# Declarative Programming and Natural Language

Søren Jakob Løvborg

**Abstract**

This paper analyzes benefits and challenges (together with possible solutions) of using natural language processing for data entry and computer programming.

The paper looks at data entry in existing declarative languages and the underlying relational model is analyzed, also covering the subject of semantic networks. Context-free grammars are described in the context of natural language parsing, and modified Earley parsing algorithm is described and implemented, dispensing with some unnecessary complexities of the original.

A number of challenges of natural language are described, from the difficulties of identifying and classifying lexemes, to the ambiguous constructions of everyday English.

The paper concludes that while natural language is too complex for computers to grok in the general case, natural language may be viable in highly domain-specific areas.

Kongens Lyngby 2007
IMM-B.Sc-2007-16

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

# Contents

# 1 Introduction

Ordinarily, computers have been programmed using imperative programming languages (such as assembler code, C or Java), characterized in that algorithms are explicitly spelled out.

Declarative programming (which include logic programming and functional programming) takes a higher-level, more mathematically stringent approach, in which only a goal is stated, and the computer is then tasked with finding an appropriate algorithm to reach this goal.

The fact that most programming remains imperative must be attributed to the fact that programmers are much better at designing algorithms than computers. Indeed, many declarative languages (such as Prolog, ML and Lisp) include imperative constructs, sacrificing purity for increased control over program execution.

Declarative programming is not without its advantages, however. Because it does away with the imperative notion of *state*, declarative programming is suitable for massive parallelization.

More interestingly, one may argue that declarative programming more closely matches the way humans represent knowledge in writing.

While imperative programming requires understanding of state and flow control, declarative programming only requires understanding of knowledge representation and syntax. This may give non-specialists insight into the operations of a program.

Another way to make the life easier for non-specialists is to use *natural language* (i.e. English) for programming. This is considerably more difficult to implement properly, as can be seen by the relatively few attempts (successful or otherwise) at doing this. However, its feasibility for domain-specific tasks is demonstrated by such diverse experiments as SHRDLU and the Inform 7 programming language.

## 1.1 Problem description

This paper analyzes benefits and challenges (together with possible solutions) of using natural language processing for data entry and computer programming. However, many of the problems are beyond the scope of this paper, and are indeed unlikely to be solved for many years.

The paper describes a number of existing declarative programming languages and natural language based languages in section 2.

The nature of words in the English language is discussed in section 3, while section 4 discuss the mathematics of language and grammar. Section 5 describes a modified Earley parser, and section 6 an actual implementation.

Representing knowledge using semantic networks is discussed in section 7.

# 2 Overview of existing solutions

Since the 1950s, much work has gone into designing languages for declarative programming, resulting in logic programming languages like Prolog and Gödel, and functional languages like Lisp, ML and Haskell.

The topic of natural programming languages has not received as much attention. Practical English-like languages have been crude, as well as few and far between. Notable attempts include AppleScript and Macromedia's Lingo (both inspired by Dan Winkler's 1987 HyperTalk language), SQL and COBOL, all of which are still in use in some form or other, for application scripting, database queries, and financial systems.

## 2.1 Mathematical notation

Although not strictly a *programming* language, mathematical notation is the most widespread formalized declarative language in use, and a language that has evolved over thousands of years.

Recall that a mathematical $n$-ary *relation* is an $n$ argument boolean-valued function (a *predicate*), defining how values relate to eachother. A simple example is the binary equals relation, defined

$$\text{equals}(a, b) \quad \Leftrightarrow \quad a = b$$

As an example, the following relations are introduced:

| | |
|---|---|
| $\text{parent}(p, c)$ | $p$ is the parent of $c$. |
| $\text{father}(p, c)$ | $p$ is the father of $c$. |
| $\text{mother}(p, c)$ | $p$ is the mother of $c$. |
| $\text{childof}(c, m, f)$ | $c$'s mother is $m$, and $c$'s father is $f$. |
| $\text{sibling}(c_1, c_2)$ | $c_1$ is the sibling of $c_2$. |

The relations are defined in terms of each other as follows:

$$\text{parent}(p, c) \Leftarrow \text{father}(p, c)$$

$$\text{parent}(p, c) \Leftarrow \text{mother}(p, c)$$

$$\text{childof}(c, m, f) \Leftrightarrow \text{mother}(m, c) \land \text{father}(f, c)$$

$$\text{sibling}(c_1, c_2) \Leftarrow c_1 \neq c_2 \land \exists p : (\text{parent}(p, c_1) \land \text{parent}(p, c_2))$$

We can then make assertions about people to construct a family tree. An example drawn from norse mythology is given in figure 1.

$$\text{childof}(\mathsf{odin}, \mathsf{bestla}, \mathsf{borr})$$
$$\text{childof}(\mathsf{vili}, \mathsf{bestla}, \mathsf{borr})$$
$$\text{childof}(\mathsf{ve}, \mathsf{bestla}, \mathsf{borr})$$

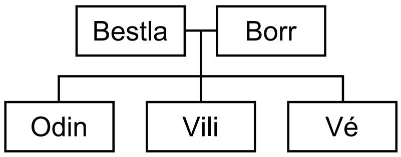

Figure 1: *A family tree in terms of* childof*-relations, and as a pedigree chart*

## 2.2 Prolog

Prolog is a logic programming language, with a pure declarative core and a number of imperative extensions.

In predicate logic, a *definite Horn clause* is an expression on the form

$$A \Leftarrow B_1 \land B_2 \land \cdots \land B_n$$

where $A$ and $B_i$ are all predicates of the form $P(p_1, \ldots, p_m)$.

A basic Prolog program consists of a number of Horn clauses, and the user can then ask the program to prove predicates. Central to Prolog is *negation as failure*, in which anything not provably true is assumed to be false.

Since Prolog is limited to Horn clauses, the rule

$$\text{childof}(c, m, f) \Leftrightarrow \text{mother}(m, c) \land \text{father}(f, c)$$

cannot be described. The best we can do is

$$\text{childof}(c, m, f) \Leftarrow \text{mother}(m, c) \land \text{father}(f, c)$$

This restriction greatly reduces the complexity of the Prolog theorem prover, but complicates programming.

```
parent(P,C) :- father(P,C).
parent(P,C) :- mother(P,C).
childof(C,M,F) :- mother(M,C), father(F,C).
sibling(C1,C2) :- parent(P,C1), parent(P,C2), \dif(C1, C2).
```

Due to the modified childof definition, the family tree cannot be specified using

```
childof(odin, bestla, borr).
childof(ve, bestla, borr).
childof(vili, bestla, borr).
```

since childof no longer implies the mother and father relation. Instead, the family tree must be specified more verbosely,

```
mother(bestla, odin).
mother(bestla, ve).
mother(bestla, vili).
father(borr, odin).
father(borr, ve).
father(borr, vili).
```

Once this is done, queries can be made against the data set:

```
| ?- childof(odin,bestla,borr).
true ? a
no

| ?- childof(C,M,F).
C = odin
F = borr
M = bestla ? a

C = ve
F = borr
M = bestla

C = vili
F = borr
M = bestla

yes
```

## 2.3 Relational database management systems and SQL

Codd's classic paper on relational database management revolutionized database construction by introducing the mathematical concept of relations to data management. [Codd70] Where earlier databases had used a simple table–record structure, and the merging of data from different tables would have had to be done in the application, this merger could now be performed in the database.

Codd's definition of *named relations* match what SQL refers to as *tables*: A named set of $n$-tuples satisfying the $n$-ary relation.

The father-relation could be created as a table with this command:

```
CREATE TABLE father (parent VARCHAR(50), father VARCHAR(50))
```

However, working with father and mother in this manner would be impractical; in SQL, it's easiest to take an "object-centrist" approach, in which each table row (or tuple) tells us as much as possible about a single object. The result is more akin to the childof relation, the names being the only difference:

```
CREATE TABLE Person (name VARCHAR(50), mother VARCHAR(50), father VARCHAR(50))
```

## 2.4 SHRDLU

Terry Winograd's SHRDLU from the late 1960s was a computer program that allowed the user to interact with a virtual world of three-dimensional shapes through natural language. At the time, its apparent intelligence was a tremendous breakthrough in artificial intelligence and natural language research, but its successful understanding of language was largely dependent on the narrow confines of its virtual world, and later research failed to make the program less *domain-specific* while retaining its (apparent) intelligence. Winograd himself declared it to be a dead-end in AI research.

Nevertheless, SHRDLU demonstrates a number of desirable qualities in a natural language processing program, as seen in these excerpts: [Winograd71]

*Basic sentences*
> Pick up a big red block.
OK.

*Comparative adjectives ("taller") and pronouns ("it")*
> Find a block which is taller than the one you are holding and put it into the box.
By "it", I assume you mean the block which is taller than the one I am holding. OK.

*Negation and counting*
> How many blocks are not in the box?
Four of them.

*Disambiguation*
> How many things are on top of green cubes?
I'm not sure what you mean by "On top of" in the phrase "on top of green cubes".
Do you mean:
1 – Directly on the surface
2 – Anywhere on top of?

*Memory of past events*
> Had you touched any pyramid before you put the green one on the little cube?
Yes, the green one.

*Learning by definition*
> A "steeple" is a stack which contains two green cubes and a pyramid.
I understand.
> Are there any steeples now?
No.
> Build one.
OK.

Greatly simplifying SHRDLU's task was the fact that it had perfect knowledge of the state of its world, although not of the *rules* – it had to *learn* that triangles wouldn't stack, for instance.

## 2.5 Inform 7

Inform 7 is a language and authoring system designed largely for writers, not programmers. A notable feature is that the programming language is based on natural language.

Thus, we can define the father, mother and childof relations as such:

```
Chapter 1 - Definitions

Fatherhood relates one man (called the father) to various people.
Motherhood relates one woman (called the mother) to various people.

The verb to be the father of implies the fatherhood relation.
The verb to be the mother of implies the motherhood relation.

Childhood relates a person (called the child) to a person (called the parent) when the parent is
the father of the child or the parent is the mother of the child.

The verb to be the child of implies the childhood relation.

Section 1.1 - Family tree

Bestla is the mother of Odin.
Borr is the father of Odin.

Odin is the father of Balder.
Frig is the mother of Balder.
Odin is the father of Hod.
Odin is the father of Thor.
```

The example illustrates how Inform 7 allows clear and concise declarations in natural language, at the cost of some quite verbose definitions.

Of particular interest is the readability of the source code. Even to readers having no previous experience with the language, the code remains quite clear, simply because it mimics the English language. In comparison, the Prolog code may be a lot more concise, but it's also completely opaque to people without the necessary mathematical background.

In this respect, Inform 7 can be seen as a perfection of Donald Knuth's "literate programming" philosophy. Unlike most realizations (TEX, javadoc, etc.), where programming code and documentation are merely interleaved in the source files, Inform 7 conflates the two: the documentation *is* the code.

[Nelson05] discuss this and other Inform 7 design issues.

# 3 Lexical analysis

During execution, a natural language processing program receives a string of characters as input. Since few non-logographic languages are best described as a string of separate characters, the input characters must be grouped into words (*tokens* or *terminals*), before being sent to the parser. This procedure is known as lexical analysis, scanning or tokenization.

It is the first step that a program must undertake when dealing with textual input, and for programming languages is often quite simple.[1]

## 3.1 Words of natural language

Of course, nothing in natural language is simple. [Trask04] gives four markedly different definitions of what a word is, two of which are useful for our purposes.

**Orthographic words** are strings of letters, numbers and hyphens, delimited by spaces or other non-alphanumeric characters. Orthographic words are thus readily identifiable in a text, and a suitable basis for tokenization.

This division does not always match the semantics, however. "Ice cream" is semantically a single word, but consists of two orthographic words.

**Lexemes** are semantic units of the text. A lexeme like JUMP correspond to the ortographic word "jump", as well as all its inflections (jumps, jumped, jumping). The set of known lexemes is referred to as the *lexicon*.

A lexeme always belong to exactly one syntactic class. As such, $\text{BILL}_\text{N}$ (the noun "a bill"), $\text{BILL}_\text{V}$ (the verb "to bill") and $\text{BILL}_\text{PN}$ (the proper noun "Bill") are three different lexemes.

A lexeme may also correspond to a sequence of orthographic words, as in "ice cream" ($\text{ICE-CREAM}_\text{N}$).

Determining exactly which orthographic words match which lexemes is a complicated procedure which requires understanding of the sentence structure, something which we only have after the parsing step. Hence the parser is fed othographic words and not lexemes.

## 3.2 Compound nouns

The above mentioned "ice cream" is an example of a compound noun. Compound nouns are nouns that consist of two or more words, and the challenge is to properly identify them as a single lexeme.

[1] A notable exception is C++, where the lexical analysis is intertwined not only with the parsing, but also with the following semantic analysis.

Although it may seem desireable for the semantic step to analyse the underlying composition (ice + cream, ash + tray), this is very difficult to do properly even for humans, as the compound does not reveal how the parts relate to eachother: Ice cream is cream made out of ice, but an ashtray is not a tray made out of ash.

One may distinguish between three ways of writing compound nouns:

The **closed form** is when the words are joined together, as in "ashtray", and is thus quite easy to handle from a parsing perspective, as we have a one-to-one mapping between orthographic word and lexeme.

The **hyphenated form** is when the words are connected by a hyphen, and is equally easy to parse.

The **open form** is when the words are separated by spaces, as in "ice cream", and presents a difficult parsing problem, easily resulting in ambiguities.

Compound nouns are further complicated by the fact that while many are limited to one of the above forms (e.g. "ice age", not "iceage" or "ice-age"), others can be written in two or all three of the above forms (both "ice cream" and "ice-cream" are correct). Additionally, some forms are in widespread use, even if not sanctioned by dictionaries ("ash tray").

One might conclude that program should accept compound nouns in all three forms, but this causes problems on its own: A "green house" is quite different from a "greenhouse".

For general-purpose natural language processing, it is not practical to build a comprehensive catalog of compound nouns either, since their number is too high, new compounds are invented every day, and old compounds change form quickly – "data base" became "database" in less than two decades.

## 3.3 Classifying words

Determining which word classes (nouns, verbs, etc.) to divide words into is in itself a difficult task.

A single class for *proper nouns* and another for *common nouns* may for instance seem appropriate, but English have two different types of common nouns: *count nouns* and *mass nouns.* Count nouns are ordinary nouns (e.g. "cat") which are countable and have a singular and plural form. Mass nouns are nouns like "water", which are uncountable, and are neither singular nor plural.

To further complicate matters, mass nouns may in some circumstances be used as count nouns, e.g. "Which of these *oils* is better for cooking?" One may even have *a beef* with someone.

It seems reasonable then to distinguish between the count noun $\textsc{beef}_{\text{CN}}$ and the mass noun $\textsc{beef}_{\text{MN}}$ at the lexeme level.

## 3.4 Inflection and stemming

A lexeme has an associated *citation form* or *lemma*; the base form of the word. For count nouns, it is the singular, for verbs it's the infinitive, for mass nouns and proper nouns, it's the only form of the word (not counting the possessive case).

The lemma is inflected to produce the different forms of the lexeme. Regular words are inflected according to a simple set of rules, but English also contains a large number of irregular words, where the different forms cannot simply be computed from the citation form, and instead must be stored explicitly. (The worst offender is the verb *to be*.)

The regular inflection rules for plural count nouns can be described thus: If the word ends on "s", "sh", "x", "z", add "es". Otherwise, if the word ends on "y", replace that "y" with "ies". Otherwise, add "s".

In shorthand notation, the inflection rules become:

| | |
|---|---|
| *Plural* | s/sh/x/z → -es |
| | y → ies |
| | $\epsilon$ → s |
| *Posessive* | s/x/z → -' |
| | $\epsilon$ → 's |
| *Comparative* | &d → -der |
| | &g → -ger |
| | &t → -ter |
| | e → -r |
| | y → -ier |
| | $\epsilon$ → -er |
| *Superlative* | &d → -dest |
| | &g → -gest |
| | &t → -test |
| | e → -st |
| | y → -iest |
| | $\epsilon$ → -est |

(Here, & indicates any vowel.)

Once we can inflect lexemes, we can also *stem* orthographic words and obtain the corresponding lexeme along with its grammatical features. This is done by adding all lemmas and irregular inflections to a *multi map*, mapping to the corresponding lexeme. A multi map is required since an orthographic word may map to multiple lexemes (possibly in different forms).

To find the lexemes corresponding to a word, and the above rules are applied in reverse, and the result is looked up in the map. An implementation of this can be found in the `jp.lexer.Feature` and `jp.lexer.Lexicon` (appendix B.3 and B.3), and a sample program can be found in `jp.StemTest` (appendix B.4).

# 4 Formal languages

A formal language is a mathematical representation of language, in which a language $L$ is defined as a set of symbol strings.[2]

The symbols can be anything, and are often characters; in describing natural languages, it's useful to define symbols as words and punctuation marks. These symbols are denoted *terminal symbols* or *tokens*, and the set $\Sigma$ of allowed symbols must be finite.

The language set contains all strings that are valid in that language. An example of a finite language could thus be the language of twin Arabic digits,

$$L = \{00, 11, 22, 33, 44, 55, 66, 77, 88, 99\}$$

over the symbols

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Most languages, however, are infinite, thus requiring us to describe them in other ways than simply enumerating every legal string. Formal devices for this purpose are called *formal grammars.*

The Chomsky hierachy (figure 2) orders languages (and the grammars describing them) after the complexity of recognizing valid strings (and rejecting invalid strings), such that every category is a proper subset of the categories above it in the hierachy.

| Language | Automaton |
|---|---|
| Recursively enumerable | Turing machine |
| Recursive | Decider |
| Context-sensitive | Linear-bounded |
| Indexed | Nested stack |
| Context-free | Nondeterministic pushdown |
| Deterministic context-free | Deterministic pushdown |
| Regular | Finite |

Figure 2: *The augmented Chomsky hierarchy of formal languages* [Elder05]

The most well-known of these are *regular languages* (typically described by regular expressions, and recognizable by a finite automaton) and *context-free languages* (described by context-free grammars, such as BNF grammars, and recognized by a non-deterministic pushdown automaton.)

[2]These strings are sometimes referred to as *words*, but to avoid confusion (as we're discussing natural language), we'll use *word* in the senses discussed in section 3.1.

## 4.1 Context-free grammars

Context-free grammars (CFGs) can themselves be divided into several complexity categories:

$$\text{LL(k)} \subset \text{LR(1)} \subseteq \text{deterministic CFG} \subset \text{unambiguous CFG} \subset \text{all CFG}$$

A context-free grammar is said to be deterministic, if it can be implemented using a deterministic pushdown automaton (as opposed to a non-deterministic PDA).

It's preferable to deal with deterministic context-free languages, since they may be parsed using efficient LL($k$) and LR(1) parsers. For this reason, most programming languages strive to be deterministic context-free.

A grammar is said to be ambiguous, if there's more than one way to derive the same string from the grammar. An example of an ambiguous CFG would be

$$\begin{aligned} \mathsf{S} &\rightarrow \mathsf{A}\ \mathsf{A} \\ \mathsf{A} &\rightarrow 1 \mid \epsilon \end{aligned}$$

which generates the language $\{\epsilon, 1, 11\}$ but has two ways of deriving the string 1 in that the 1 may be the expansion of either the first or the second $\mathsf{A}$.

A context-free grammar can easily become ambiguous, as seen in this highly simplifed expression grammar:

$$\mathsf{expr} \rightarrow \mathsf{number} \mid \mathsf{expr} + \mathsf{expr} \mid \mathsf{expr} * \mathsf{expr}$$

Here, ambiguity causes problems because the individual productions are assigned semantics.

A smaller problem with the above grammar is that the expression `1 + 2 + 3` can be parsed as either $(1 + 2) + 3$ or $1 + (2 + 3)$. Either way, we get the correct mathematical result, and we can ignore the superfluous parse by e.g. requiring the parser to proceed from left to right.[3]

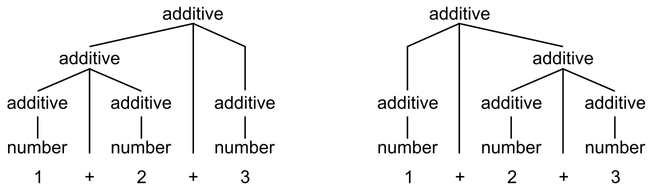

Figure 3: *The ambiguous grammar allows for multiple derivations.*

[3]This parser-level "hack" is usually employed to handle the *dangling else* problem of many programming languages.

Much worse is that the expression `1 + 2 * 3` can be parsed as either $(1+2) \cdot 3$ or $1 + (2 \cdot 3)$. In mathematics, we use operator precedence to remove this ambiguity, and we can encode this in the grammar:

$$\begin{array}{rcl} \mathsf{multiplicative} & \rightarrow & \mathsf{number} \quad | \quad \mathsf{number} * \mathsf{multiplicative} \\ \mathsf{additive} & \rightarrow & \mathsf{multiplicative} \quad | \quad \mathsf{multiplicative} + \mathsf{additive} \end{array}$$

In general, however, it might not be that easy. Any sufficiently complex grammar of natural language will almost certainly be ambiguous and thus non-deterministic, requiring the use of more complex parsers than e.g. those used in parsing programming languages.

# 5 Parsing

While many simple and highly efficient algorithms exist for parsing LL and LR grammars, parsing algorithms for arbitrary context free grammars are somewhat more complex.

Several algorithms require that the grammar be rewritten to Chomsky normal form (for the Cocke-Younger-Kasami algorithm) or Greibach normal form (after which the input can be recognized by a simple non-deterministic push-down automaton).

These algorithms are general since the required normalization can be performed on any CFG, without changing the recognized language. However, since we attach semantic meaning to the productions of our grammar, rewriting the grammar eases parsing at the cost of complicating the semantic analysis.

Instead, the Earley parser may be applied, which accepts any context-free grammar in Backus-Naur form.

## 5.1 A modified Earley parser

[Earley70] skims over the issue of constructing a parse tree, and the described algorithm contains unnecessary complexities (such as look-ahead[4]), as noted by later authors. The following formulation of a modified Earley parser is thus based not only on Earley's paper, but also later resources ([Aycock02] and [Dollin02]).

### Input and preparations

The parser takes as input:

- A set of terminals $a, b, \ldots$
- A set of nonterminals $A, B, \ldots$, of which one (denoted $R$) is the *root*
- A set of BNF productions $A \rightarrow \alpha$, where $\alpha$ is a string consisting of symbols from the two sets just mentioned
- An $n$ symbol string $X_0 \cdots X_{n-1}$ to be parsed

As a preparation, we choose a new (unused) terminal symbol $\dashv$ (the *terminator* and a new nonterminal $\phi$, which is promoted to root, and given the following production:

$$\phi \rightarrow R \dashv$$

$\dashv$ is appended to the input string as well, becoming $X_n$.

[4][Earley70] suggests a look-ahead of 1; [Aycock02] and [Dollin02] dispenses with look-ahead altogether.

## Parse trees

The goal of the parser is to construct one or more valid parse trees, describing ways to parse the input according to the given productions.

A parse tree is an ordered tree consisting of nodes, which may be either simple *terminal nodes* (without child nodes), or *production nodes* (which *may* have child nodes). Each node is assigned a value, which for terminal nodes is a terminal symbol $a$, and for production nodes is a production $A \to \alpha$.

A production node can thus also be seen as a tuple

$$\langle A \to \alpha, \langle c_1, c_2, \ldots, c_n \rangle \rangle$$

consisting of the production $A \to \alpha$ and the child nodes $c_1, c_2, \ldots, c_n$.

In the following, we use the short-hand notation $A(c_1 c_2 \cdots c_n)$ to denote such a production node.[5] This enables the following notation for whole parse trees:

$$\phi(E(a + E(a)))$$

## Parse states

Being a chart parser, the Earley parser constructs a number of parse states during execution.

These states are maintained in $n + 2$ *ordered* sets $S_0 \ldots S_{n+1}$: one set for every position between input symbols (as well as at the beginning and end of the input).

A parse state is a tuple $\langle P \to \alpha, p, i, T \rangle$, where $P \to \alpha$ is a production we're currently parsing, $p$ is our current position in the string $\alpha$ (the expansion of that production), $i$ denotes the input position at which we began parsing $P$, and $T$ is a partial parse tree, which may end up in the resulting parse tree(s).

As a shorthand, we write

$$P \to \alpha \,.\, \beta \quad i \quad T \tag{1}$$

to denote the state $\langle P \to \alpha\beta, p, i, T \rangle$ where the position $p$ is between $\alpha$ and $\beta$, as denoted by the dot.

In a state set $S_j$, the state (1) represents the following facts:

- We're currently testing whether input characters starting with $X_i$ can be derived from $P \to \alpha\beta$.
- $\alpha \overset{*}{\Rightarrow} X_i \cdots X_{j-1}$, that is, the input symbols $X_i$ through $X_{j-1}$ have been verified to match the part of the production before the dot, $\alpha$.

[5]For the sake of simplicity, this notation dispenses with the right-hand side of the production, $\alpha$. An actual implementation would usually need to track the whole production $A \to \alpha$.

- $T$ is a partial parse tree for a parse starting at $X_i$ with root $P$.

If the dot is at the end of the production, we further have that

- $P \overset{*}{\Rightarrow} X_i \cdots X_{j-1}$ (the input symbols $X_i$ through $X_{j-1}$ can be derived from $P$.)
- $T$ is a complete parse tree for a parse of $X_i$ through $X_{j-1}$ with root $P$.

At the end of the parse, the following statements are equivalent:

- The state is in $S_{n+1}$.
- The state is $\phi \to R \dashv \ . \quad 0 \quad \phi(T \dashv)$.
- $\phi(T \dashv)$ is a complete parse tree from a parse of the entire input (including $\dashv$) with root $\phi$, from which the relevant parse tree $T$ with root $R$ can trivially be extracted.

## Algorithm

The algorithm starts out by adding a single state to $S_0$,

$$\phi \to \ . \ R \dashv \quad 0 \quad \phi()$$

indicating that we're currently testing whether our whole input can be derived from our root $\phi$ and that no input symbols have been verified to match so far.

The algorithm then iterates over the state sets $S_0$ through $S_n$ in ascending order, and for each set $S_j$, the algorithm processes the states of $S_j$ in order.

Processing a state in $S_j$ may cause new states to be added to $S_j$; these must be processed as well, thus the requirement that the sets be ordered.

Depending on the state to be processed, one of three actions may be taken.

**The scanner** applies when the symbol following the dot is a terminal, $a$:

$$P \to \alpha \ . \ a \ \beta \quad i \quad T$$

We compare $a$ to the next input symbol $X_j$, and, if it's a match, adds the state

$$P \to \alpha \ a \ . \ \beta \quad i \quad T$$

to $S_{j+1}$, indicating that we successfully parsed the $a$.

**The predictor** applies when the symbol following the dot is a nonterminal, $A$:

$$P \to \alpha \ . \ A \ \beta \quad i \quad T$$

The predictor adds a state to $S_j$ for every production $A \rightarrow \gamma$ of that nonterminal,

$$A \rightarrow \ . \, \gamma \quad j \quad A()$$

Thus, we begin a check of whether $A$ matches the input at position $j$.

**The completer** applies when the dot is at the end of the production:

$$A \rightarrow \gamma \ . \quad i \quad T$$

This means that we successfully parsed an $A$ at position $i$, so we go back to $S_i$, and for every state which predicted $A$,

$$P \rightarrow \alpha \ . \ A \ \beta \quad i' \quad P(s)$$

we add a new state to $S_j$

$$P \rightarrow \alpha \ A \ . \ \beta \quad i' \quad P(s \ T)$$

indicating that we successfully parsed $A$, with the resulting parse tree $T$.

## Example

As an example, take the input $a + a$ and the following simple grammar:

$$\begin{array}{lcl} \phi & \rightarrow & E \dashv \\ E & \rightarrow & a \quad \mid \quad E + E \end{array}$$

After parsing completes, we will have produced the following state sets:

| | | | | |
|---|---|---|---|---|
| | | $\phi \rightarrow \ . \ E \ \dashv$ | 0 | $\phi()$ |
| $S_0$ | | $E \rightarrow \ . \ a$ | 0 | $E()$ |
| | | $E \rightarrow \ . \ E \ + \ E$ | 0 | $E()$ |
| | | $E \rightarrow a \ .$ | 0 | $E(a)$ |
| $S_1$ | $*$ | $\phi \rightarrow E \ . \ \dashv$ | 0 | $\phi(E(a))$ |
| | | $E \rightarrow E \ . \ + \ E$ | 0 | $E(E(a))$ |
| | | $E \rightarrow E \ + \ . \ E$ | 0 | $E(E(a)+)$ |
| $S_2$ | | $E \rightarrow \ . \ a$ | 2 | $E()$ |
| | $*$ | $E \rightarrow \ . \ E \ + \ E$ | 2 | $E()$ |
| | | $E \rightarrow a \ .$ | 2 | $E(a)$ |
| | | $E \rightarrow E \ + \ E \ .$ | 0 | $E(E(a) + E(a))$ |
| $S_3$ | $*$ | $E \rightarrow E \ . \ + \ E$ | 2 | $E(E(a))$ |
| | | $\phi \rightarrow E \ . \ \dashv$ | 0 | $\phi(E(E(a) + E(a)))$ |
| | $*$ | $E \rightarrow E \ . \ + \ E$ | 0 | $E(E(E(a) + E(a)))$ |
| $S_4$ | | $\phi \rightarrow E \ \dashv \ .$ | 0 | $\phi(E(E(a) + E(a)) \dashv)$ |

Blind alleys in the parsing process are marked with an $*$.

## 5.2 The epsilon problem

Earley briefly touches upon a problematic feature of many context-free grammars. Productions of the form $P \to \epsilon$, so-called epsilon productions, and more generally, any *nullable* nonterminal $E \overset{*}{\Rightarrow} \epsilon$.

Such a nullable nonterminal $E$ may be both predicted *and* completed in the same state set $S_j$, as no input symbols are scanned:

$$\begin{array}{lll} A \to\ .\, E & j & A() \\ A \to E\ . & j & A() \end{array}$$

Since we're still in the middle of processing $S_j$, we can't "go back" and iterate over all states of $S_j$, as a later predictor may yet add more states to $S_j$.

As a contrived example, take the input + and the grammar

$$\begin{array}{lll} \phi & \to & S \dashv \\ S & \to & E \ \mid \ P \\ P & \to & Q + \\ Q & \to & E \\ E & \to & \epsilon \end{array}$$

Clearly, $\phi(S(P(Q(E())+)))$ is a valid parse. However, the algorithm rejects the input if we're not careful. During execution, we end up with the following states in $S_0$,

$$\begin{array}{llllll} & & \phi & \to & .\, S \dashv & 0 & \phi() \\ * & (1) & S & \to & .\, E & 0 & S() \\ & & S & \to & .\, P & 0 & S() \\ & (2) & E & \to & . & 0 & E() \\ & (3) & P & \to & .\, Q + & 0 & P() \end{array}$$

just as we're about to process state (2). As the dot is at the end of the production, we run the completer, and finding that (1) is the only state with $E$ to the right of the dot, the following state is added to $S_0$:

$$\begin{array}{llllll} * & \quad & S & \to & E\ . & 0 & S(E()) \end{array}$$

However, then running the predictor on (3) results in a new state being added to $S_0$:

$$Q \quad \to \quad .\, E \quad 0 \quad Q()$$

This state too has $E$ to the right of the dot, but we missed it when we ran the completer earlier.

**Solutions**

[Aycock02] suggests that the parser keep track of the nullable productions, and pre-emptively complete them in the predictor step. This is certainly an excellent solution when implementing a recognizer, but there's no easy way to generate a valid parse tree, if nullable productions are simply skipped in this fashion.

A clean and simple solution, and the solution used by this author, is to alternate between running the predictor and completer, until neither has any more states to add to $S_j$. After this, the scanner may be run on all appropriate states in $S_j$ to construct $S_{j+1}$.

To speed this up, the predictor and completer are only run again if the predictor predicted a nullable production. (Which productions are nullable can trivially be determined before the parsing starts.)

## 6 Parser implementation

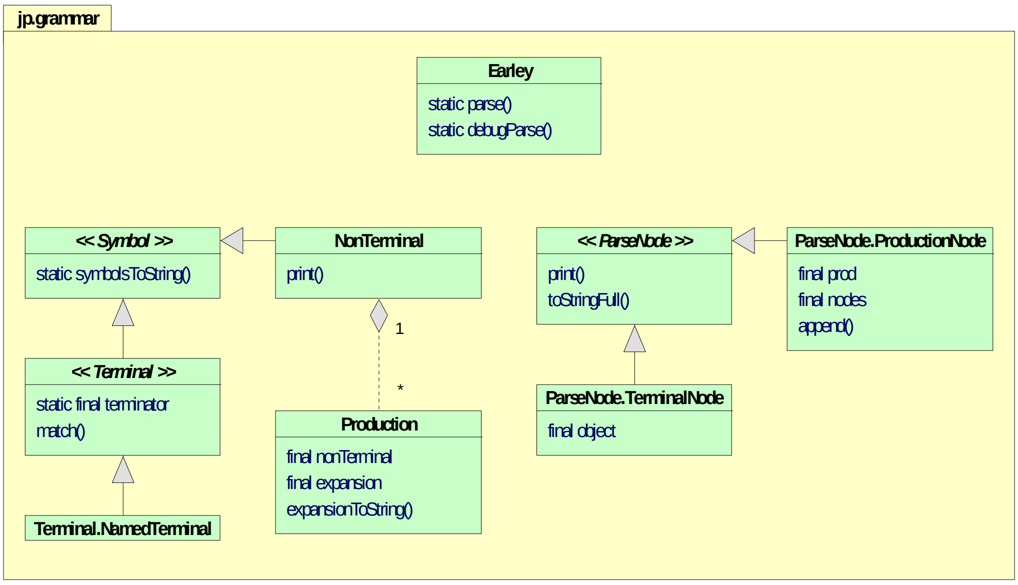

Figure 4: *The classes of the* `jp.grammar` *package.*

The modified Earley parser described above is implemented in the non-instantiable class `jp.grammar.Earley`, which provide two static methods:

```
static public Set<ParseNode.ProductionNode>
    parse(NonTerminal root, Object... input)

static public Set<ParseNode.ProductionNode>
    debugParse(NonTerminal root, Object... input)
```

`debugParse` is identical to `parse`, except that it also prints the generated parse states to standard output, for debugging.

The grammar used for parsing must be given in terms of `jp.grammar.Terminal`, `jp.grammar.NonTerminal` and `jp.grammar.Production` objects, which represent the corresponding mathematical objects.

A simple example of how to use the parser is given in `jp.ExprTest`, from which the following snippet is taken:

```
        Terminal a = new Terminal.NamedTerminal("a");
        Terminal plus = new Terminal.NamedTerminal("+");
        NonTerminal E = new NonTerminal("E");

        new Production(E, a);
        new Production(E, E, plus, E);

        Set<ParseNode.ProductionNode> result = Earley.debugParse(E, a, plus, a);

        System.out.println("\nParse trees:");
        for (ParseNode.ProductionNode l: result) l.print(0);
```

This implements the grammar described in the example of section 5.1 (page 16).

In the code above, `debugParse` is used to display the internal parse states, and the returned parse trees are simply printed. (A typical application would instead have to interpret the parse trees.)

## 6.1 Grammar compiler

A complex context-free grammar converted to Java code isn't a pretty sight, and rather tiresome to write. Instead, the `jp.GC` grammar compiler may be used to convert a grammar source file written in a BNF-like syntax.

The grammar compiler accepts the file names of one or more grammar specifications, and outputs the corresponding Java code to standard output.

The syntax is quite simple. Each line describes one or more alternatives for a given nonterminal, using the following syntax:

$$\begin{array}{rcl} \mathsf{line} & \rightarrow & \mathsf{nonterminal}\ \text{“}\mathsf{=}\text{”}\ \mathsf{symbolString\ alternatives} \\ \mathsf{alternatives} & \rightarrow & \epsilon \\ & | & \text{“}\mathsf{|}\text{”}\ \mathsf{symbolString\ alternatives} \\ \mathsf{symbolString} & \rightarrow & \epsilon \\ & | & \mathsf{symbol\ symbolString} \end{array}$$

Here, nonterminal and symbol are arbitrary Java identifiers. The generated code will define local variables for the nonterminals. Presumably, symbols refer to either nonterminals defined by the grammar, or terminals defined by the host application; the grammar compiler does not verify their validity.

On all lines containing the comment symbol #, # and everything following it is ignored. Blank lines are ignored as well.

A line may be continued by indenting the following lines by one or more whitespace characters.

The grammar syntax can describe itself as follows:[6]

```
line = symbol EQUAL symbolString alternatives

alternatives = # epsilon
             | PIPE symbolString alternatives

symbolString = # epsilon
             | symbol symbolString
```

(Here, `EQUAL` and `PIPE` must be defined by the host application to represent the equal sign and pipe symbol respectively.) The result is shown in figure 6.1.

A larger example can be found in appendix B.5, and the generated code in appendix B.4.

```
  NonTerminal symbolString = new NonTerminal("symbolString");
  NonTerminal line = new NonTerminal("line");
  NonTerminal alternatives = new NonTerminal("alternatives");
  new Production(symbolString);
  new Production(symbolString, symbol, symbolString);
  new Production(line, symbol, EQUAL, symbolString, alternatives);
  new Production(alternatives);
  new Production(alternatives, PIPE, symbolString, alternatives);
```

Figure 5: *The Java code generated by the grammar compiler*

Since Java does not support include files, and since the generated code references symbols defined by the host application, the generated code must be pasted into the appropriate place in the host application source code, either manually or using some external tool to merge the compiler output into the host application code. The `gram` target of this project's makefile uses `sed` for this purpose.

[6]The parser of the grammar compiler is not actually implemented using the `jp.grammar` package, but instead employs a handwritten non-recursive parser.

# 7 Semantics

Semantics is the study of meaning and understanding, a field in which philosophy and science intersects. In natural language processing, *semantic analysis* translates (ambiguous) natural language into simpler, more well-defined (usually non-ambiguous) elements.

A central notion in semantics is the basic unit of knowledge, which I'll refer to as *idea*, a word used by John Locke to denote "whatever is the object of understanding when a man thinks" in his *Essay Concerning Human Understanding*. [Locke90] Ideas thus include all things, whether concrete physical objects (e.g. the Eiffel tower), physical properties (the color purple), abstract classes (mammals), as well as abstract concepts (good and evil).

[Locke90] also puts emphasis on the duality of knowledge and language, suggesting the conclusion that humans cannot think what they cannot put into words (nor, obviously, put into words what they cannot think).[7]

This serves as a caution against straying too far from the original natural language when doing the semantic processing.

## 7.1 Meanings of "to be"

In implementing semantics, it's important to recognize the different meanings of "to be" ("is the same as", "is a subset of" and "is characterized by"), as demonstrated in the following sentences:

*Cats are felines.* "cats" and "felines" are the same; both refer to animals of the *Felidae* family. (Although "cat" can also refer specifically to the domestic cat, a person, and a number of other things.) We say that cat = mammal.

*Cats are mammals.* Cats are a proper subset of mammals. We say that cat is-a mammal.

*Cats are cute.* Cats are characterized by being cute. This paper will use the notation cat is cute.

## 7.2 Semantic networks using is-a

A semantic network is a directed graph, in which each node represents an idea, and the edges are binary relations, typically is-a-relations.

Each idea is further assigned a number of properties. Although some semantic network implementations make a distinction between properties and nodes, in the

[7]As Wittgenstein put it in his 1922 *Tractatus Logico-Philosophicus*: "The limits of my language mean the limits of my world." Also, compare Orwell's *Nineteen Eighty-Four*, in which the goal of the *Newspeak* language is that only loyal thoughts be expressible, and thus, thinkable.

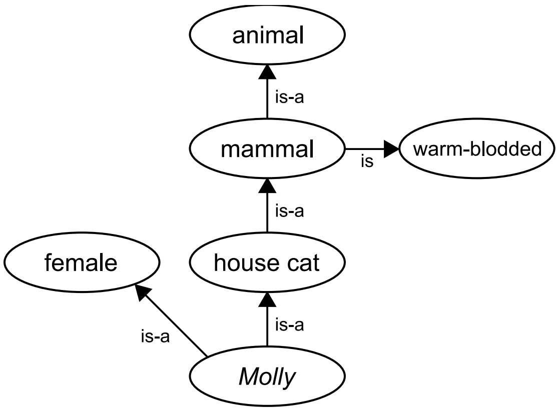

Figure 6: *A semantic network*

general case, properties (like "warm-blooded" in figure 6) are just ideas, linked with a different type of relation (here denoted is). This way, the semantic network can also make assertions about properties, and properties may be assigned other properties.

The interesting feature of is-a is that it enables properties to be inherited. Once we know that mammal is warm-blooded and Molly is-a house-cat is-a mammal, we can assume that Molly is warm-blooded.

## 7.3 Cancellation

Property inheritance, however, is only an assumption. The classic example is bird is capable-of-flight, which is usually true, but not always, e.g. in the case of penguins.

To encode this in a semantic network, one explicitly asserts the opposite (penguin is-not capable-of-flight), thus *cancelling* inheritance of the capable-of-flight property.

## 7.4 The duck test

The is-a inheritance can also be used in reverse, to determine the class of an idea. For instance, assuming that all we know about Tom is that Tom is warm-blooded, we might then assume that Tom is-a mammal.

Each is-a relation is thus accompanied by an implicit reverse relation, which this paper will refer to as "implies".

[Brachman83] correctly argues that just because an entity has certain properties, all of which are typical for a certain class, that entity is not necessarily a member of that class.

However, inductive reasoning prescribes the opposite: If an entity has all or some of the known properties of a class, and these properties are unique to that class, we must assume that entity to belong to that class, at least until new information comes to light and suggests otherwise.

This boils down to an old philosophic discussion about the *identity of indiscernibles*, aptly formulated by James W. Riley in an oft-quoted poem:

> When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

On a mathematical level, the identity of indiscernibles is still debated, but in a physical reality, it's an indispensable rule, and humans certainly apply this so-called *duck test* all the time to determine identities.

## 7.5 Uncertainty

[Brachman83] has a point though: our assumption about Tom are based on a fragile premise: that no other animal (or generally, no other idea) could be warm-blooded. Indeed, Tom might turn out to be another warm-blooded animal, such as a white shark. To deal with this, relations in the semantic network can be assigned a numerical level of confidence.

To gauge the confidence of "warm-blooded implies mammal", however, we must assess the probability of new, contradictory information (such that the fact that non-mammals may be warm-blooded) appearing, which is of course difficult. 50% may be as good as any other value.

The remaining 50% must then be divided between mammal and white shark, recognizing that

- We know of one mammal, but of no white sharks, which arguably makes mammal more likely than shark.
- We don't know if the class of mammals and the class of white sharks overlap. Tom could be both.

Again, the best we can do is a wild guess.

Another problem is that "warm-blooded" isn't a scientifically well-defined term, and some may thus disagree with the claim that white sharks are warm-blooded. We may indicate this by assigning a confidence of (say) 50% to the white-shark is warm-blooded relation. Mammals, on the other hand, are most certainly warm-blooded.

The result is a semantic network looking like figure 7. At this point, all we need to do is to apply a feed-back mechanism that adjusts the confidence levels in response to user input, and we'll essentially have a neural network.

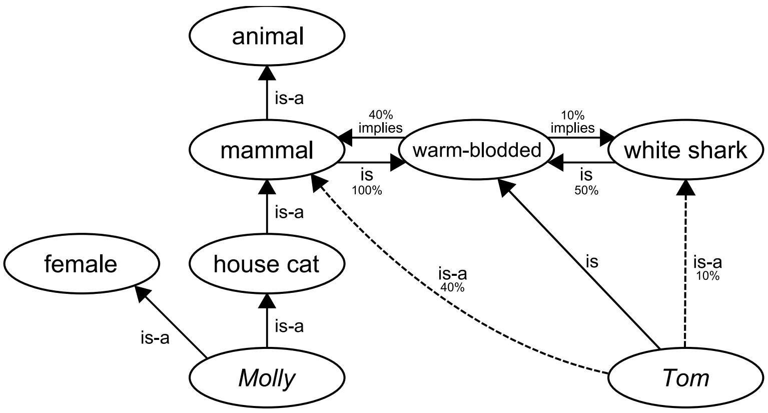

Figure 7: *The updated semantic network*

## 7.6 Individuals versus generics

[Brachman83] argues that one should treat *generics* (sets, e.g. humans) and *individuals* (set members, e.g. John) differently. This leads to a distinction between two kinds of is-a relationships, namely $\subseteq$ and $\in$.

But for many purposes, treating individuals as single-member sets is a usuable abstraction, such that only $\subseteq$ remains:

$$\mathsf{John} = \{john\} \subseteq \mathsf{humans}$$

In this representation, the value *john* is never directly referred to. As such we cannot enumerate the known elements of the set humans, but we *can* enumerate the known *subsets* (which include John).

## 7.7 Meta-semantics

Take the following sentence, "A person is either male or female." The sentence carries an implicit "but not both", which can be expressed using predicate logic as such ($\forall p$):

$$\mathsf{person}(p) \Rightarrow (\mathsf{male}(p) \vee \mathsf{female}(p)) \wedge \overline{\mathsf{male}(p) \wedge \mathsf{female}(p)}$$

– or equivalently using set logic:

$$\mathsf{persons} \subseteq (\mathsf{males} \cup \mathsf{females}) \cap \overline{\mathsf{males} \cap \mathsf{females}}$$

The similar sentence, if interpreted literally, "A person can be either male or female." is more vague. The use of "either" still carries an implicit "but not both", but the exchange of "is" for "can be" means that a person *also* could be something else entirely. The sentence thus boils down to "A person cannot *both* be male and female".

$$\mathsf{person}(p) \Rightarrow \overline{\mathsf{male}(p) \wedge \mathsf{female}(p)}$$

Finally, "A person can be male or female." dismisses with both "either" and "is", reducing the sentence to the tautology

$$\mathsf{person}(p) \Rightarrow \mathsf{male}(p) \vee \mathsf{female}(p) \vee \mathsf{true}$$

The sentence is thus devoid of semantics, but not *meta-semantics*: It tells us something very important, namely that some persons *may* possess the properties male and female.

Such semantics may be described using a "can-be" relation in the semantic network. For a general solution, a second-order predicate $\mathrm{can}(P, x, y)$ may instead be introduced, where $P$ is any first-order relation ("is", "is-a", "implies", etc.), $x$ the class of ideas that the relation *could* apply to, and $y$ the associated idea. For instance:

$$\mathrm{can}(\mathrm{is}, \mathsf{person}, \mathsf{female})$$

Note that in order to represent second-order predicates in a semantic network, the different types of relations must be represented as ideas as well. To represent *tertiary* relations like the above, it may be necessary to have *relation instances* (like person is female) represented as ideas as well. At this point, it becomes clear that the semantic network is reasoning about its own reasoning.

# 8 Challenges in the English grammar

The grammar of natural language is immensely complex, and hard to pin down as well. As [Winograd71] says, "the task of codifying an entire language in any formalism is so large that it would be folly to try in the course of a single research project."

Different grammars may accept the same sentences, but result in different parse trees, allowing different levels of insight into the semantics of the sentence.

Take the simple sentence "John ate the apple." Classic English grammar renders this as as sentence consisting of a noun phrase ("John") and verb phrase or *predicate* ("ate the apple").

A *systemic grammar*, as for instance employed by SHRDLU, would render the sentence as a clause consisting of a noun group ("John"), a verb group ("ate") and another noun group ("the apple"). This is illustrated in figure 8.

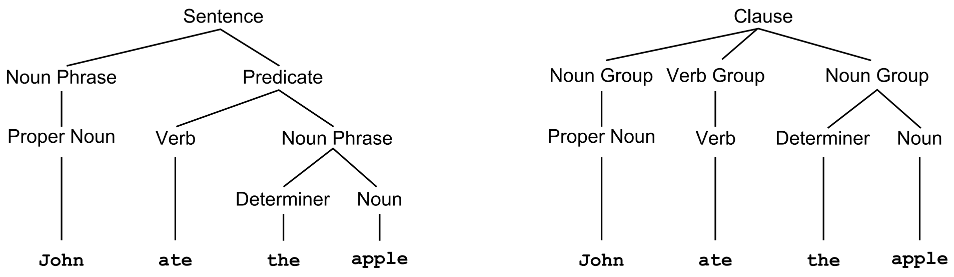

Figure 8: *Same sentence, different grammars*

## 8.1 Participial ambiguity

One difficult challenge is sentences that can only be interpreted in context of other sentences. [Allen95] gives the following example of this, in the form of a participial ambiguity:

```
Visiting relatives can be trying.
```

"Visiting" is the present participle of VISIT$_V$; however, "Visiting relatives" is a noun phrase that can be interpreted in two ways:

| Participal use | Interpretation | Head | Number |
|---|---|---|---|
| Gerund | the act of visiting relatives | "visiting" | singular |
| Verbal adjective | relatives that are visiting | "relatives" | plural |

As can be seen, the two interpretations have different heads and grammatical number, which would have allowed disambiguation had the sentence been `Visiting relatives are trying.` or `Visiting relatives is trying.`.

The first sentence remains ambiguous because “can be” does not indicate number.

### 8.2 Phrasal verbs

Phrasal verbs are compounds consisting of a verb together with one or more particles. Phrasal verbs are grammatical exceptions, in that they do not simply denote that the verb is modified by the particles, but have an entirely new meaning.

An example is the intransitive phrasal verb “give up”, which has nothing to do with either giving nor the upwards direction. Additionally, the particle is fixed for this particular meaning of the verb; the syntactically equivalent “give down” is meaningless, for instance.

As such, “give”, “give up”, “give in”, “give over”, etc. are all different verbs, and all but the first are phrasal verbs.

One way to deal with phrasal verbs is to incorporate them into the grammar, but since a goal is often to design a simple, unchanging grammar (with only the lexicon growing over time), this is less than ideal.

# 9 Conclusion

The Earley parser is a simple and efficient parser for dealing with arbitrary context-free grammars, although Earley’s original formulation has since been improved upon, as detailed in this paper. However, parsing is only the first step a program must take to gain understanding of natural language, and models of semantics and knowledge representation (such as semantic networks) have yet to reach a level of maturity, where they may be used for general purpose natural language processing.

Computers will not understand natural language any day soon. Not only is the rules of natural language incredibly complex, possibly more complex than today’s computers can handle, but they’re indeed too complex for even humans to fully comprehend them, cf. the continuing research into which kind of grammar is more suitable.

Still, even if they are few and far between, a number of projects have demonstrated that natural language may be viable for highly domain-specific programming and data entry, cf. SHRDLU and Inform 7, just like artificial intelligence can compete with humans in specific domains, such as chess.

## A References

[Allen95] James Allen. *Natural language understanding.* 2nd edition, 1995.

[Aycock02] John Aycock and R. Nigel Horspool. *Practical Earley parsing.* The Computer Journal, 45, no. 6; 620–630, 2002.

[Brachman83] Ronald J. Brachman. *What IS-A is and isn't: An analysis of taxonomic links in semantic networks.* IEEE Computer, 16, no. 10; 30–36, 1983.

[Codd70] Edgar F. Codd. *A relational model of data for large shared data banks.* Communications of the ACM, 13, no. 6; 377–387, Jun 1970.

[Dollin02] Chris Dollin. *Earley parser.* comp.programming, Apr 2002.

[Earley70] Jay C. Earley. *An efficient context-free parsing algorithm.* Communications of the ACM, 13, no. 2; 94–102, Feb 1970.

[Elder05] Murray Elder. *G-automata, counter languages and the Chomsky hierarchy.* London Mathematical Society Lecture Note Series no. 339, 2005.

[Locke90] John Locke. *An Essay Concerning Human Understanding*, 1690.

[Nelson05] Graham Nelson. *Natural language, semantic analysis and interactive fiction*, 2005.

[Trask04] Larry Trask. *What is a word?*, Jan 2004.

[Winograd71] Terry Winograd. *Procedures as a representation for data in a computer program for understanding natural language*, 1971.

# B Source

## B.1 jp.util package

The `jp.util` package contains two utility classes, one (`MultiMap`) serving as the bass for `jp.lexer.Lexicon`, the other (`TablePrint`) provides pretty debug output for `jp.grammar.Earley`.

### jp.util.MultiMap

```java
package jp.util;

import java.util.*;

public class MultiMap<K, V>
{
    private Map<K, Set<V>> map = new HashMap<K, Set<V>>();

    private class MultiMapEntry<K, V> implements Map.Entry<K, V>
    {
        private final K key;
        private final V value;

        public MultiMapEntry(K key, V value)
        {
            this.key = key;
            this.value = value;
        }

        /** Compares the specified object with this entry for equality. */
        public boolean equals(Object o)
        {
            if (!(o instanceof Map.Entry)) return false;

            Map.Entry entry = (Map.Entry) o;
            return (entry.getKey() == key && entry.getValue() == value);
        }

        /** Returns the key corresponding to this entry. */
        public K getKey() { return key; }

        /** Returns the value corresponding to this entry. */
        public V getValue() { return value; }

        /** Returns the hash code value for this map entry. */
        public int hashCode() { return key.hashCode() ^ value.hashCode(); }

        /**
         *  Replaces the value corresponding to this entry with the specified
         *  value (optional operation).
         */
        public V setValue(V value)
        {
            throw new UnsupportedOperationException();
        }
    }

    /** Removes all mappings from this map. */
    public void clear() { map.clear(); }

    /** Returns true if this map contains a mapping for the specified key. */
    public boolean containsKey(Object key) { return map.containsKey(key); }

    /** Returns true if this map maps one or more keys to the specified value.*/
    public boolean containsValue(Object value)
    {
        for (Set<V> values: map.values())
            if (values.contains(value)) return true;

        return false;
    }

    /** Returns a set view of the mappings contained in this map. */
    public Set<Map.Entry<K,V>> entrySet()
    {
        Set<Map.Entry<K,V>> result = new HashSet<Map.Entry<K,V>>();
        for (Map.Entry<K,Set<V>> entry: map.entrySet())
        {
            for (V v: entry.getValue())
                result.add(new MultiMapEntry<K, V>(entry.getKey(), v));
        }
        return result;
    }

    /** Compares the specified object with this map for equality. */
    public boolean equals(Object o)
    {
        if (!(o instanceof MultiMap)) return false;
        return map.equals(((MultiMap) o).map);
    }

    /** Returns the values to which this map maps the specified key. */
    @SuppressWarnings("unchecked")
    public Set<V> get(Object key)
    {
        Set<V> c = map.get(key);
        if (c == null) return Collections.EMPTY_SET; // unchecked
        return Collections.unmodifiableSet(c);
    }

    /** Returns the hash code value for this map. */
    public int hashCode() { return map.hashCode(); }

    /** Returns true if this map contains no key-value mappings. */
    public boolean isEmpty() { return map.isEmpty(); }

    /** Returns a set view of the keys contained in this map. */
    public Set<K> keySet() { return map.keySet(); }

    /** Associates the specified value with the specified key in this map. */
    public void put(K key, V value)
    {
        Set<V> set = map.get(key);
        if (set == null) map.put(key, set = new HashSet<V>());
        set.add(value);
    }

    /** Associates the specified values with the specified key in this map. */
    public void put(K key, Collection<V> values)
    {
        Set<V> set = map.get(key);
        if (set == null) map.put(key, set = new HashSet<V>());
        set.addAll(values);
    }

    /** Copies all of the mappings from the specified map to this map. */
    public void putAll(Map<? extends K,? extends V> t)
    {
        for (Map.Entry<? extends K,? extends V> entry: t.entrySet())
            put(entry.getKey(), entry.getValue());
    }

    /** Removes the mapping for this key from this map if it is present. */
    public void remove(Object key) { map.remove(key); }

    /** Returns the number of key mappings in this map. */
    public int size() { return map.size(); }

    /**  Returns a set view of the values contained in this map. */
    public Set<V> values()
    {
        Set<V> result = new HashSet<V>();

        for (Map.Entry<K,Set<V>> entry: map.entrySet())
            result.addAll(entry.getValue());

        return result;
    }
}
```

## jp.util.TablePrint

```
package jp.util;

import java.util.*;

public class TablePrint
{
    private List<String[]> rows = new ArrayList<String[]>();
    private int[] columnWidths;
    private int cSpace;

    public TablePrint(int columnCount, int cSpace)
    {
        columnWidths = new int[columnCount];
        this.cSpace = cSpace;
    }

    public void addRow(String... row)
    {
        assert(row.length <= columnWidths.length);
        rows.add(row);
        for (int i = 0; i < row.length; i++)
        {
            if (columnWidths[i] < row[i].length())
                columnWidths[i] = row[i].length();
        }
    }

    public void addRow(Object... row)
    {
        assert(row.length <= columnWidths.length);
        String[] strs = new String[row.length];

        for (int i = 0; i < row.length; i++) strs[i] = row[i].toString();
        addRow(strs);
    }

    public void print()
    {
        for (String[] row: rows)
        {
            for (int i = 0; i < row.length; i++)
            {
                System.out.print(row[i]);
                for (int j = row[i].length(); j < columnWidths[i] + cSpace; j++)
                    System.out.print(" ");
            }
            System.out.println();
        }
    }
}
```

## B.2 jp.grammar package

The `jp.grammar` package contains classes for managing a context-free grammar and an implementation of a modified Earley parser, as described in section 6. See also the class diagram on page 18.

### jp.grammar.Earley

```java
package jp.grammar;

import java.util.*;
import jp.util.TablePrint;

public class Earley
{
    private Earley() { }

    private static class ParseState
    {
        Production prod; // production
        int prodPos;     // position in the expansion of the production
        int inputPos;    // position in input of the beginning of the production
        ParseNode.ProductionNode parse;

        public boolean equals(Object o)
        {
            ParseState s = (ParseState) o;
            return s.prod == prod && s.prodPos == prodPos
                && s.inputPos == inputPos && s.parse.equals(parse);
        }

        ParseState(Production prod, int prodPos, int inputPos,
            ParseNode.ProductionNode parse)
        {
            this.prod = prod;
            this.prodPos = prodPos;
            this.inputPos = inputPos;
            this.parse = parse;
        }

        public String toString()
        {
            return prod.nonTerminal + " -> " +
                   prod.expansionToString(prodPos) +
                   "  " + inputPos;
        }

        boolean hasNextSymbol()
        {
            return (prodPos < prod.expansion.length);
        }

        Symbol nextSymbol()
        {
            return hasNextSymbol() ? prod.expansion[prodPos] : null;
        }

        // "Move the dot", that is, increase the prodPos.
        ParseState moveOver(ParseNode.ProductionNode parse)
        {
            return new ParseState(prod, prodPos + 1, inputPos, parse);
        }
    }

    // An ordered set, and thus not an ordinary Java Set.
    private static class StateSet extends AbstractCollection<ParseState>
    {
        private List<ParseState> set = new ArrayList<ParseState>();

        // For debugging
        public void print()
        {
            TablePrint tbl = new TablePrint(4, 2);

            for (ParseState ps: set)
            {
                tbl.addRow(ps.prod.nonTerminal,
                    " -> " + ps.prod.expansionToString(ps.prodPos),
                    ps.inputPos,
                    ps.parse.toStringFull());
            }
            tbl.print();
        }

        public Iterator<ParseState> iterator() { return set.iterator(); }

        public boolean add(ParseState elm)
        {
            if (set.contains(elm)) return false;
            set.add(elm);
            return true;
        }

        public ParseState get(int i) { return set.get(i); }

        public int size() { return set.size(); }
    }

    static private Set<ParseNode.ProductionNode> parseInternal(
        NonTerminal root, Object[] input_,
        Production phiProduction, StateSet[] stateSets)
    {
        int inputLength = input_.length; // 'n' in Earley70

        // Append a terminator to the input.
        Object[] input = new Object[input_.length + 1];
        System.arraycopy(input_, 0, input, 0, input_.length);
        input[input_.length] = Terminal.terminator;

        stateSets[0].add(new ParseState(phiProduction, 0, 0,
                new ParseNode.ProductionNode(phiProduction)));

        for (int inputPos = 0; inputPos <= inputLength; inputPos++)
        {
            StateSet set = stateSets[inputPos];

            int oldSetSize = set.size();
            boolean runAgain = false;

            // Complete the set S[inputPos].
            for (int i = 0; i < set.size(); i++)
            {
                ParseState state = set.get(i);

                // If no next symbol in prod: Do the COMPLETER operation.
                if (!state.hasNextSymbol())
                {
                    // Look at the input pos where we entered this parsestate.
                    // (Might be the current set.)
                    StateSet beginSet = stateSets[state.inputPos];
                    for (int j = 0; j < beginSet.size(); j++)
                    {
                        ParseState ps = beginSet.get(j);
                        if (state.prod.nonTerminal != ps.nextSymbol()) continue;

                        set.add(ps.moveOver(ps.parse.append(state.parse)));
                    }
                }

                // If next symbol is a NonTerminal: Do the PREDICTOR operation.
                else if (state.nextSymbol() instanceof NonTerminal)
                {
                    NonTerminal nt = (NonTerminal) state.nextSymbol();
                    if (nt.isNullable)
                    {
                        // This non-terminal may expand to the empty string.
                        // We thus may need to run the completer on previous
                        // entries in the set.
                        runAgain = true;
                    }

                    for (Production prod: nt.alternatives)
                    {
                        ParseState s = new ParseState(prod, 0, inputPos,
                            new ParseNode.ProductionNode(prod));
                        set.add(s);
                    }
                }
                else assert(state.nextSymbol() instanceof Terminal);
            }

            if (runAgain && set.size() > oldSetSize)
            {
                // Run completer/predictor again on all entries in the set.
                inputPos--;
                continue;
            }

            // Construct the set S[inputPos + 1].
            // We can use foreach here, because the current set doesn't change.
            for (ParseState state: set)
            {
                // If the next symbol is a Terminal: Do the SCANNER operation.
                if (state.nextSymbol() instanceof Terminal)
                {
                    Terminal t = (Terminal) state.nextSymbol();
                    ParseNode.TerminalNode node = t.match(input[inputPos]);
                    if (node != null)
                    {
                        ParseState ps = state.moveOver(
                            state.parse.append(node));

                        stateSets[inputPos + 1].add(ps);
                    }
                }
            }
        }

        Set<ParseNode.ProductionNode> result
            = new HashSet<ParseNode.ProductionNode>();

        StateSet finalSet = stateSets[inputLength + 1];
        for (ParseState ps: finalSet)
        {
            assert(ps.prod == phiProduction && ps.prodPos == 2 &&
                ps.inputPos == 0);
            result.add((ParseNode.ProductionNode) ps.parse.nodes[0]);
        }

        return result;
    }

    static public Set<ParseNode.ProductionNode>
        parse(NonTerminal root, Object... input)
    {
        StateSet[] stateSets = new StateSet[input.length + 2];
        for (int i = 0; i < stateSets.length; i++)
            stateSets[i] = new StateSet();

        NonTerminal phi = new NonTerminal("#");
        Production phiProduction = new Production(
            phi, root, Terminal.terminator);

        phi.initialize(new HashSet<Symbol>());

        return parseInternal(
            root, input, phiProduction, stateSets);
    }

    static public Set<ParseNode.ProductionNode>
        debugParse(NonTerminal root, Object... input)
    {
        StateSet[] stateSets = new StateSet[input.length + 2];
        for (int i = 0; i < stateSets.length; i++)
            stateSets[i] = new StateSet();

        NonTerminal phi = new NonTerminal("#");
        Production phiProduction = new Production(
            phi, root, Terminal.terminator);

        phi.initialize(new HashSet<Symbol>());

        Set<ParseNode.ProductionNode> result = parseInternal(
            root, input, phiProduction, stateSets);

        for (int i = 0; i < stateSets.length; i++)
        {
            System.out.println("============ S[" + i + "] ============");
            stateSets[i].print();
        }

        return result;
    }
}
```

## jp.grammar.NonTerminal

```
package jp.grammar;

import java.util.*;

/**
 *  A NonTerminal is a symbol in the grammar that may match a varying number of
 *  symbols in an input string, depending on its associated alternatives (a list
 *  of productions.)
 */
public class NonTerminal extends Symbol
{
    private final String name;

    /* package private */ List<Production> alternatives
        = new ArrayList<Production>();

    public NonTerminal(String name) { this.name = name; }

    public String toString() { return name; }

    public void print()
    {
        for (Production p: alternatives)
        {
            System.out.println(p);
        }
    }

    /* package private */ void initialize(Set<Symbol> done)
    {
        if (done.contains(this)) return;
        done.add(this);

        assert(alternatives.size() > 0) : name + " has no alternatives";
        isNullable = false;

        for (Production p: alternatives)
        {
            boolean thisAlternativeIsNullable = true;

            for (Symbol s: p.expansion)
            {
                s.initialize(done);
                if (!s.isNullable) thisAlternativeIsNullable = false;
            }
            if (thisAlternativeIsNullable) isNullable = true;
        }
    }
}
```

## jp.grammar.ParseNode

```
package jp.grammar;

import java.util.*;

abstract public class ParseNode
{
    public void print(int indent)
    {
        while (indent-- > 0) System.out.print(" ");
        System.out.println(this);
    }

    public String toStringFull() { return toString(); }

    static public class TerminalNode extends ParseNode
    {
        public final Object symbol;

        public TerminalNode(Object t) { symbol = t; }

        public String toString() { return symbol.toString(); }
    }

    static public class ProductionNode extends ParseNode
    {
        public final ParseNode[] nodes;

        public final Production prod;

        public ProductionNode(Production prod)
        {
            this.prod = prod;
            this.nodes = new ParseNode[0];
        }

        private ProductionNode(Production p, ParseNode[] nodes, ParseNode extra)
        {
            this.prod = p;
            this.nodes = new ParseNode[nodes.length + 1];
            System.arraycopy(nodes, 0, this.nodes, 0, nodes.length);
            this.nodes[nodes.length] = extra;
        }

        public boolean equals(Object o)
        {
            ParseNode.ProductionNode pnl = (ParseNode.ProductionNode) o;
            return pnl.prod == prod && Arrays.equals(pnl.nodes, nodes);
        }

        public ParseNode.ProductionNode append(ParseNode node)
        {
            return new ProductionNode(prod, nodes, node);
        }

        public void print(int indent)
        {
            super.print(indent);

            indent += 2;
            for (ParseNode n: nodes) n.print(indent);
        }

        public String toString() { return prod.toString(); }

        public String toStringFull()
        {
            StringBuilder sb = new StringBuilder();
            sb.append(prod.nonTerminal.toString());
            sb.append("(");
            for (ParseNode node: nodes)
            {
                sb.append(" ");
                sb.append(node.toStringFull());
                sb.append(" ");
            }
            sb.append(")");
            return sb.toString();
        }
    }
}
```

## jp.grammar.Production

```
package jp.grammar;

import java.util.Set;

/**
 *  A Production is a single expansion of an associated nonterminal.
 */
public class Production
{
    public final NonTerminal nonTerminal;
    public final Symbol[] expansion;

    public Production(NonTerminal nonTerminal, Symbol... expansion)
    {
        assert(nonTerminal != null);
        this.nonTerminal = nonTerminal;
        this.expansion = expansion;

        nonTerminal.alternatives.add(this);
    }

    public String toString()
    {
        return nonTerminal + " -> " + Symbol.symbolsToString(expansion);
    }

    public String expansionToString(int dotPos)
    {
        String result = "";

        if (dotPos == 0) result += ".";
        else result += " ";

        int i = 0;
        for (Symbol sym: expansion)
        {
            result += sym;
            if (++i == dotPos) result += ".";
            else result += " ";
        }

        return result;
    }
}
```

## jp.grammar.Symbol

```
package jp.grammar;

import java.util.Set;

abstract public class Symbol
{
    public boolean isNullable = false;

    /** Recursively initializes symbols (determine isNullable) before parse. */
    /* package private */ void initialize(Set<Symbol> done) { }

    /** Converts an array of symbols to a string, for debug purposes. */
    public static String symbolsToString(Symbol[] symbols)
    {
        String result = "";
        for (Symbol sym: symbols)
        {
            if (result.length() > 0) result += " ";
            result += sym;
        }
        return result;
    }
}
```

## jp.grammar.Terminal

```
package jp.grammar;

import java.util.Set;

/**
 *  A Terminal is a symbol in the grammar that may match exactly one symbol in
 *  an input string. Typically, a Terminal will match a specific input symbol,
 *  but it may also match a group of input symbols (e.g. any letter.)
 */
abstract public class Terminal extends Symbol
{
    /** A pre-defined NamedTerminal with name "|-" */
    static final public Terminal terminator = new NamedTerminal("|-");

    /** A NamedTerminal has a name and matches itself */
    static public class NamedTerminal extends Terminal
    {
        private final String name;

        public NamedTerminal(String name) { this.name = name; }

        public String toString() { return name; }
    }

    /**
     *  The match method of a Terminal determines if the terminal matches
     *  the argument. If so, it returns a ParseNode.TerminalNode, otherwise
     *  it returns null.
     *  By default, we match ourselves.
     */
    public ParseNode.TerminalNode match(Object t)
    {
        if (t == this) return new ParseNode.TerminalNode(t);
        return null;
    }
}
```

## B.3 jp.lexer package

The `jp.lexer` package contains classes related to lexical analysis, including a tokenizer and classes for dealing with lexicons, lexemes, word classes and grammatical features.

### jp.lexer.ClassResolver

```
package jp.lexer;

import jp.grammar.*;

import java.util.*;

/**
 *  Interface used by ClassTerminal to test if a word belongs to a certain word
 *  class.
 *  The StaticCR implementation naively looks the word up in the lexicon.
 *  LenientCR simply matches if the word is all-lowercase, and if so, adds the
 *  word to the lexicon.
 */
public interface ClassResolver
{
    public Object resolve(String word);

    public static class StaticCR implements ClassResolver
    {
        public final WordClass wordClass;
        public final Lexicon lexicon;

        public StaticCR(Lexicon lexicon, WordClass wordClass)
        {
            this.lexicon = lexicon;
            this.wordClass = wordClass;
        }

        public Object resolve(String word)
        {
            Set<Lexeme> lexemes = lexicon.getLexemes(word, wordClass);
            if (lexemes.size() > 0) return lexemes;
            return null;
        }

        public String toString()
        {
            return "<" + wordClass.toString() + ">";
        }
    }

    public static class LenientCR extends StaticCR
    {
        public LenientCR(Lexicon lexicon, WordClass wordClass)
        {
            super(lexicon, wordClass);
        }

        public Object resolve(String word)
        {
            for (int i = 0; i < word.length(); i++)
                if (Character.isUpperCase(word.charAt(i))) return null;

            return lexicon.addLexeme(word, wordClass);
        }
    }
}
```

## jp.lexer.ClassTerminal

```java
package jp.lexer;

import jp.grammar.*;

import java.util.*;

/**
 * A CFG terminal matching a whole class of words (e.g. all verbs).
 * The actual test is performed by a ClassResolver.
 */
public class ClassTerminal extends Terminal
{
    public ClassResolver resolver;

    public ClassTerminal() { }

    public ParseNode.TerminalNode match(Object t)
    {
        if (t instanceof String)
        {
            Object o = resolver.resolve((String) t);
            if (o != null) return new ParseNode.TerminalNode(o);
        }
        return null;
    }

    public String toString() { return resolver.toString(); }
}
```

## jp.lexer.ExtendedLexicon

```java
package jp.lexer;

import jp.grammar.*;

import java.util.*;

/**
 * A Lexicon that contains all words of a baseLexicon, plus possible additions.
 * This enables a lexicon segmented into a core vocabulary, and an extended
 * vocabulary.
 */
public class ExtendedLexicon extends Lexicon
{
    public final Lexicon baseLexicon;

    public ExtendedLexicon(Lexicon baseLexicon)
    {
        this.baseLexicon = baseLexicon;
    }

    public Set<Lexeme> getLexemes(String word)
    {
        Set<Lexeme> set = new HashSet<Lexeme>(baseLexicon.getLexemes(word));
        set.addAll(super.getLexemes(word));
        return set;
    }
}
```

## jp.lexer.Feature

```java
package jp.lexer;

import java.util.*;

```

```
/**
 *  Grammatical features, along with inflection and stemming rules.
 *  Each class is assigned a unique bit-index, to allow packing a set of
 *  features into a bit field.
 */

public enum Feature
{
    // Person:
    firstPerson,
    secondPerson,
    thirdPerson,

    // Number
    singular,
    plural (new Inflx("s/sh/x/z", "-es"),
            new Inflx("y", "ies"),
            new Inflx("", "s")),

    // Tense:
    nonpast,
    past,

    // big, bigger, biggest
    positive,
    comparative(new Inflx("&d", "-der"), new Inflx("&g", "-ger"),
        new Inflx("&t", "-ter"), new Inflx("e", "-r"),
        new Inflx("y", "-ier"), new Inflx("", "-er")),
    superlative(new Inflx("&d", "-dest"), new Inflx("&g", "-gest"),
        new Inflx("&t", "-test"), new Inflx("e", "-st"),
        new Inflx("y", "-iest"), new Inflx("", "est")),

    // Etc:
    possessive(new Inflx("s/x/z", "-'"), new Inflx("", "'s"))
    ;

    private static class Inflx
    {
        public final String[] oldEndings;
        public final String newEnding;
        public final boolean append;

        /**
         *  oldEndings: a slash ('/') separated list of old endings to match.
         *  An & matches any vowel (a, e, i, o, u).
         */
        public Inflx(String oldEndings, String newEnding)
        {
            this.oldEndings = oldEndings.split("/");

            append = newEnding.startsWith("-");

            this.newEnding = append ? newEnding.substring(1) : newEnding;
        }

        public static boolean endingMatches(String word, String ending)
        {
            int wordIdx = word.length() - ending.length();
            if (wordIdx < 0) return false;
            for (int i = 0; i < ending.length(); i++, wordIdx++)
            {
                char w = word.charAt(wordIdx);
                char e = ending.charAt(i);
                if (e == '&')
                {
                    if (!(w == 'a' || w == 'e' || w == 'i' || w == 'o'
                        || w == 'u')) return false;
                }
                else if (e != w) return false;
            }
            return true;
        }

        public String apply(String word)
        {
            for (String ending: oldEndings)
            {
                if (!endingMatches(word, ending)) continue;
                if (append) return word + newEnding;
                return word.substring(0, word.length() - ending.length())
                     + newEnding;
            }
            return null;
        }

        public boolean reverse(String word, List<String> results)
        {
            if (!word.endsWith(newEnding)) return false;

            if (append)
            {
                String oldWord =
                    word.substring(0, word.length() - newEnding.length());

                for (String oldEnding: oldEndings)
                {
                    if (!endingMatches(oldWord, oldEnding)) continue;

                    results.add(oldWord);
                    return true;
                }
                return false;
            }

            boolean anyWordsFound = false;
            for (String oldEnding: oldEndings)
            {
                String oldWord =
                    word.substring(0, word.length() - newEnding.length())
                     + oldEnding;

                if (!endingMatches(oldWord, oldEnding)) continue;

                results.add(oldWord);
                anyWordsFound = true;
            }
            return anyWordsFound;
        }
    }

    public final int bit;
    private final Inflx[] regularInflections;

    private Feature()
    {
        bit = 1 << ordinal();
        regularInflections = new Inflx[0];
    }

    private Feature(Inflx... regularInflections)
    {
        bit = 1 << ordinal();
        this.regularInflections = regularInflections;
    }

    public static String toString(int features)
    {
        StringBuilder sb = new StringBuilder();

        for (Feature f: Feature.values())
```

```java
        {
            if ((f.bit & features) == 0) continue;

            if (sb.length() != 0) sb.append("/");
            sb.append(f);
        }

        return sb.toString();
    }

    public String inflect(String word)
    {
        for (Inflx inflx: regularInflections)
        {
            String result = inflx.apply(word);
            if (result != null) return result;
        }
        return word;
    }

    public List<String> stem(String word)
    {
        List<String> results = new ArrayList<String>();

        for (Inflx inflx: regularInflections)
        {
            inflx.reverse(word, results);
        }

        // Remove bogus results by checking with inflect().
        Iterator<String> it = results.iterator();
        while (it.hasNext())
        {
            if (!inflect(it.next()).equals(word)) it.remove();
        }
        return results;
    }
}
```

## jp.lexer.Lexeme

```java
package jp.lexer;

import java.util.*;

public class Lexeme
{
    /** A helper class coupling a lexeme with a set of grammatical features */
    public static class Inflected
    {
        public Lexeme lexeme;
        public int features;

        public Inflected(Lexeme lexeme, int features)
        {
            this.lexeme = lexeme;
            this.features = features;
        }

        public String toString()
        {
            return lexeme.toString() + "(" + Feature.toString(features) + ")";
        }
    }

    public final WordClass wordClass;

    public final String lemma; // citation form

    public final Map<Integer, String> inflections
        = new HashMap<Integer, String>();

    public Lexeme(String lemma, WordClass wordClass)
    {
        this.wordClass = wordClass;
        this.lemma     = lemma;
    }

    public String toString()
    {
        return lemma + "[" + wordClass.abbreviation + "]";
    }

    public String inflect(int features)
    {
        // Check for hardcoded inflection for this specific feature set.
        String str = inflections.get(features);
        if (str != null) return str;

        str = lemma;

        // Check for closest hardcoded inflection.
        // Closeness is measured as number of bits in common.
        // The match must not include features not specified in our argument.
        int closestCount = 0;
        int closestFeatures = 0;
        for (Map.Entry<Integer, String> entry: inflections.entrySet())
        {
            int refFeatures = entry.getKey();
            if ((refFeatures & ~features) != 0) continue;
            int count = Integer.bitCount(refFeatures & features);

            if (count <= closestCount) continue;

            // A better match!
            closestCount = count;
            str = entry.getValue();
            closestFeatures = refFeatures;
        }
        // Remove features already encoded
        features &= ~closestFeatures;

        for (Feature f: Feature.values())
        {
            if ((f.bit & features) == 0) continue;

            str = f.inflect(str);
        }
        return str;
    }

    public Lexeme conjugate(int features, String value)
    {
        if (!inflections.containsKey(features) &&
            inflect(features).equals(value)) return this;

        inflections.put(features, value);
        return this;
    }

    public Lexeme conjugate(Feature person, Feature number, Feature tense,
        String value)
    {
        return conjugate(person.bit | number.bit | tense.bit, value);
    }

    public Lexeme conjugate(Feature person, Feature tense,
        String singular, String plural)
    {
        conjugate(person, Feature.singular, tense, singular);
        return conjugate(person, Feature.plural, tense, plural);
    }
}
```

## jp.lexer.LexemeMatch

```
package jp.lexer;

import jp.grammar.*;

import java.util.*;

/**
 * A tuple of orthographic word, lexeme, and features.
 */
public class LexemeMatch
{
    public final String word;
    public final Lexeme lexeme;
    public final int features;

    public LexemeMatch(String word, Lexeme lexeme, int features)
    {
        this.word = word;
        this.lexeme = lexeme;
        this.features = features;
    }

    public String toString()
    {
        return lexeme.toString() + "[" + Feature.toString(features) + "]";
    }
}
```

## jp.lexer.LexerException

```
package jp.lexer;

public class LexerException extends Exception
{
    public LexerException(String msg)
    {
        super(msg);
    }
}
```

## jp.lexer.Lexer

```
package jp.lexer;

import jp.grammar.*;

import java.util.*;

/**
 * A basic tokenizer, which splits a string into orthographic words,
 * punctuation and quoted strings.
 */

public class Lexer
{
    final private static int mNone = 0;
    final private static int mWord = 1;
    final private static int mString = 2;

    static public class WordToken extends Terminal
    {
        public final String word;

        public WordToken(String word) { this.word = word; }

        public String toString() { return word; }
    }

    static public class StringToken extends Terminal
    {
        public final String string;

        public StringToken(String string) { this.string = string; }

        public String toString() { return '"' + string + '"'; }
    }

    static public class PunctToken extends Terminal
    {
        public final char punct;

        public PunctToken(char punct) { this.punct = punct; }

        public String toString() { return Character.toString(punct); }
    }

    protected Object wordToken(String token) { return new WordToken(token); }
    protected Object punctToken(char token)  { return new PunctToken(token); }

    public List<Object> tokenize(String string) throws LexerException
    {
        List<Object> result = new ArrayList<Object>();

        int tokenIndex = 0;

        int mode = mNone;

        for (int i = 0; i < string.length(); i++)
        {
            char c = string.charAt(i);

            switch (mode)
            {
            case mWord:
                if (Character.isLetter(c)) continue;

                result.add(wordToken(string.substring(tokenIndex, i)));
                mode = mNone;

                // fall through //

            case mNone:
                switch (c)
                {
                case ' ': continue;
                case '"':
                    mode = mString;
                    tokenIndex = i;
                    break;
                case '.':
                case ',':
                case ':':
                case '(':
                case ')':
                case ';':
                    result.add(punctToken(c));
                    break;

                default:
                    if (Character.isLetter(c))
                    {
                        tokenIndex = i;
                        mode = mWord;
                    }
                    else throw new LexerException("Invalid character at #" + i);
                }
                break;

            case mString:
                if (c == '"')
                {
                    mode = mNone;
                    result.add(new StringToken(
                        string.substring(tokenIndex + 1, i)));
                }
                break;
            }
        }

        switch (mode)
        {
        case mWord:
            result.add(wordToken(string.substring(tokenIndex)));
            break;
        case mNone: /* Do nothing */ break;
        case mString: throw new LexerException("Unterminated string");
        }
        return result;
    }
}
```

## jp.lexer.Lexicon

```
package jp.lexer;

import java.util.*;

import jp.util.*;

public class Lexicon implements Iterable<Lexeme>
{
    private MultiMap<String, Lexeme> lexemes = new MultiMap<String, Lexeme>();
    private MultiMap<String, Lexeme> irregular = new MultiMap<String, Lexeme>();

    public Set<Lexeme> getLexemes(String word)
    {
        Set<Lexeme> set = new HashSet<Lexeme>(lexemes.get(word));
        return set;
    }

    public Set<Lexeme> getLexemes(String word, WordClass wc)
    {
        Set<Lexeme> set = getLexemes(word);

        Iterator<Lexeme> it = set.iterator();
        while (it.hasNext())
        {
            if (it.next().wordClass != wc) it.remove();
        }
        return set;
    }

    public Set<Lexeme> addLexeme(String lemma, WordClass wordClass)
    {
        Set<Lexeme> lexes = getLexemes(lemma, wordClass);
        if (lexes.size() == 0)
        {
            Lexeme lexeme = new Lexeme(lemma, wordClass);
            lexes.add(lexeme);
            lexemes.put(lemma, lexeme);
        }
        return lexes;
    }

    public Lexeme addLexeme(Lexeme lexeme)
    {
        lexemes.put(lexeme.lemma, lexeme);
        for (String s: lexeme.inflections.values())
            irregular.put(s, lexeme);

        return lexeme;
    }

    public WordTerminal wordTerm(String word)
    {
        addLexeme(word, WordClass.particle);
        return new WordTerminal(word);
    }

    public WordTerminal wordTerm(String word, WordClass... classes)
    {
        for (WordClass wc: classes) addLexeme(word, wc);
        return new WordTerminal(word);
    }

    public Iterator<Lexeme> iterator() { return lexemes.values().iterator(); }

    private void internalStem(String goal, String word, int features,
        List<Lexeme.Inflected> results)
    {
        for (Feature f: Feature.values())
        {
            for (String stem: f.stem(word))
            {
                assert(stem.length() < word.length());

                // For every result, recurse with the new feature set.
                internalStem(goal, stem, features | f.bit, results);
            }
        }

        for (Lexeme lexeme: lexemes.get(word))
        {
            if (lexeme.inflect(features).equals(goal))
                results.add(new Lexeme.Inflected(lexeme, features));
        }
    }

    public List<Lexeme.Inflected> stemAndLookUp(String word)
    {
        List<Lexeme.Inflected> results = new ArrayList<Lexeme.Inflected>();

        for (Lexeme lexeme: irregular.get(word))
        {
            for (Map.Entry<Integer, String> e: lexeme.inflections.entrySet())
                if (e.getValue().equals(word))
                    results.add(new Lexeme.Inflected(lexeme, e.getKey()));
        }

        internalStem(word, word, 0, results);
        return results;
    }
}
```

### jp.lexer.WordClass

```
package jp.lexer;

/**
 *  The syntactic classes of words.
 *  Each class is assigned a unique bit-index, to allow packing a set of
 *  word-classes into a bit field.
 */

public enum WordClass
{
    particle ("·"),
    countNoun ("CN"),
    massNoun ("MN"),
    properNoun ("PN"),
    adjective ("Adj"),
    verb ("V");

    public final int bit;
    public final String abbreviation;

    WordClass(String abbreviation)
    {
        bit = 1 << ordinal();
        this.abbreviation = abbreviation;
    }

    public static String toString(int classes)
    {
        StringBuilder sb = new StringBuilder();

        for (WordClass wc: WordClass.values())
        {
            if ((wc.bit & classes) == 0) continue;

            if (sb.length() != 0) sb.append("/");
            sb.append(wc);
        }

        return sb.toString();
    }
}
```

## jp.lexer.WordTerminal

```
package jp.lexer;

import jp.grammar.*;

/** A CFG terminal matching a specific word (String). */
public class WordTerminal extends Terminal
{
    public final String word;

    public WordTerminal(String word) { this.word = word; }

    public String toString() { return word; }

    public ParseNode.TerminalNode match(Object t)
    {
        if (t instanceof String && t.equals(word))
            return new ParseNode.TerminalNode(word);
        return null;
    }
}
```

## B.4 jp package

The jp package contains the grammar compiler (GC) described in section 6.1, and various test-classes, some of which implement examples found in this paper.

### jp.GC

```java
/*
 *   GC: Grammar Compiler
 *
 *   Usage: java GC [files]
 *
 *   Compiles input files containing BNF productions, then prints Java code
 *   suitable for use with the jp.grammar package to standard output.
 */

package jp;

import java.util.*;
import java.io.*;

public class GC
{
    public static class CompilerException extends Exception
    {
        public CompilerException(File filename, int lineNo, String message)
        {
            super(filename + ":" + lineNo + ": " + message);
        }
    }

    public static class Grammar
    {
        private static class Production
        {
            private List<String> tokens = new ArrayList<String>();

            public void addToken(String token) { tokens.add(token); }

            public void print()
            {
                for (int i = 0; i < tokens.size(); i++)
                {
                    System.out.print(", ");
                    System.out.print(tokens.get(i));
                }
            }
        }

        // Maps nonTerminals to productions
        private Map<String, List<Production>> productions
            = new HashMap<String, List<Production>>();

        private String currentNonTerminal;
        private Production currentExpansion;

        public void addExpansionToken(String token)
        {
            currentExpansion.addToken(token);
        }

        public void beginProduction(String nonTerminal)
        {
            finishProduction();
            currentNonTerminal = nonTerminal;
            currentExpansion = new Production();

            if (!productions.containsKey(nonTerminal))
                productions.put(nonTerminal, new ArrayList<Production>());
        }

        public void finishExpansion()
        {
            if (currentExpansion != null)
                productions.get(currentNonTerminal).add(currentExpansion);
            currentExpansion = new Production();
        }

        public void finishProduction()
        {
            if (currentNonTerminal == null) return;

            finishExpansion();
            currentNonTerminal = null;
        }

        public void print()
        {
            for (Map.Entry<String, List<Production>> entry:
                productions.entrySet())
            {
                System.out.print("    NonTerminal " + entry.getKey()
                    + " = new NonTerminal(\"" + entry.getKey() + "\");\n");
            }

            for (Map.Entry<String, List<Production>> entry:
                productions.entrySet())
            {
                for (Production p: entry.getValue())
                {
                    System.out.print(
                        "    new Production(" + entry.getKey());
                    p.print();
                    System.out.print(");\n");
                }
            }
        }
    }

    public static boolean isValidIdentifier(String s)
    {
        if (s.length() == 0 || !Character.isJavaIdentifierStart(s.charAt(0)))
            return false;

        for (int i = 1; i < s.length(); i++)
        {
            if (!Character.isJavaIdentifierPart(s.charAt(i))) return false;
        }
        return true;
    }

    public static void compile(File filename)
        throws IOException, CompilerException
    {
        BufferedReader reader = new BufferedReader(new FileReader(filename));

        Grammar grammar = new Grammar();

        int lineNo = 0;
        String line;
        while ((line = reader.readLine()) != null)
        {
            lineNo++;

            {
                int i = line.indexOf('#');
                if (i != -1) line = line.substring(0, i);
            }

            String[] tokens = line.split("\\s+");
            int tokenIndex = 0;

            // Second check due to Sun JRE bug: split should return an empty
            // list for an empty input, but doesn't.
            if (tokens.length == 0 ||
                (tokens.length == 1 && tokens[0].length() == 0)) continue;

            if (tokens[0].length() > 0)
            {
                String nonTerminal = tokens[0];
                if (!isValidIdentifier(nonTerminal))
                    throw new CompilerException(filename, lineNo,
                        "Expected start of production, but \""
                        + nonTerminal + "\" is not a valid identifier");

                grammar.beginProduction(nonTerminal);

                if (tokens.length < 2 || !tokens[1].equals("="))
                {
                    throw new CompilerException(filename, lineNo,
                        '"' + nonTerminal + "\" not followed by =");
                }
                tokenIndex = 2;
            }
            else if (grammar.currentNonTerminal == null)
            {
                throw new CompilerException(filename, lineNo,
                    "First line must not start with a space");
            }

            for (int i = tokenIndex; i < tokens.length; i++)
            {
                if (tokens[i].length() == 0) continue;

                if (tokens[i].equals("|")) grammar.finishExpansion();
                else grammar.addExpansionToken(tokens[i]);
            }
        }

        grammar.finishProduction();

        grammar.print();
    }

    public static void main(String[] args)
        throws IOException
    {
        try
        {
            for (int i = 0; i < args.length; i++)
            {
                compile(new File(args[i]));
            }
        }
        catch (CompilerException e)
        {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

### jp.EpsTest

```
package jp;
```

```

import jp.grammar.*;

import java.util.*;

/** An implementation of the epsilon problem */
public class EpsTest
{
    public static void main(String[] args) throws Exception
    {
        Terminal plus = new Terminal.NamedTerminal("+");
        NonTerminal S = new NonTerminal("S");
        NonTerminal P = new NonTerminal("P");
        NonTerminal Q = new NonTerminal("Q");
        NonTerminal E = new NonTerminal("E");

        new Production(S, E);
        new Production(S, P);
        new Production(P, Q, plus);
        new Production(Q, E);
        new Production(E);

        Set<ParseNode.ProductionNode> result = Earley.debugParse(S, plus);

        System.out.println("\nParse trees:");
        for (ParseNode.ProductionNode l: result) l.print(0);
    }
}
```

### jp.ExprTest

```
package jp;

import jp.grammar.*;

import java.util.*;

/** An implementation of the Parsing example */
public class ExprTest
{
    public static void main(String[] args) throws Exception
    {
        Terminal a = new Terminal.NamedTerminal("a");
        Terminal plus = new Terminal.NamedTerminal("+");
        NonTerminal E = new NonTerminal("E");

        new Production(E, a);
        new Production(E, E, plus, E);

        Set<ParseNode.ProductionNode> result = Earley.debugParse(E, a, plus, a);

        System.out.println("\nParse trees:");
        for (ParseNode.ProductionNode l: result) l.print(0);
    }
}
```

### jp.StemTest

```
package jp;

import jp.grammar.*;
import jp.lexer.*;

import static jp.lexer.Lexicon.*;
import static jp.lexer.WordClass.*;
import static jp.lexer.Feature.*;

import java.util.*;

/** Runs a number of inflection and stemmer tests */
public class StemTest
{
    public static Lexicon lexicon = new Lexicon();

    public static void inflectAndStem(Lexeme lex, int features)
    {
        String word = lex.inflect(features);
        System.out.println(word + " = " + lexicon.stemAndLookUp(word));
    }

    public static void main(String[] args) throws Exception
    {
        Lexeme weatherCN = new Lexeme("weather", countNoun);
        Lexeme cloudCN   = new Lexeme("cloud", countNoun);
        Lexeme rainV     = new Lexeme("rain", verb);
        Lexeme bigAdj    = new Lexeme("big", adjective);

        Lexeme beV = new Lexeme("be", verb)
            .conjugate(firstPerson,  nonpast, "am",   "are")
            .conjugate(secondPerson, nonpast, "are",  "are")
            .conjugate(thirdPerson,  nonpast, "is",   "are")
            .conjugate(firstPerson,  past,    "was",  "were")
            .conjugate(secondPerson, past,    "were", "were")
            .conjugate(thirdPerson,  past,    "was",  "were");

        lexicon.addLexeme(beV);
        lexicon.addLexeme(weatherCN);
        lexicon.addLexeme(rainV);
        lexicon.addLexeme(bigAdj);

        inflectAndStem(weatherCN, singular.bit);
        inflectAndStem(weatherCN, plural.bit);
        inflectAndStem(beV, firstPerson.bit | nonpast.bit | singular.bit);
        inflectAndStem(rainV, nonpast.bit | firstPerson.bit | singular.bit);
        inflectAndStem(bigAdj, superlative.bit);

        for (String s: new String[] { "big", "large", "fair", "hot", "long" })
        {
            System.out.println(
                positive.inflect(s) + ", " +
                comparative.inflect(s) + ", " +
                superlative.inflect(s));
        }

        System.out.println("class   -> " + plural.stem("class"));
        System.out.println("ties    -> " + plural.stem("ties"));
        System.out.println("hottest -> " + superlative.stem("hottest"));
    }
}
```

## jp.Test

```
package jp;

import jp.grammar.*;
import jp.lexer.*;

import static jp.lexer.Lexicon.*;
import static jp.lexer.WordClass.*;
import static jp.lexer.Feature.*;

import java.util.*;

public class Test
{
    /** A Lexer that return the same punctuation token for a given character. */
    private static class MyLexer extends Lexer
    {
        private Map<Character, PunctToken> punctTokens
            = new HashMap<Character, PunctToken>();

        protected String wordToken(String token)
        {
            return token;
        }

        protected PunctToken punctToken(char token)
        {
            PunctToken t = punctTokens.get(token);
            if (t != null) return t;

            t = new PunctToken(token);
            punctTokens.put(token, t);
            return t;
        }
    }

    /** A ClassResolver for proper nouns. Accepts new PNs if capitalized. */
    private static class ProperNounCR implements ClassResolver
    {
        public final Lexicon lexicon;

        public ProperNounCR(Lexicon lexicon)
        {
            this.lexicon = lexicon;
        }

        public Object resolve(String word)
        {
            Set<Lexeme> lexemes = lexicon.getLexemes(word, properNoun);
            if (lexemes.size() > 0) return lexemes;
            return Character.isUpperCase(word.charAt(0)) ?
                   lexicon.addLexeme(word, properNoun) : null;
        }

        public String toString()
        {
            return "<properNoun*>";
        }
    }

    public static void main(String[] args) throws Exception
    {
        MyLexer lexer = new MyLexer();

        Terminal period = lexer.punctToken('.');

        Lexicon baseLex = new Lexicon();

        baseLex.addLexeme(new Lexeme("be", verb)
            .conjugate(firstPerson,  nonpast, "am",  "are")
            .conjugate(secondPerson, nonpast, "are", "are")
            .conjugate(thirdPerson,  nonpast, "is",  "are")
            .conjugate(firstPerson,  past,    "was",  "were")
            .conjugate(secondPerson, past,    "were", "were")
            .conjugate(thirdPerson,  past,    "was",  "were"));

        baseLex.addLexeme(new Lexeme("star", countNoun));

        WordTerminal shines = baseLex.wordTerm("shines", verb);
        WordTerminal is     = new WordTerminal("is");
        WordTerminal are    = new WordTerminal("are");

        WordTerminal the   = baseLex.wordTerm("the");
        WordTerminal it    = baseLex.wordTerm("it");
        WordTerminal in    = baseLex.wordTerm("in");
        WordTerminal at     = baseLex.wordTerm("at");
        WordTerminal on     = baseLex.wordTerm("on");
        WordTerminal under = baseLex.wordTerm("under");
        WordTerminal above = baseLex.wordTerm("above");
        WordTerminal beside = baseLex.wordTerm("beside");
        WordTerminal behind = baseLex.wordTerm("behind");

        ClassTerminal _countNoun_ = new ClassTerminal();
        ClassTerminal _massNoun_ = new ClassTerminal();
        ClassTerminal _adjective_ = new ClassTerminal();
        ClassTerminal _properNoun_ = new ClassTerminal();

        ClassResolver[] baseResolvers = new ClassResolver[] {
            new ClassResolver.StaticCR(baseLex, countNoun),
            new ClassResolver.StaticCR(baseLex, massNoun),
            new ClassResolver.StaticCR(baseLex, adjective),
            new ClassResolver.StaticCR(baseLex, properNoun)
        };

        ClassTerminal[] classTerminals = new ClassTerminal[]
            { _countNoun_, _massNoun_, _adjective_, _properNoun_ };

// GRAMMAR BEGIN //
    NonTerminal transitiveVerb = new NonTerminal("transitiveVerb");
    NonTerminal sentence = new NonTerminal("sentence");
    NonTerminal staticSpatialAdposition = new NonTerminal("
         staticSpatialAdposition");
    NonTerminal input = new NonTerminal("input");
    NonTerminal simpleSentence = new NonTerminal("simpleSentence");
    NonTerminal adjectives = new NonTerminal("adjectives");
    NonTerminal nounPhrase = new NonTerminal("nounPhrase");
    NonTerminal adpositionalPhrase = new NonTerminal("adpositionalPhrase");
    NonTerminal intransitiveVerb = new NonTerminal("intransitiveVerb");
    NonTerminal predicate = new NonTerminal("predicate");
    NonTerminal determiner = new NonTerminal("determiner");
    NonTerminal sentences = new NonTerminal("sentences");
    new Production(transitiveVerb, is);
    new Production(transitiveVerb, are);
    new Production(sentence, simpleSentence);
    new Production(staticSpatialAdposition, in);
    new Production(staticSpatialAdposition, on);
    new Production(staticSpatialAdposition, at);
    new Production(staticSpatialAdposition, under);
    new Production(staticSpatialAdposition, above);
    new Production(staticSpatialAdposition, behind);
    new Production(staticSpatialAdposition, beside);
    new Production(input, sentence);
    new Production(input, sentences);
    new Production(simpleSentence, nounPhrase, predicate);
    new Production(adjectives, _adjective_, adjectives);
    new Production(adjectives);
    new Production(nounPhrase, determiner, _countNoun_);
    new Production(nounPhrase, _properNoun_);
    new Production(adpositionalPhrase, staticSpatialAdposition, nounPhrase);
    new Production(intransitiveVerb, shines);
    new Production(intransitiveVerb, is);
    new Production(intransitiveVerb, are);
    new Production(predicate, intransitiveVerb);
    new Production(predicate, intransitiveVerb, adpositionalPhrase);
    new Production(predicate, transitiveVerb, nounPhrase);
    new Production(predicate, transitiveVerb, _adjective_);
    new Production(determiner, the, adjectives);
    new Production(sentences, sentence, period, sentences);
    new Production(sentences, sentence, period);
// GRAMMAR END //

        // Parse every argument as a separate sentence.
        for (String s: args)
        {
            List<Object> tokens = lexer.tokenize(s);
            System.out.println(tokens);

            for (int i = 0; i < classTerminals.length; i++)
                classTerminals[i].resolver = baseResolvers[i];

            System.out.println("Old lexemes:");
            for (Lexeme lex: baseLex)
            {
                System.out.print(lex + "; ");
            }
            System.out.println();

            System.out.println("Parse:");
            Set<ParseNode.ProductionNode> result = Earley.parse(input,
                tokens.toArray(new Object[0]));

            if (result.size() == 0)
            {
                System.out.println("No results; trying new words.");

                Lexicon newLex = new ExtendedLexicon(baseLex);

                ClassResolver[] lenientResolvers = new ClassResolver[] {
                    new ClassResolver.LenientCR(newLex, countNoun),
                    new ClassResolver.LenientCR(newLex, massNoun),
                    new ClassResolver.LenientCR(newLex, adjective),
                    new ProperNounCR(newLex)
                };

                for (int i = 0; i < classTerminals.length; i++)
                    classTerminals[i].resolver = lenientResolvers[i];

                result = Earley.parse(input, tokens.toArray(new Object[0]));

                if (result.size() != 0)
                {
                    System.out.println("New lexemes:");
                    for (Lexeme lex: newLex)
                    {
                        System.out.print(lex + "; ");
                    }
                    System.out.println();
                }
            }

            for (ParseNode.ProductionNode l: result)
            {
                l.print(0);
            }
        }
    }
}
```

## B.5 Other files

### Makefile

```
Test:   Test.java
        javac -cp .. Test.java util/*.java grammar/*.java lexer/*.java semantics/*.java
#       java -cp .. jp.Test "the sky is blue"
#       java -ea -cp .. jp.Test "the stars are shiny"
        java -ea -cp .. jp.Test "the cat is wet"

StemTest:       StemTest.java
        javac -cp .. StemTest.java util/*.java grammar/*.java lexer/*.java
        java -ea  -cp .. jp.StemTest

ExprTest:       ExprTest.java
        javac -cp .. ExprTest.java util/*.java grammar/*.java
        java -ea  -cp .. jp.ExprTest

EpsTest:        EpsTest.java
        javac -cp .. EpsTest.java util/*.java grammar/*.java
        java -ea  -cp .. jp.EpsTest

SemTest:        SemTest.java
        javac -cp .. SemTest.java util/*.java lexer/*.java semantics/*.java
        java -ea  -cp .. jp.SemTest

rungc:  GC.class
        java -ea  -cp .. jp.GC TestGrammar.gr

gram:   TestGrammar.gr GC.class
        sed '/GRAMMAR BEGIN/,$$d' Test.java > Test.java.new
        echo '// GRAMMAR BEGIN //' >> Test.java.new
        java -ea  -cp .. jp.GC TestGrammar.gr >> Test.java.new
        sed -n '/GRAMMAR END/,$$p' Test.java >> Test.java.new
        mv Test.java Test.java.bak
        mv Test.java.new Test.java

GC.class: GC.java
        javac -cp .. GC.java

clean:
        rm -f *.class **/*.class *.bak
```

### TestGrammar.gr

The grammar for `jp.Test`, for use with the grammar compiler.

```
#################
# Words         #
#################

adjectives      = _adjective_ adjectives |

determiner      = the adjectives

transitiveVerb = is | are

intransitiveVerb = shines | is | are

staticSpatialAdposition = in | on | at | under | above | behind | beside

#################
# Phrases       #
#################

```

```
nounPhrase       = determiner _countNoun_
                 | _massNoun_
                 | _properNoun_

adpositionalPhrase = staticSpatialAdposition nounPhrase

predicate        = intransitiveVerb
                 | intransitiveVerb adpositionalPhrase
                 | transitiveVerb nounPhrase
                 | transitiveVerb _adjective_

#################
# Sentence      #
#################

simpleSentence  = nounPhrase predicate

sentence         = simpleSentence

sentences        = sentence period sentences
                 | sentence period

input            = sentence
                 | sentences
```