np, [toby | Words], Words).
```

by

```
np([toby | Words], Words).
```

```
np([toby | Words], Words).
np([scotch | Words], Words).
np([drinks | Words], Words).
v([drinks | Words], Words).

s(Input, ResultInput):-
            np(Input, RemainingInput),
            vp(RemainingInput, ResultInput).
vp(Input, ResultInput):-
            v(Input, ResultInput).
vp(Input, ResultInput):-
            v(Input, RemainingInput),
            np(RemainingInput, ResultInput).
```

Program 4.11: A direct Prolog implementation of the parser in program 4.10

5 Definite Clause Grammars

It is possible to characterize the meaning of a (pure) logic program in two ways: by giving its

1. **declarative semantics** or by giving its
2. **procedural semantics** relative to some interpreter.

In describing parsing in the previous chapter, we strongly emphasized the procedural semantics of the programs relative to the Prolog interpreter, i.e. we have emphasized *how* Prolog uses its predicate definitions to parse a given input string.

We now consider the declarative semantics of programs.

Prolog programs are a notational variant of a restricted version of first-order logic, known as **definite clause logic**. Definite clauses are formulae of logic having the following form:

$$\alpha_1 \wedge \ldots \wedge \alpha_n \supset \alpha_{n+1}$$

where each α_i is a positive literal (i.e. a predicate applied to some arguments).

Definite clauses comprise a conjunction of n $(n \geq 0)$ positive literals as the antecedent of any material implication whose consequent is a single positive literal.

Any variables in the literals are taken to be universally quantified over, but limited to, the whole clause.

Prolog facts and rules are obviously just a notational variant of these definite clauses.

Specifically, the head of a Prolog rule corresponding to the implication above will be the consequent of the implication (α_{n+1}) and will be written to the left of the `:-` symbol;

the body of the rule will be the antecedent of the implication ($\alpha_1 \wedge \ldots \wedge \alpha_n$) and will be written to the right of the `:-` symbol, with the conjunction symbols ($\wedge$) replaced by commas and with a full stop terminating the rule.

(There are other notational variations: e.g. a common convention when writing formulae of first order logic is to use uppercase letters to begin predicate symbols, function symbols and constant symbols, and lowercase letters to begin variables, while this convention is reversed when writing Prolog.)

what we did in the preceding chapter was merely to translate a grammar expressed in one notation, that of CF-PSGs, into one expressed in another notation, that of definite clause logic. A CF-PSG rule such as

$$\mathrm{S} \rightarrow \mathrm{NP\ VP}$$

is translated into a definite clause

$$\mathrm{NP}(x_0, x_1) \wedge \mathrm{VP}(x_1, x) \supset \mathrm{S}(x_0, x)$$

or, taking into account Prolog's syntactic rules, it is translated into the Prolog rule

```
s(S0, S):- np(S0, S1), vp(S1, S).
```

Grammars expressed as definite clauses are referred to as **Definite Clause Grammars** (DCGs) and are one type of so-called **Logic Grammar**.

This perspective gives us a declarative semantics for our grammar.

The definite clause $\mathrm{NP}(x_0, x_1) \wedge \mathrm{VP}(x_1, x) \supset \mathrm{S}(x_0, x)$ has a declarative semantics which we can paraphrase informally into English as saying

'list x_0 less list x is an S if list x_0 less list x_1 is an NP and list x_1 less list x is a VP'.

Lists might feel like a substantial innovation, bringing a new mechanism into logic. But this is not so. Lists are simply (nested) compound terms. Nothing need be added to the logical formalism.

For example, using the binary function symbol '.' and the constant [ ], we can construct list-like objects as follows:

[ ]	empty list
.(a, [ ])	list containing just 'a'
.(a, .(b, [ ]))	list containing 'a' and 'b'
.(a, .(b, .(c, [ ])))	list containing 'a', 'b' and 'c'
.(a, x)	list which starts with 'a' and continues with other unspecified material

We allow ourselves some ‘syntactic sugar’ to improve the presentation. In place of these ugly compound terms (formed using ‘.’ and ‘[ ]’), we could allow ourselves to write the following:

[ ]	empty list
[a]	list containing just ‘a’
[a, b]	list containing ‘a’ and ‘b’
[a, b, c]	list containing ‘a’, ‘b’ and ‘c’
[a, $\mid x$]	list which starts with ‘a’ and continues with other unspecified material

These are merely a neat notation for the corresponding compound terms.

```
 ?- display([a,b,c]).
.(a, .(b, .(c, [])))

Yes
```

From this point of view, program 4.11 is not so much a parser as just another way of presenting grammars: using logic.

It is only when we imbue it with a procedural semantics relative to some interpreter that it becomes a parsing program

In that case the *type* of parsing program that it becomes depends upon the interpreter we use.

If we use Prolog, then, because Prolog works top-down from left-to-right doing depth-first search, program 4.11 becomes a top-down, left-to-right, depth-first parser.

If we had substituted some other interpreter for definite clause logic programs, one that had a different execution strategy, we would get some other type of parser.

For example, a definite clause logic program interpreter that worked bottom-up, from facts through antecedents of rules to consequents of rules, would give a bottom-up parser.

Definite Clause Grammars (DCGs), which are simply logical definitions, generalise CF-PSGs by allowing the nonterminals of the grammar (which are simply predicates of the logic) to have arguments (enclosed in parentheses), as normal in logic.

The grammars that we have used in the preceding chapters have been very unsophisticated. If we were to try to extend them to correctly handle the number agreement:

(44) *"This knight"*

(45) *"These knights"*

simply by adding the following grammar rule and lexical entries

$$\text{NP} \rightarrow \text{Det N} \qquad \textit{this} : \text{Det} \qquad \textit{these} : \text{Det}$$

$$\textit{knight} : \text{N} \qquad \textit{knights} : \text{N}$$

the resulting grammar will overgenerate, allowing not only ‘this knight’ and ‘these knights’, but also

(46) *“*This knights”*

(47) *“*These knight”*

The problem here is that a particular form of the determiner can only co-occur with a particular form of the noun: singular with singular (*“this knight”*) and plural with plural (*“these knights”*).

A solution to the problem is to replace the simple CF-PSG by a DCG, in which the **number** of a noun or determiner can be stated as an argument of the category names N and Det, as in the following lexical entries:

$$\textit{this} : \mathrm{Det(SING)} \qquad \textit{knight} : \mathrm{N(SING)}$$

$$\textit{these} : \mathrm{Det(PLU)} \qquad \textit{knights} : \mathrm{N(PLU)}$$

showing *"this"* and *"knight"* to be singular (SING), and *"these"* and *"knights"* to be plural (PLU).

The grammar should be able to handle other agreement phenomena, such as the following

(48) *"I am"*

(49) *"We are"*

(50) *"She is"*

(51) *"They are"*

in which we find not only singular/plural agreement again (this time between a pronoun and a verb), but also another pattern of agreement between singular *"I"* and *"am"* on the one hand and singular *"she"* and *"is"* on the other. The difference between *"I"* and *"she"* is traditionally labelled **person**, with *"I"* being classified as *first person* and *"she"* as *third person*.

Since nonterminals in a DCG can have several arguments, as, for example, we can specify both the *person* and *number* of pronouns:

$$\textit{I} : \text{Pro}(1, \text{SING}) \qquad \textit{she} : \text{Pro}(3, \text{SING})$$

$$\textit{we} : \text{Pro}(1, \text{PLU}) \qquad \textit{they} : \text{Pro}(3, \text{PLU})$$

showing, e.g., *"she"* to be a third person singular pronoun (Pro(3, SING)).

In this formalism, therefore, category names are no longer atomic.

They comprise what we will call a **backbone category** (Det, N, Pro, etc.) and zero, one or more **arguments**.

One immediate advantage of this change is that it allows us to continue to represent the fact that *“knight”* and *“knights”* are both nouns by virtue of the fact that they belong to the same backbone category, N, while at the same time allowing us to distinguish them from one another as N(SING) and N(PLU),

By allowing argument positions to be occupied by *variables* we gain economy in the specification of lexical entries for certain words. (We will use w, ws, x, xs, y, ys, z and zs, possibly subscripted, as variables.)

The use of variables allows us to leave a category unspecified with respect to a certain argument, so that, for words such as the determiner *"the"*, which, unlike *"this"* and *"these"*, can co-occur with either a singular or a plural noun (*"the knight"*, *"the knights"*), or a noun like *"sheep"*, which, unlike *"knight"* and *"knights"*, can co-occur with either a singular or plural determiner (*"this sheep"*, *"these sheep"*), only one lexical category is needed:

$$\textit{you} : \mathrm{Pro}(2, x)$$

$$\textit{the} : \mathrm{Det}(x)$$

$$\textit{sheep} : \mathrm{N}(x)$$

Nonterminals in *rules* will also have arguments, and these may be constants (e.g. 1, 2, 3, SING, PLU) or variables (e.g. $x, x_0, x_1, xs, xs_0, xs_1, y, y_0, y_1$), as in the following examples:

$$\text{NP}(x, y) \rightarrow \text{Pro}(x, y)$$

$$\text{NP}(3, x) \rightarrow \text{Det}(x)\ \text{N}(x)$$

Multiple occurrences of a variable within a rule must all match to the same single expression.

The first of these rules therefore specifies that the person (first argument) and number (second argument) of a noun phrase whose sole constituent is a pronoun are the same as the person and number of that pronoun. It is the multiple occurrences of x and y in the rule that constrain the person and number of the NP to be the same as the person and number of the Pro.

The second rule specifies the constraint that the number of an NP is the same as that of the Det which it immediately dominates, and the number of the NP and the Det must also be the same as the number of the N which the NP immediately dominates. Intuitively then, this rule ought to license the string *"this knight"*, in which the number of the Det and the N are both SING, but ought to reject the string **"this knights"* in which the number of the Det and N differ, which would require the variable x to match with more than one constant. The second rule also stipulates that an NP that comprises a Det and an N is always a third person NP.

Using these ideas, we can formulate a simple DCG (grammar 5.1) which incorporates the ideas we have just been discussing.

$$\begin{aligned}
\text{S} &\rightarrow \text{NP}(x, y)\ \text{VP}(x, y) \\
\text{NP}(x, y) &\rightarrow \text{Pro}(x, y) \\
\text{NP}(3, x) &\rightarrow \text{Det}(x)\ \text{Nbar}(x) \\
\text{Nbar}(x) &\rightarrow \text{N}(x) \\
\text{Nbar}(x) &\rightarrow \text{Adj}\ \text{Nbar}(x) \\
\text{VP}(x, y) &\rightarrow \text{Vi}(x, y) \\
\text{VP}(x, y) &\rightarrow \text{Vbe}(x, y)\ \text{Adj}
\end{aligned}$$

$$\begin{aligned}
\textit{this} &: \text{Det(SING)} \\
\textit{these} &: \text{Det(PLU)} \\
\textit{the} &: \text{Det}(x) \\
\textit{knight} &: \text{N(SING)} \\
\textit{knights} &: \text{N(PLU)} \\
\textit{sheep} &: \text{N}(x) \\
\textit{slept} &: \text{Vi}(x, y) \\
\textit{drink} &: \text{Vi}(1, x), \\
&\quad \text{Vi}(2, x), \\
&\quad \text{Vi}(3, \text{PLU}) \\
\textit{drinks} &: \text{Vi}(3, \text{SING})
\end{aligned}$$

$$\begin{aligned}
\textit{I} &: \text{Pro}(1, \text{SING}) \\
\textit{we} &: \text{Pro}(1, \text{PLU}) \\
\textit{you} &: \text{Pro}(2, x) \\
\textit{it} &: \text{Pro}(3, \text{SING}) \\
\textit{they} &: \text{Pro}(3, \text{PLU}) \\
\textit{am} &: \text{Vbe}(1, \text{SING}) \\
\textit{are} &: \text{Vbe}(1, \text{PLU}), \\
&\quad \text{Vbe}(2, x), \\
&\quad \text{Vbe}(3, \text{PLU}) \\
\textit{is} &: \text{Vbe}(3, \text{SING}) \\
\textit{tipsy} &: \text{Adj}
\end{aligned}$$

Grammar 5.1 A simple DCG incorporating number and person agreement

It should be clear that we get a direct Prolog implementation of a top-down DCG parser, just as we got a direct implementation of a top-down CF-PSG parser.

We simply take the DCG, adapt it to Prolog syntax, and add list processing arguments. A DCG grammar rule such as

$$\mathrm{NP}(3, x) \rightarrow \mathrm{Det}(x), \mathrm{N}(x)$$

for, example, becomes

```
np(3, X, L0, L) :- det(X, L0, L1), n(X, L1, L).
```

Backbone categories such as NP and VP, function symbols such as PER and NUM,

and constants such as SING and PLU must now start with lower case letters, and variables must start with upper case letters.

The arrow $\rightarrow$ is replaced by Prolog 'if', i.e. `:-`.

Categories on the right-hand sides of Prolog rules are separated by commas, and the rules are terminated with a full stop.

Two extra arguments (`L0, L1` and `L` in the above) are added to pass the input string around.

Difference lists

An alternative to the standard way of representing a list is to regard it as the difference between a pair of lists. For example, the list

```
[a, b, c]
```

can be represented in any of the following ways

```
[a, b, c] - []
```

```
[a, b, c, d] - [d]
```

```
[a, b, c, d, e] - [d, e]
```

```
[a, b, c | X] - X
```

Such representations are termed **difference lists** and the intended interpretation is that the list `[a, b, c]` is the difference between each of the pairs of lists above; i.e.

`[a, b, c]` with `[ ]` taken away is

`[a, b, c]`;

`[a, b, c, d]` with `[d]` taken away is

`[a, b, c]`;

`[a, b, c, d, e]` with `[d, e]` taken away is again

`[a, b, c]`; and so on.

The last version above (`[a, b, c | X] - X`) is the called **most general difference list** encoding of the list `[a, b, c]`.

The empty list is the difference between a list and itself:

```
[a, b, c] - [a, b, c]

[a, b, c, d] - [a, b, c, d]

[a, b, c, d, e] - [a, b, c, d, e]

[ ] - [ ]
```

In its most general difference list form, the empty list is simply two identical variables – any list subtracted from itself:

```
X - X
```

We employ difference lists transparently in our encoding of lexical categories, writing the two parts of the list as the last two arguments to a predicate.

Lexical entries such as

$$\textit{knight} : \mathrm{N}(\mathrm{SING})$$
$$\textit{sheep} : \mathrm{N}(x)$$
$$\textit{the} : \mathrm{Det}(x)$$

become the following:

```
n(sing, [knight| X], X).
n(X,    [sheep | X], X).
det(X,  [the   | X], X).
```

where the pair `[knight|X], X` is the difference between any list and the list beginning with the word *"knight"*.

Not so transparent is the fact that the not only the lexicon but also the grammar rules in our DCGs employ difference lists.

5.3 Parsing DCGs

```
s(L0, L) :- np(X, Y, L0, L1), vp(X,
     Y, L1, L).
np(X, Y, L0, L) :- pro(X, Y, L0, L).
np(3, X, L0, L) :- det(X, L0, L1), n
     (X, L1, L).
vp(X, Y, L0, L) :- vi(X, Y, L0, L).
vp(X, Y, L0, L) :- vbe(X, Y, L0, L1)
     , adj(L1, L).
```

```
pro(1, sing, [i|L], L).
pro(1, plu,  [we|L], L).
pro(2, X,    [you|L], L).
pro(3, sing, [it|L], L).
pro(3, plu,  [they|L], L).

adj([tipsy|L], L).
```

```
det(sing, [this|L], L).
det(plu,  [these|L], L).
det(X,    [the|L], L).

n(sing, [knight|L], L).
n(plu,  [knights|L], L).
n(X,    [sheep|L], L).
```

```
vbe(1, sing, [am|L], L).
vbe(1, plu,  [are|L], L).
vbe(2, X,    [are|L], L).
vbe(3, plu,  [are|L], L).
vbe(3, sing, [is|L], L).

vi(X, Y,    [slept|L], L).
vi(1, X,    [sleep|L], L).
vi(2, X,    [sleep|L], L).
vi(3, plu,  [sleep|L], L).
vi(3, sing, [sleeps|L], L).
```

Program 5.2 Top-down DCG parser for grammar 5.1 in Prolog

Building trees We showed how to turn a recognizer into a parser by building a phrase structure tree for the string being parsed. That involved modifying the program.

DCGs offer an alternative way of building a parse tree.

Let us suppose we represent trees as Prolog terms, with the function symbol corresponding to the label of the parent node and the arguments corresponding to the children:

```
VP(V(drinks), NP(scotch))
```

If we take grammar 4.1 as our basis, we can start by modifying the lexical entries so that they carry an argument which represents their tree structure, as follows:

$$\textit{Toby} : \mathrm{NP}(\mathrm{NP}(\mathrm{Toby}))$$

$$\textit{drinks} : \mathrm{V}(\mathrm{V}(\mathrm{drinks}))$$

The categories in the grammar rules also require an argument to hold the tree representation. The S expansion rule

$$\mathrm{S} \rightarrow \mathrm{NP\ VP}$$

becomes the following DCG rule:

$$\mathrm{S(S(x,\ y))} \rightarrow \mathrm{NP(x)\ VP(y)}$$

The complete grammar:

$$\mathrm{S(S(x,\ y))} \rightarrow \mathrm{NP(x)\ VP(y)} \qquad \textit{Toby} : \mathrm{NP(NP(Toby))}$$

$$\mathrm{VP(VP(x))} \rightarrow \mathrm{V(x)} \qquad \textit{scotch} : \mathrm{NP(NP(scotch))}$$

$$\mathrm{VP(VP(x,\ y))} \rightarrow \mathrm{V(x)\ NP(y)} \qquad \textit{drinks} : \mathrm{NP(NP(drinks)),\ V(V(drinks))}$$

Grammar 5.2: Simple DCG with tree-building arguments

```
s(s(NP, VP), L0, L):-  np(NP, L0, L1), vp(VP, L1, L).

vp(vp(V), L0, L):-  v(V, L0, L).

vp(vp(V, NP), L0, L):- v(V, L0, L1), np(NP, L1, L).

np(np(toby), [toby | X], X).

np(np(scotch), [scotch | X], X).

np(np(drinks), [drinks | X], X).

v(v(drinks), [drinks | X], X).
```

Program 5.2a: Prolog implementation of Grammar 5.2

Grammar rule notation

All the predicates in program 5.2 contained two arguments for passing around lists and ‘remainders’ of lists.

Since any Prolog parser of this kind is necessarily going to include these two arguments, is it not unreasonable to relieve the programmer of the responsibility for adding them.

We would like a program that takes as its input a file containing something resembling a DCG, without these two arguments, and produces as its output a Prolog parser that has these extra two arguments added throughout.

Most Prolog systems come with just such a program in-built. The input to the program is a grammar typed into a file using a notation called **Grammar Rule Notation** (GRN), which is very similar to that used for CF-PSGs.

However, we henceforth refer to this as Prolog-GRN in order to distinguish it from BH-GRN.

Prolog-GRN uses an infix operator `-->` (two minus signs and a greater-than sign) instead of the right arrow ($\rightarrow$) used in CF-PSGs.

It separates items in the right-hand side of a grammar rule by commas, and terminates the rule with a full stop.

```
s --> np(X, Y), vp(X, Y).
```

Lexical items have a slightly different syntax; they comprise the lexical category, the operator `-->`, and the word or words enclosed in square brackets.

```
n(sing) --> [knight].
```

When we type the DCG into a file using Prolog-GRN, and then consult the file into Prolog, it is automatically converted by the in-built preprocessor program into a top-down Prolog parser like that in program 5.2, the extra list processing arguments being automatically added as the *final* two arguments of every predicate.

```
s --> np(Per, Num), vp(Per, Num).
np(Per, Num) --> pro(Per, Num).
np(3, Num) --> det(Num), nbar(Num).
nbar(Num) --> n(Num).
nbar(Num) --> adj, n(Num).
vp(Per, Num) --> vi(Per, Num).
vp(Per, Num) --> vbe(Per, Num), adj.
```

```
pro(1, sing) --> [i].
pro(1, plu) --> [we].
pro(2, _Num) --> [you].
pro(3, sing) --> [it].
pro(3, plu) --> [they].
adj --> [tipsy].
```

```
det(sing) --> [this].
det(plu) --> [these].
det(_Num) --> [the].

n(sing) --> [knight].
n(plu) --> [knights].
n(_Num) --> [sheep].

vbe(1, sing) --> [am].
vbe(1, plu) --> [are].
vbe(2, _Num) --> [are].
vbe(3, plu) --> [are].
vbe(3, sing) --> [is].

vi(_Per, _Num) --> [slept].
vi(1, _Num) --> [sleep].
vi(2, _Num) --> [sleep].
vi(3, plu) --> [sleep].
vi(3, sing) --> [sleeps].
```

Grammar 5.6: Prolog-GRN version of program 5.2

If you load Grammar 5.6 and try to find out if the predicate `-->/2` is defined, this is what happens:

```
?- listing(-->).
Correct to: *->? n
ERROR: No predicates for `-->'

No
?- listing(s).


s(A, B) :-
        np(C, D, A, E),
        vp(C, D, E, B).

Yes
  ?-
```

Equally, we can write DCGs in BH-GRN. Such DCGs can be used directly with both the top-down and bottom-up parsers discussed earlier.

It is also possible to compile a BH-GRN into a Prolog parser in exactly the same manner that Prolog-GRN is compiled.

In appendix C, there is a Prolog program (program C.1) for translating a BH-GRN grammar into a Prolog parser like that in program 5.1.

We do not make any effort to explain how this program works, as this need not be of concern for the purposes of this book. Pointers to the relevant literature, where some explanation can be found, are given at the end of the chapter.

If we

1. put the BH-GRN grammar 5.7 into a file,
2. invoke Prolog,
3. execute the operator definitions,
4. consult program C.1 into the Prolog memory,
5. and then consult the BH-GRN grammar,

a parser, logically equivalent to program 5.2, will be asserted into the Prolog memory.

Warning! When using BH-GRN, you should be very careful to avoid making typing mistakes. In particular, if you mistakenly type only two minus signs instead of three, then your erroneous rule will be caught by the in-built Prolog-GRN preprocessor. This will give erroneous results for the uses to which we put BH-GRN.

```
s ---> [np(Per, Y), vp(Per, Y)].
np(Per, Y) ---> [pro(Per, Y)].
np(3, Num) ---> [det(Num), nbar(Num)].

nbar(Num) ---> [n(Num)].
nbar(Num) ---> [adj, nbar(Num)].

vp(Per, Y) ---> [vi(Per, Y)].
vp(Per, Y) ---> [vbe(Per, Y), adj].


det(sing) ===> [this].
det(plu) ===> [these].
det(_Num) ===> [the].

n(sing) ===> [knight].
n(plu) ===> [knights].
n(_Num) ===> [sheep].
```

```
pro(1, sing) ===> [i].
pro(1, plu) ===> [we].
pro(2, _Num) ===> [you].
pro(3, sing) ===> [it].
pro(3, plu) ===> [they].

adj ===> [tipsy].

vbe(1, sing) ===> [am].
vbe(1, plu) ===> [are].
vbe(2, _Num) ===> [are].
vbe(3, plu) ===> [are].
vbe(3, sing) ===> [is].

vi(_Per, _Num) ===> [slept].
vi(1, _Num) ===> [sleep].
vi(2, _Num) ===> [sleep].
vi(3, plu) ===> [sleep].
vi(3, sing) ===> [sleeps].
```

Grammar 5.7: BH-GRN version of program 5.2

5.3.5 Embedding arbitrary Prolog goals into GRN

In program 4.3 for a depth-bounded top-down parser we showed one way of enabling a left-to-right, top-down parser to handle left-recursive rules.

In that program, it was the *parser* that kept track of the depth of the trees being built.

Now that we are translating straight from a grammar to a definite clause logic program, how do we implement such non-grammatical information?

Firstly, we observe that

$$\mathrm{NP}(x_0, x_1) \wedge \mathrm{VP}(x_1, x) \supset \mathrm{S}(x_0, x)$$

is the general form of a definite clause corresponding to a CF-PSG rule.

Since logical predicates are not in principle restricted to any specific number of arguments, we can generalize this to allow more than two arguments to a predicate.

We can exploit this extension to write a DCG version of the depth-bounded parser.

We start by using an additional argument to carry the depth information, moving this information from the interpreter (in the original version) into the grammar itself, and add the clauses from the interpreter which check and decrement the depth level.

```
s(Depth, S0, S) :-
     Depth > 0,
     NewDepth is Depth - 1,
     np(NewDepth, S0, S1),
     vp(NewDepth, S1, S).
```

The lexical entries simply require the extra argument:

```
np(Depth, [toby|L], L).
```

One problem with this is that it takes a lot of typing! At the very least, we would like the convenience of not having to type in the two list processing arguments, as in GRN. But, when a BH-GRN rule (or its Prolog-GRN equivalent) such as the following

```
s(Depth) ---> [Depth > 0, NewDepth is Depth-1,
                          np(NewDepth), vp(NewDepth)].
```

is consulted into Prolog, program C.1 (or, for Prolog-GRN, the in-built preprocessor) will convert the rule to:

```
s(Depth, L0, L) :- >(Depth, 0, L0, L1),
                   is(NewDepth, Depth-1, L1, L2),
                   np(NewDepth, L2, L3), vp(NewDepth, L3, L).
```

This is not what we want because we now see that what we had intended to be calls to the evaluable *binary* predicates `>` and `is` have been given extra arguments too. What is needed is a way of inhibiting the preprocessor from inserting these extra arguments.

Both program C.1 and the Prolog-GRN preprocessor come with a solution to this. We can tell either preprocessor *not* to add these arguments to certain predicates by enclosing those predicates in curly braces ('{' and '}'). For example:

```
s(Depth) ---> {Depth > 0, NewDepth is Depth-1},
                        [np(NewDepth), vp(NewDepth)].
```

converts to

```
s(Depth, L0, L) :- Depth > 0, NewDepth is Depth-1,
              np(NewDepth, L0, L1), vp(NewDepth, L1, L).
```

which is correct.

```
s(Depth)  ---> {Depth > 0, NewDepth is Depth - 1},
               [np(NewDepth), vp(NewDepth)].
vp(Depth) ---> {Depth > 0, NewDepth is Depth - 1},
               [v(NewDepth)].
vp(Depth) ---> {Depth > 0, NewDepth is Depth - 1},
               [v(NewDepth), np(NewDepth)].
vp(Depth) ---> {Depth > 0, NewDepth is Depth - 1},
               [vp(NewDepth), pp(NewDepth)].
pp(Depth) ---> {Depth > 0, NewDepth is Depth - 1},
               [p(NewDepth), np(NewDepth)].
np(Depth) ---> {Depth > 0, NewDepth is Depth - 1},
               [np(NewDepth), pp(NewDepth)].

np(_) ===> [toby].
np(_) ===> [scotch].
np(_) ===> [drinks].
np(_) ===> [ice].
v(_) ===> [drinks].
p(_) ===> [on].
```

1. Load Grammar 5.7 (the BH-GRN version)
2. Type `?-listing(--->)` to confirm that Prolog has loaded the grammar rules
3. Load the top-down recognizer, program4.1
4. Confirm that the top-down parser and grammar work together by parsing 'These tipsy knights sleep'

   ```
   ?- td_parse(T, [these, knights, sleep], []).
   ```

5. Load the bottom-up shift-reduce parser, program4.4.
6. Type `?-listing(--->)` to confirm that Prolog has loaded the grammar rules
7. Confirm that the bottom-up parser and grammar work together by parsing 'These tipsy knights sleep'

   ```
   ?- parse([these, tipsy, knights, sleep], X).
   ```

8. Load programC.1
9. Load Grammar 5.7 again
10. Type `?-listing(--->)` to confirm that Prolog no longer knows the grammar rules
11. Check that the grammar has been compiled into a DCG by parsing 'These tipsy knights sleep'

    ```
    ?- s([these, tipsy, knights, sleep], []).
    ```

12. Take the depth-bounded DCG, program 5.8 and modify it so that it also builds parse trees.

```
:-op(1100, xfx, --->).
:-op(1100, xfx, ===>).


s ---> [np(Per, Num, nom), vp(Per, Num)].

np(Per, Num, Case) ---> [pro(Per, Num, Case)].

np(3, Num, _Case) ---> [det(Num), nbar(Num)].
```

```
nbar(Num) ---> [n(Num)].
nbar(Num) ---> [ap, n(Num)].

ap ---> [adj].
```

```
vp(Per, Num) ---> [vi(Per, Num)].
vp(Per, Num) ---> [vt(Per, Num), np(_Per, _Num, acc)].
vp(Per, Num) ---> [vbe(Per, Num), adj].

det(sing) ===> [this].
det(plu) ===> [these].
det(_Per) ===> [the].

n(sing) ===> [knight].
n(plu) ===> [knights].
n(_Per) ===> [sheep].
```

```
pro(1, sing, nom) ===> [i].
pro(1, sing, acc) ===> [me].
pro(1, plu, nom) ===> [we].
pro(1, plu, acc) ===> [us].
pro(2, _Per, _Case) ===> [you].
pro(3, sing, _Case) ===> [it].
pro(3, sing, nom) ===> [she].
pro(3, sing, acc) ===> [her].
pro(3, sing, nom) ===> [he].
pro(3, sing, acc) ===> [him].
pro(3, plu, nom) ===> [they].
pro(3, plu, acc) ===> [them].
```

```
adj ===> [tipsy].

vbe(1, sing) ===> [am].
vbe(1, plu) ===> [are].
vbe(2, _Per) ===> [are].
vbe(3, plu) ===> [are].
vbe(3, sing) ===> [is].

vi(_Per, _Num) ===> [slept].
vi(1, _Per) ===> [sleep].
vi(2, _Per) ===> [sleep].
vi(3, plu) ===> [sleep].
vi(3, sing) ===> [sleeps].
```

```
vt(_Per, _Num) ===> [helped].
vt(1, _Per) ===> [help].
vt(2, _Per) ===> [help].
vt(3, plu) ===> [help].
vt(3, sing) ===> [helps].
```

```
?- s(I, []).

I = [i, slept] ;

I = [i, sleep] ;

I = [i, helped, me] ;

I = [i, helped, us] ;

I = [i, helped, you] ;

I = [i, helped, it] ;

I = [i, helped, her] ;

I = [i, helped, him] ;
```

```
[the, tipsy, sheep, helps, these, tipsy, knights]
[the, tipsy, sheep, helps, these, tipsy, sheep]
[the, tipsy, sheep, helps, the, knight]
[the, tipsy, sheep, helps, the, knights]
[the, tipsy, sheep, helps, the, sheep]
[the, tipsy, sheep, helps, the, tipsy, knight]
[the, tipsy, sheep, helps, the, tipsy, knights]
[the, tipsy, sheep, helps, the, tipsy, sheep]
[the, tipsy, sheep, are, tipsy]
[the, tipsy, sheep, is, tipsy]
```

6 Semantics

6.7 From English to Logic

English	Category	FOL Translation	Type	Denotation
"Duncan"	NP	d	**e**	Individual
"Macbeth"	NP	m	**e**	Individual
"died"	Vi	$\lambda x[died1(x)]$	$\langle \mathbf{e}, \mathbf{t} \rangle$	characteristic function (of the set of individuals that died)
"Duncan died"	S	$died1(d)$	**t**	truth value

Table 3: Some correspondences between English and FOL

Logic expression	Example	Prolog representation	Example
Individual constant	d	Prolog constant	`d`
Predicate constant	$died1$	Prolog function symbol	`died1`
Formula	$died1(d)$	Prolog term	`died1(d)`
Variable	x	Prolog variable	`X`
Lambda expression	$\lambda x[died1(x)]$	Prolog term containing ^	`X^died1(X)`

Table 4: Prolog representations of logic expressions

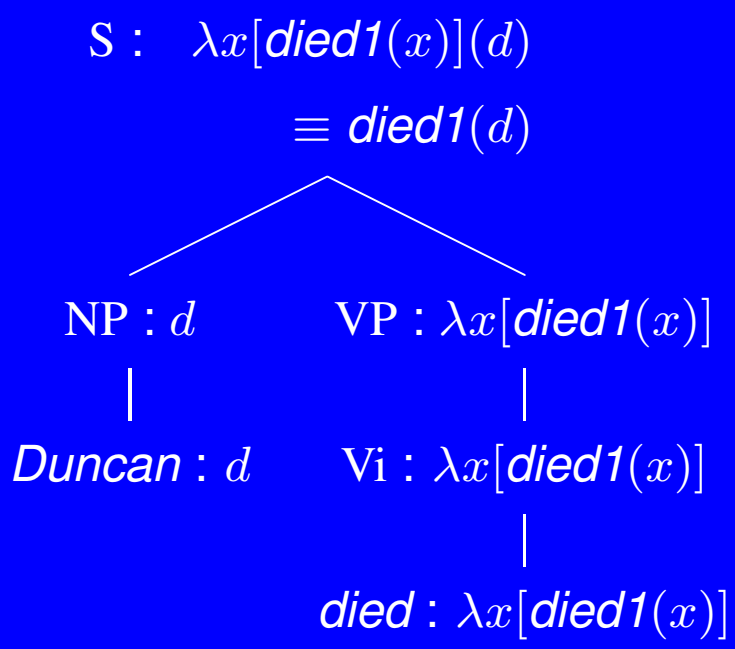


Figure 22: Annotated tree for *"Duncan died"*

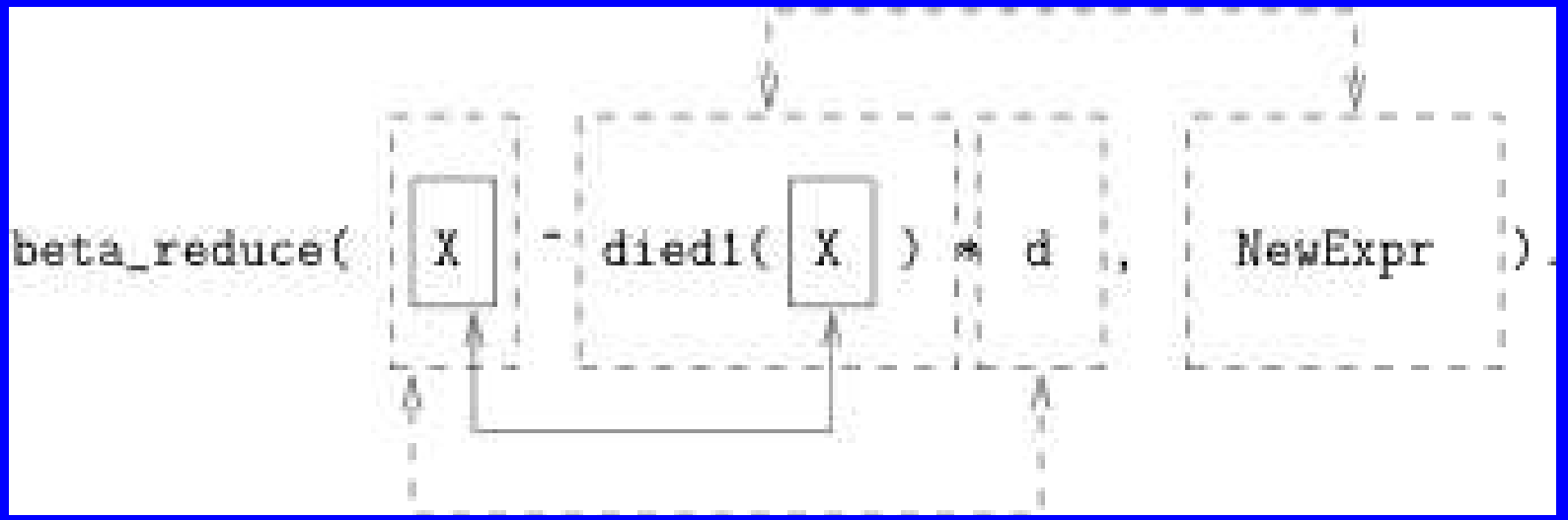
English syntax	FOL translation
$S \rightarrow NP\ VP;$	$S' = VP'(NP')$

- ```
  s(S_trans) ---> [np(NP_trans), vp(VP_trans)],
                  {beta_reduce(VP_trans*NP_trans, S_trans)}.
  ```

- ```
  beta_reduce(Arg^Expr*Arg, Expr).
  ```

- ```
  ?- beta_reduce(X^died1(X)*d, NewExpr ).
  NewExpr = died1(d)

  yes
  ```
  ```



Operation of the Prolog definition of beta-reduction

```
?- s(S_trans, [macbeth, killed, duncan], []).}
S_trans = killed1(m, d)

% BH-GRN grammar for semantic translation

s(S_trans)    ---> [np(NP_trans), vp(VP_trans)],
                   {beta_reduce((VP_trans, NP_trans), S_trans)}.
vp(Vi_trans)  ---> [vi(Vi_trans)].
vp(VP_trans)  ---> [vt(Vt_trans), np(NP_trans)],
                   {beta_reduce(Vt_trans*NP_trans, VP_trans)}.

np(d)                  ===> [duncan].
np(m)                  ===> [macbeth].
vi(X^died1(X))         ===> [died].
vt(Y^X^killed1(X, Y))  ===> [killed].

beta_reduce(Arg^Expr*Arg, Expr).
```

Grammar 6.2: BH-GRN grammar for semantic translation

6.7.1 Executing arbitrary Prolog goals with the shift-reduce parser

```
:-op(1100, xfx, --->).
:-op(1100, xfx, ===>).

term_expansion((X ---> (Y, Code)), ((X ---> Y):- Code)).
term_expansion((X ---> Y), (X ---> Y)).

term_expansion((X ===> (Y, Code)), ((X ===> Ydl):- Code)):-
  make_dl(Y, Ydl).
term_expansion((X ===> Y), (X ===> Ydl)):-
  make_dl(Y, Ydl).

make_dl(L, OpenL-Var):-
  append(L, Var, OpenL).
```

```
parse(Input, Result):-
parse([], [Result], Input, []).

parse([Stack], [Stack], [], []).

parse(Stack0, Stack, Words0, Words):-
  shift(Stack0, Stack1, Words0, Words1),
  reduce(Stack1, Stack2, Words1, Words1),
  parse(Stack2, Stack, Words1, Words).
```

```
shift(Stack, [Cat|Stack], Words0, Words):-
     (Cat ===> Words0-Words).

reduce([Top|Stack0], Stack, Words, Words):-
  reduce_aux(Stack0, [Top], Stack1),
  reduce(Stack1, Stack, Words, Words).

reduce(Stack, Stack, Words, Words).

reduce_aux(Stack, Top, [Cat|Stack]):-
  ( Cat ---> Top).

reduce_aux([Top|Stack0], Store, Stack):-
  reduce_aux(Stack0, [Top|Store], Stack).

'{}'(X):- call(X).
```

Program 6.1 Shift-reduce parser that can evaluate extra code

```
?- parse([macbeth, died], X).

X = s(died1(m)) ;

No
?- parse([macbeth, killed, duncan], X).

X = s(killed1(m, d))

Yes
```

```
[trace]  ?- parse([duncan,died], X) .
   Call: (8) parse([duncan, died], _G312) ?
   Call: (9) parse([], [_G312], [duncan, died], []) ?
   Call: (10) shift([], _L364, [duncan, died], _L365) ?
   Call: (11) _G388===>[duncan, died]-_G392 ?
   Exit: (11) np(_G396^ (_G396*d))===>[duncan, died]-[died] ?
   Exit: (10) shift([], [np(_G396^ (_G396*d))], [duncan, died], [died]) ?
   Call: (10) reduce([np(_G396^ (_G396*d))], _L366, [died], [died]) ?
   Call: (11) reduce_aux([], [np(_G396^ (_G396*d))], _L427) ?
   Call: (12) _G405--->[np(_G396^ (_G396*d))] ?
   Fail: (12) _G405--->[np(_G396^ (_G396*d))] ?
   Fail: (11) reduce_aux([], [np(_G396^ (_G396*d))], _L427) ?
   Redo: (10) reduce([np(_G396^ (_G396*d))], _L366, [died], [died]) ?
   Exit: (10) reduce([np(_G396^ (_G396*d))], [np(_G396^ (_G396*d))], [died],
                                    [died]) ?
```

```
Call: (10) parse([np(_G396^ (_G396*d))], [_G312], [died], []) ?
Call: (11) shift([np(_G396^ (_G396*d))], _L425, [died], _L426) ?
Call: (12) _G402===>[died]-_G406 ?
Exit: (12) vi(_G410^died1(_G410))===>[died]-[] ?
Exit: (11) shift([np(_G396^ (_G396*d))], [vi(_G410^died1(_G410)),
                np(_G396^ (_G396*d))], [died], []) ?
Call: (11) reduce([vi(_G410^died1(_G410)), np(_G396^ (_G396*d))],
                  _L427, [], []) ?
Call: (12) reduce_aux([np(_G396^ (_G396*d))], [vi(_G410^died1(_G410))],
                  _L488) ?
Call: (13) _G418--->[vi(_G410^died1(_G410))] ?
Exit: (13) vp(_G410^died1(_G410))--->[vi(_G410^died1(_G410))] ?
Exit: (12) reduce_aux([np(_G396^ (_G396*d))], [vi(_G410^died1(_G410))],
                [vp(_G410^died1(_G410)), np(_G396^ (_G396*d))]) ?
```

```
   Call: (12) reduce([vp(_G410^died1(_G410)), np(_G396^ (_G396*d))],
                     _L427, [], []) ?
   Call: (13) reduce_aux([np(_G396^ (_G396*d))], [vp(_G410^died1(_G410))],
                     _L547) ?
   Call: (14) _G426--->[vp(_G410^died1(_G410))] ?
   Fail: (14) _G426--->[vp(_G410^died1(_G410))] ?
   Redo: (13) reduce_aux([np(_G396^ (_G396*d))], [vp(_G410^died1(_G410))],
                     _L547) ?
   Call: (14) reduce_aux([], [np(_G396^ (_G396*d)), vp(_G410^died1(_G410))],
                     _L547) ?
   Call: (15) _G429--->[np(_G396^ (_G396*d)), vp(_G410^died1(_G410))] ?
   Call: (16) {beta_reduce(_G396^ (_G396*d)*_G410^died1(_G410), _G432)} ?
^  Call: (17) call(beta_reduce(_G396^ (_G396*d)*_G410^died1(_G410),
                     _G432)) ? s
^  Exit: (17) call(beta_reduce((d^died1(d))^ (d^died1(d)*d)*d^died1(d),
                     died1(d))) ?
   Exit: (16) {beta_reduce((d^died1(d))^ (d^died1(d)*d)*d^died1(d),
                     died1(d))} ?
```

```
Exit: (15) s(died1(d))--->[np((d^died1(d))^ (d^died1(d)*d)),
                vp(d^died1(d))] ?
Exit: (14) reduce_aux([], [np((d^died1(d))^ (d^died1(d)*d)),
                vp(d^died1(d))], [s(died1(d))]) ?
Exit: (13) reduce_aux([np((d^died1(d))^ (d^died1(d)*d))],
                [vp(d^died1(d))], [s(died1(d))]) ?
Call: (13) reduce([s(died1(d))], _L427, [], []) ?
Call: (14) reduce_aux([], [s(died1(d))], _L621) ?
Call: (15) _G481--->[s(died1(d))] ?
Fail: (15) _G481--->[s(died1(d))] ?
Fail: (14) reduce_aux([], [s(died1(d))], _L621) ?
Redo: (13) reduce([s(died1(d))], _L427, [], []) ?
Exit: (13) reduce([s(died1(d))], [s(died1(d))], [], []) ?
Exit: (12) reduce([vp(d^died1(d)), np((d^died1(d))^ (d^died1(d)*d))],
                [s(died1(d))], [], []) ?
Exit: (11) reduce([vi(d^died1(d)), np((d^died1(d))^ (d^died1(d)*d))],
                [s(died1(d))], [], []) ?
```

```
   Call: (11) parse([s(died1(d))], [_G312], [], []) ?
   Exit: (11) parse([s(died1(d))], [s(died1(d))], [], []) ?
   Exit: (10) parse([np((dˆdied1(d))ˆ (dˆdied1(d)*d))], [s(died1(d))],
                    [died], []) ?
   Exit: (9) parse([], [s(died1(d))], [duncan, died], []) ?
   Exit: (8) parse([duncan, died], s(died1(d))) ?

X = s(died1(d))

Yes
```

7 Quantified Noun Phrases

Grammar	Logic
$S \rightarrow NP\ VP$	$NP'(VP')$
$VP \rightarrow Vi$	Vi'
$VP \rightarrow Vt\ NP$	$Vt'(NP')$
$NP \rightarrow Det\ Nbar$	$Det'(Nbar')$
$NP \rightarrow PName$	$PName'$
$Nbar \rightarrow N$	N'

Lexicon	Logic
every : Det	$\lambda Q[\lambda P[\forall x(Q(x) \supset P(x))]]$
soldier : N	$\lambda x[soldier1(x)]$
Duncan : PName	$\lambda P[P(d)]$
died : Vi	$\lambda x[died1(x)]$
killed : Vt	$\lambda \mathcal{P}[\lambda y[\mathcal{P}(\lambda x[killed1(y,x)])]]$

Grammar 7.6: Grammar to be extended by PPs

Higher order expressions

a. $\lambda P[P(d)]$

b. `pname(P^P(d)) ===> [duncan].   % Illegal Prolog`

c. `pname(P^(P*d)  ===> [duncan].`

Connectives

$\neg$	=	~
$\wedge$	=	&
$\vee$	=	#
$\supset$	=	=>
$\leftrightarrow$	=	<=>

```
:- op(500, xfy, &).
```

Quantifiers

$\forall$ = `for_all` (e.g. `for_all(X, died1(X))`

$\exists$ = `exists` (e.g. `exists(X, died1(X))`

Logic formula	Prolog representation
$king1(d) \wedge \neg king1(m)$	`king1(d) & ~(king1(m))`
$\forall x(soldier1(x) \supset died1(x))$	`for_all(X, soldier1(X) => died1(X))`

Program betareduce.pl: Beta-reduction for higher-order logic

```
% Beta-reduction for  higher order logic
% betaredreduce.pl
% Based on code by Patrick Blackburn & Johan Bos
% Operator for functional application
:- op(400, yfx, *).

beta_reduce(Expression, Result):-
        beta_reduce(Expression, Result, []).

beta_reduce(Expr1, Expr2, []):-
    var(Expr1),
   Expr1=Expr2.
```

```
beta_reduce(Expression, Result, Stack):-
        nonvar(Expression),
        Expression = Functor*Argument,
        beta_reduce(Functor, Result, [Argument|Stack]),!.

beta_reduce(Expression, Result, [Var|Stack]):-
        nonvar(Expression),
        Expression = Var^Body,!,
        beta_reduce(Body, Result, Stack).
```

```
beta_reduce(Expression, Result, []):-
        nonvar(Expression),
        \+ (Expression = X^_, nonvar(X)),!,
        compose(Expression, Functor, Expressions),
        beta_reduce_all(Expressions, Results),
        compose(Result, Functor, Results).

compose(Term,Symbol,ArgList):-
    Term =.. [Symbol|ArgList].

beta_reduce_all([], []).
beta_reduce_all([First|Rest], [NewFirst|NewRest]):-
        beta_reduce(First, NewFirst),
        beta_reduce_all(Rest, NewRest).
```

Operators, quantifiers and connectives

```
:- op( 30, fy , ~).
:- op(500, xfy , &).
:- op(500, xfy , #).
:- op(510, xfy , =>).
:- op(510, xfy , <=>).
```

```
s(S_trans)    --> [np(NP_trans), vp(VP_trans)],
                  {beta_reduce(NP_trans*VP_trans, S_trans)}.
vp(Vi_trans)  --> [vi(Vi_trans)].
vp(VP_trans)  --> [vt(Vt_trans), np(NP_trans)],
                  {beta_reduce(Vt_trans*NP_trans, VP_trans)}.
np(NP_trans)  --> [det(Det_trans), nbar(Nbar_trans)],
                  {beta_reduce(Det_trans*Nbar_trans, NP_trans)
                       }.
np(Pname_trans) --> [pname(Pname_trans)].
nbar(N_trans) --> [n(N_trans)].
```

```
det(Q^P^for_all(X, Q*X => P*X)) ===> [every].
n(X^soldier1(X))                ===> [soldier].
np(P^(P*d))                     ===> [duncan].
np(P^(P*m))                     ===> [macbeth].
vi(X^died1(X))                  ===> [died].
vt(NP1^Y^(NP1*X^killed1(Y, X))) ===> [killed].
```

Grammar 7.8: BH-GRN grammar for semantic translation

```
?-  parse([duncan, killed, every, soldier], X).
X = s(for_all(_G284, soldier1(_G284)=>killed1(_G284, d)))

Yes
```

1. Load program C.1
2. Load the higher-order beta-reduction program
3. Load Grammar 7.8
4. Sample call:

```
?- s(X, [every, soldier, died], []).
X = for_all(_G223, soldier1(_G223)=>died1(_G223))

Yes
```

1. Quit Prolog
2. Load the extended shift-reduce parser
3. Load the higher-order beta-reduction program
4. Load Grammar 7.8
5. Sample call:

```
?- parse([every, soldier, died], X)).
X = s(for_all(_G223, soldier1(_G223)=>died1(_G223)))

Yes
```

Scope ambiguity and Cooper storage

Grammar	Matrix rule	Store rule
$\text{S} \rightarrow \text{NP VP}$	$\text{S}' = \text{NP}'(\text{VP}')$	$\text{S}'' = \text{NP}'' \cup \text{VP}''$
$\text{VP} \rightarrow \text{Vi}$	$\text{VP}' = \text{Vi}'$	$\text{VP}'' = \text{Vi}''$
$\text{VP} \rightarrow \text{Vt NP}$	$\text{VP}' = \text{Vt}'(\text{NP}')$	$\text{VP}'' = \text{Vt}'' \cup \text{NP}''$
$\text{NP} \rightarrow \text{Det Nbar}$	$\text{NP}' = \lambda P[P(z)]$	$\text{NP}'' = \text{Det}'' \cup \text{Nbar}'' \cup \{\langle \text{Det}'(\text{Nbar}'), z\rangle\}$
$\text{NP} \rightarrow \text{PName}$	$\text{NP}' = \text{PName}'$	$\text{NP}'' = \text{PName}''$
$\text{Nbar} \rightarrow \text{N}$	$\text{Nbar}' = \text{N}'$	$\text{Nbar}'' = \text{N}''$

Lexicon	Matrix	Store
every : Det	$\lambda Q[\lambda P[\forall x(Q(x) \supset P(x))]]$	$\emptyset$
soldier : N	$\lambda x[soldier1(x)]$	$\emptyset$
Duncan : PName	$\lambda P[P(d)]$	$\emptyset$
died : Vi	$\lambda x[died1(x)]$	$\emptyset$
killed : Vt	$\lambda \mathcal{P}[\lambda y[\mathcal{P}(\lambda x[killed1(y,x)])]]$	$\emptyset$

Grammar 7.9: Simplified storage grammar

Prolog implementation

A reminder about difference lists

```
vp(S0, S):-
    vt(S0, S1),
    np(S1, S).

vp([drinks, scotch, on, ice], []):-
    vt([drinks, scotch, on, ice], [scotch, on, ice]),
    np([scotch, on, ice], []).

vt([drinks | Words], Words).

vp(S0–S):-
    vt(S0–S1),
    np(S1–S).
```

Grammar	Matrix rule	Store rule
$VP \rightarrow Vt\ NP$	$VP' = Vt'(NP')$	$VP'' = Vt'' \cup NP''$

```
vp(VPmx, Store0-Store) --->
   [vt(Vtmx, Store0-Store1), np(NPmx, Store1-Store)],
   {beta_reduce(Vtmx*NPmx, VPmx)}.
```

Grammar	Matrix rule	Store rule
$\text{NP} \rightarrow \text{Det Nbar}$	$\text{NP}' = \lambda P[P(z)]$	$\text{NP}'' = \text{Det}'' \cup \text{Nbar}'' \cup \{\langle \text{Det}'(\text{Nbar}'), z \rangle\}$

```
np(P^(P*Z), [stored(NPmx, Z) | Store0]-Store) ---->
   [det(Detmx, Store0-Store1), nbar(Nbarmx, Store1-Store)],
   {beta_reduce(Detmx*Nbarmx, NPmx)}.
```

```
s(Smx, Store0-Store) --->
  [np(NPmx, Store0-Store1), vp(VPmx, Store1-Store)],
  {beta_reduce(NPmx*VPmx, Smx)}.
vp(Vimx, Store) --->
  [vi(Vimx, Store)].
vp(VPmx, Store0-Store) --->
  [vt(Vtmx, Store0-Store1), np(NPmx, Store1-Store)],
  {beta_reduce(Vtmx*NPmx, VPmx)}.
np(P^(P*Z), [stored(NPmx, Z) | Store0]-Store) --->
  [det(Detmx, Store0-Store1), nbar(Nbarmx, Store1-Store)],
  {beta_reduce(Detmx*Nbarmx, NPmx)}.
np(PNamemx, Store) --->
  [pname(PNamemx, Store)].
nbar(Nmx, Store) --->
   [n(Nmx, Store)].
```

```
det(Q^P^for_all(X, Q*X => P*X), S-S) ===> [every].
det(Q^P^exists(X, Q*X & P*X), S-S)   ===> [some].

n(X^soldier1(X), S-S)                ===> [soldier].
n(X^witch1(X), S-S)                  ===> [witch].

pname(P^(P*d), S-S)                  ===> [duncan].

vi(X^died1(X), S-S)                  ===> [died].
vt(NP^Y^(NP*X^killed1(Y, X)), S-S)     ===> [killed].
```

Grammar 7.10 A BH-GRN grammar that achieves simplified Cooper storage

1. Load the extended shift-reduce parser
2. Load the higher-order beta-reduction program
3. Load Grammar 7.10
4. Sample call (needs additional lexical entries for ‘some’ and ‘witch’):

```
?- parse([every, soldier, killed, some, witch], s(Matrix,
     Store)).
Matrix = killed1(_G361, _G517)
Store = [stored(_G292^for_all(_G295, soldier1(_G295)
                  =>_G292*_G295), _G361),
      stored(_G541^exists(_G544, witch1(_G544)
                  &_G541*_G544), _G517)|_G556]-_G556

Yes
```

Quantifier retrieval

Let S' be the matrix of the S.

An item in the store is of the form $\langle \mathrm{NP}', z \rangle$, i.e. NP' is the stored NP translation and z is the stored variable (which is free in S').

To use a retrieved item to scope S', we compute:

$$\mathrm{NP}'(\lambda z[\mathrm{S}'])$$

```
?- parse([every, soldier, killed, some, witch],
         s(Smx, Sstore-[])),
         scope(Smx, Sstore, ScopedStrans).
```

```
scope(Mx, [], Mx).
scope(Oldmx, StoredNPs, Scopedmx) :-
    remove(stored(NPmx, Var), StoredNPs, NewStore),
    apply_np(NPmx, Var, Oldmx, Newmx),
    scope(Newmx, NewStore, Scopedmx).

apply_np(NPmx, Var, Oldmx, Newmx) :-
    beta_reduce(NPmx*(Var^Oldmx), Newmx).

% remove(X, Ys, Zs) --- list Zs is the result
%                       of removing an occurrence of
%                       item X from list Ys
remove(X, [X|Xs], Xs).
remove(X, [Y|Ys], [Y|Zs]) :- remove(X, Ys, Zs).
```

Prolog code for retrieving stored NP translations

```
?- parse([every, soldier, killed ,some,witch],
                    s(Smx, Sstore-[])),
       scope(Smx, Sstore, ScopedStrans).

Smx = killed1(_G310, _G488)

ScopedStrans = exists(_G488, witch1(_G488)
                  &for_all(_G310, soldier1(_G310)
                           =>killed1(_G310, _G488))) ;

Smx = killed1(_G310, _G488)

ScopedStrans = for_all(_G310, soldier1(_G310)
                  =>exists(_G488, witch1(_G488)
                           &killed1(_G310, _G488))) ;

No
```

1. Load the extended shift-reduce parser
2. Load the higher-order beta-reduction program
3. Load the scoping code
4. Load Grammar 7.10
5. Sample call (needs additional lexical entries for ‘some’ and ‘witch’):

```
?- parse([every, soldier, killed ,some,witch],
                        s(Smx, Sstore-[])),
        scope(Smx, Sstore, ScopedStrans).
Smx = killed1(_G310, _G488)
ScopedStrans = exists(_G488, witch1(_G488)
                        &for_all(_G310, soldier1(_G310)
                                =>killed1(_G310, _G488))) ;
```

6. Force Prolog to find alternative solutions:

```
Smx = killed1(_G310, _G488)

ScopedStrans = for_all(_G310, soldier1(_G310)
                    =>exists(_G488, witch1(_G488)
                          &killed1(_G310, _G488))) ;

No
```

1. Load program C.1
2. Load the higher-order beta-reduction program
3. Load the scoping code
4. Load Grammar 7.10
5. Sample call (needs additional lexical entries for ‘some’ and ‘witch’):

```
s(Smx,Sstore-[],[every,soldier,killed,some,witch],[]),
        scope(Smx, Sstore, ScopedStrans).
Smx = killed1(_G296, _G427)
ScopedStrans = exists(_G427, witch1(_G427)
                        &for_all(_G296, soldier1(_G296)
                                =>killed1(_G296, _G427))) ;
```

6. Force Prolog to find alternative solutions:

```
Smx = killed1(_G296, _G427)

ScopedStrans = for_all(_G296, soldier1(_G296)
                              =>exists(_G427, witch1(_G427)
                                  &killed1(_G296, _G427))) ;

No
```

8 Interfaces to databases

```
linguistic_analyser(String) :-
        setof(parse(Matrix, Store),
              s(Matrix, Store-[], String, []),
              Parses), !,
        findall(reading(ScopedTrans),
                (member(parse(Mx, St), Parses),
                scope(Mx, St, ScopedTrans)),
                    Readings), !,
        answer(Readings).
```

```
?- setof(X, member(X, [a,b,c,a]), Set).

X = _G159
Set = [a, b, c]

Yes
?- setof(item(X), member(X, [a,b,c,a]), Set).

X = _G147
Set = [item(a), item(b), item(c)]

Yes
```

```
?- setof(parse(Matrix, Store),
          s(Matrix, Store-[], [every, soldier, died], []),
        Parses).

Matrix = _G170
Store = _G171
Parses = [parse(died1(_G314),
        [stored(_G322^for_all(_G325, soldier1(_G325)
                =>_G322*_G325), _G314)])]
Yes
```

```
?- Parses = [parse(died1(_G291),
                   [stored((_G302^_G303)^for_all(_G302, soldier1(
                        _G302)
                       =>_G303), _G291)])],
findall(reading(ScopedTrans),
          (member(parse(Mx, St), Parses),
           scope(Mx, St, ScopedTrans)),
        Readings).

Readings = [reading(for_all(_G435, soldier1(_G435)
                       =>died1(_G435)))]

Yes
```

9 Database

```
witch1(w1).
witch1(w2).
witch1(w3).
witch1(w4).

died1(X):-
        killed1(_, X).

soldier1(s1).
soldier1(s2).
soldier1(s3).
soldier1(s4).
```

```
killed1(m, d).
killed1(m, w1).

killed1(s1, w1).
killed1(s2, w2).
killed1(s3, w3).
killed1(s4, w4).
```

Interpretation of quantifiers and connectives

Connectives

```
~P :- \+ P.

(P & Q):-
        call(P),
        call(Q).
```

Quantifiers

```
for_all(_X, P => Q):-
        \+ (call(P),
                \+ call(Q)).

exists(_X, P):-
        call(P).

?- exists(X, soldier1(X) & exists(Y, witch1(Y) & killed1(X, Y))
    ).

X = s1
Y = w1

Yes
```

Semantic evaluation

```
answer(Readings):-
        member(reading(ScopedTrans), Readings),
        message(ScopedTrans),
        fail.
answer(_).

message(Formula):-
        write(Formula), nl,
        ( call(Formula)->
        write('That''s right.'), nl
        ;
        write('I don''t think so.')
        )
        , nl.
```

```
?- linguistic_analyser([every, soldier,killed,some,witch]).
exists(_G353, witch1(_G353)&for_all(_G361, soldier1(_G361)
       =>killed1(_G361, _G353)))

I don't think so.

for_all(_G326, soldier1(_G326)
      =>exists(_G334, witch1(_G334)&killed1(_G326, _G334)))

That's right.

Yes
```

Loading files automatically:

create a file (load_td.pl) which contains a directive like the following

```
:- [operators, programC-1,betareduce,scoping,frontend].
```

where this is a list of the Prolog files required.

Compile `load.pl`:

```
%  programC-1 compiled 0.01 sec, 0 bytes
%  betareduce compiled 0.00 sec, 0 bytes
%  scoping compiled 0.00 sec, 0 bytes
%  frontend compiled 0.00 sec, 128 bytes
% /tmp/prolcomp7920fli compiled 0.02 sec, -2,340 bytes

Yes
```

Then consult a grammar file.

Bottom up parsing

We need to modify the code for the linguistic analyser so that it uses a bottom up parser for the parsing part of the program:

```
linguistic_analyser(String) :-
   setof(parse(Matrix, Store),
         parse(String, s(Matrix, Store-[])),
                %shift reduce parser
        Parses), !,
   findall(reading(ScopedTrans),
           (member(parse(Mx, St), Parses),
           scope(Mx, St, ScopedTrans)),
               Readings), !,
        answer(Readings).
```

To automatically load the code required, create a file (`load_bu.pl`) which contains a directive like the following:

```
:-[operators, betareduce, scoping, frontend_bu, program6-1]
```

where this is a list of the Prolog files required (and `program6-1.pl` is the program that contains the version of the shift-reduce parser that can execute Prolog code contained in curly brackets).

Compile `load_bu.pl`:

```
%  betareduce compiled 0.01 sec, 0 bytes
%  scoping compiled 0.00 sec, 0 bytes
%  frontend_bu compiled 0.01 sec, 0 bytes
%  program6-1 compiled 0.00 sec, 0 bytes
% /tmp/prolcomp931BLe compiled 0.02 sec, 3,964 bytes

Yes
```

Then compile your grammar and run the front end:

```
?- linguistic_analyser([every, soldier,killed, some, witch]
exists(_G348, witch1(_G348)
        &for_all(_G356, soldier1(_G356)
                 =>killed1(_G356, _G348)))
I don't think so.
for_all(_G321, soldier1(_G321)
         =>exists(_G329, witch1(_G329)
                  &killed1(_G321, _G329)))
That's right.


Yes
```

Making it all neater

```
nlidb :-
  write('Welcome to BH-NLIDB'), nl, nl,
    repeat,
    nl,
    write('> '),
    read_string(String), nl,
    process_string(String).

process_string(String):-
        String = [stop], !,
        write('Goodbye'), nl.

process_string(String):-
    linguistic_analyser(String),
    fail.
```

```
?- nlidb.
Welcome to BH-NLIDB

> macbeth died

died1(m)
I don't think so.

> duncan died

died1(d)
That's right.
```

```
> every soldier killed some witch

exists(_G416, witch1(_G416)
         &for_all(_G424, soldier1(_G424)
                =>killed1(_G424, _G416)))
I don't think so.
for_all(_G389, soldier1(_G389)
          =>exists(_G397, witch1(_G397)
                 &killed1(_G389, _G397)))
That's right.

> stop

Goodbye

Yes
```

Finally, add an error message for the situation when things go wrong:

```
linguistic_analyser(_) :-
  write('Linguistic analyser: Could not process'), nl.
```

A second clause for the original `linguistic_analyser/1` .

```
Welcome to BH-NLIDB


> died duncan

Linguistic analyser: Could not process

>
```