Computer Science Department Technical Report
Artificial Intelligence Laboratory
University of California
Los Angeles, CA 90024-1596

THE RHAPSODY PHRASAL PARSER AND GENERATOR

John F. Reeves

December 1989
CSD-890064

The Rhapsody Phrasal Parser and Generator

John F. Reeves*
Artificial Intelligence Laboratory
Computer Science Department
University of California, Los Angeles, CA 90024

November 11, 1989

Abstract

PPARSE (Phrasal PARSEr) and PGEN (Phrasal GENerator) are programs for natural language parsing and generation used with the Rhapsody knowledge representation package[Turner and Reeves, 1987]. Both programs used a database of knowledge about language encoded as *phrases.* Each phrase associates a conceptual object with a linguistic structure, such as a word, idiom, or syntactic form. The mechanics of phrasal parsing and generation with PPARSE/PGEN, and issues in phrasal processing and lexical construction are discussed. A technical description of PPARSE/PGEN is given, and an annotated sample lexicon and program runs are presented.

1 Introduction

PPARSE (Phrasal PARSEr) and PGEN (Phrasal GENerator) are two programs for natural language parsing and generation used with the Rhapsody knowledge representation package[Turner and Reeves, 1987]. Both programs used a database of knowledge about language encoded as *phrases.* Each phrase associates a conceptual object with a linguistic structure, such as a word, idiom, or syntactic form. PPARSE takes as input a natural language sentence as a list of atoms, and uses the phrases to produce a conceptual representation of the sentence. PGEN does the reverse process; the input is a conceptual object, and a list of natural language words is produced.

Phrasal parsing integrates syntactic and semantic information during the parsing process. Phrasal parsing evolved from case grammar[Fillmore, 1968] where syntactic rules were used to fill in semantic case frames. Functional grammar[Kay, 1979] extended this approach by using a single kind of formal structure to specify patterns of features, function assignments, lexical items, and constituent orderings. Different variations of functional grammar have been proposed, depending on the focus of research and type of formal structure used. For example, lexical-functional grammar[Bresnan and Kaplan, 1982] emphasizes the role of the lexicon within a transformational grammar approach, and definite-clause grammars[Pereira

*This work was supported by a grant from the Hughes Artificial Intelligence Center.

and Warren, 1980] focus on how to specify lexical and syntactic information in logic. (See also Winograd [1983, pp. 311-351] for an overview of function grammars.)

PPARSE and PGEN are intended as tools for language analysis and use by programs using Rhapsody. The aims of PPARSE/PGEN are (1) to provide a simple, extensible mechanism for producing conceptual representations from text (and vice versa), and (2) to provide a readable, declarative representation for the linguistic knowledge used by the parser and generator. The restrictions placed on the PPARSE/PGEN user are those of symbolic, frame-based knowledge representation and computing. PPARSE, PGEN, and Rhapsody were designed to allow the user to specify the knowledge used by their programs, without making a priori representational decisions about the content of the knowledge that the program uses. The generality of phrases, the form of the conceptualizations underlying the language, and the representation of linguistic knowledge are in the hands of the PPARSE/PGEN user. However, computational models of natural language parsing generation are research topics and some design decisions in PPARSE/PGEN have been made in the interests of modularity and ease of use, instead of for cognitive validity or optimal performance. Users of PPARSE/PGEN are encourage to copy and modify the package for their own purposes, and to offer improvements and alternatives.

PPARSE/PGEN were developed for use in the THUNDER (THematic UNDerstanding from Ethical Reasoning) project[Reeves, 1988; Reeves, 1989; Reeves, Forthcoming]. THUNDER is a story understanding system that reads sentences and short narratives, and answers questions about the ethical judgments, irony, and thematic aspects of the input. A sample of THUNDER's I/O behavior is shown in figure 1. THUNDER uses a hybrid architecture: PPARSE/PGEN are used to apply knowledge about linguistic structure in parsing and generation, while semantic interpretation and reasoning are done using demon-based processing. Demons are fired from the phrases to perform disambiguation, semantic attachment, and explanation tasks. The general strategy taken in THUNDER is that the parser should handle the application of all language-specific knowledge, and that decisions requiring semantic reasoning, reference to context, or other extra-linguistic knowledge are external to the parser. In this paper, the knowledge representation and example phrases are taken from THUNDER, but PPARSE/PGEN are not dependent on the particular representation, task domain, or overall system architecture. PPARSE/PGEN are also currently being used for natural language I/O in the STARE legal reasoning system[Goldman et al., 1987], and the ARIEL analogical reasoning system[August and Dyer, 1985a; August and Dyer, 1985b].

This document is organized in four sections. The first gives an overview of how phrasal parsing and generation are implemented in PPARSE/PGEN. Section two discusses problem areas in phrasal parsing, generation, and lexical construction, and how those issues are addressed in PPARSE/PGEN. The third section is a technical description of PPARSE/PGEN, and is intended as a user's manual. Section four provides a sample lexicon and an annotated trace.

2 PPARSE and PGEN Overview

The implementation of phrasal parsing and generation used in PPARSE/PGEN is based on the PHRAN phrasal parser[Wilensky and Arens, 1980; Wilensky, 1981; Jacobs, 1985b;

```
Input: TO GET THE MONEY TO BUY A NEW CAR *COMMA* JOHN ROBBED
A BANK *PERIOD*

Why is the plan wrong?

IT IS WRONG BECAUSE HE TOOK MONEY FROM THE BANK DEPOSITORS

Additional reasons why the plan is wrong:

BECAUSE HE THREATENED THE BANK TELLER *POSSESSIVE* HEALTH

BECAUSE HE GOT THE CAR BUT HE TOOK MONEY FROM THE BANK
DEPOSITORS AND SAVING MONEY IS-MORE-IMPORTANT-THAN GETTING
A CAR

BECAUSE HE GOT THE CAR BUT HE THREATENED THE BANK TELLER
POSSESSIVE* HEALTH AND KEEPING HEALTH IS-MORE-IMPORTANT-THAN
GETTING A CAR

BECAUSE HE MIGHT GET ARRESTED

Reasons why the plan is right:

BECAUSE HE GOT THE CAR

Inferences from evaluation:

HE BELIEVES-THAT GETTING A CAR IS-MORE-IMPORTANT-THAN THE BANK
DEPOSITORS SAVING MONEY

HE BELIEVES-THAT GETTING A CAR IS-MORE-IMPORTANT-THAN THE BANK
TELLER KEEPING HEALTH
```

Figure 1: Sample THUNDER I/O — PPARSE/PGEN are used to process the text in capital letters.

Arens, 1986]. Lexical entries are associated with functional groups of text—words, idioms, and syntactic patterns—and are composed of a pattern and a concept (termed a *PC pair*). For example, the lexical entry for the word "robbed" could be defined as:

```
(phrase:define 'ph-robbed
     (comment "robbed")
     (pattern 'robbed)
     (concept (action 'type   'atrans
                      'actor  ?robber
                      'object ?money
                      'from   ?victim
                      'to     ?robber
                      'time   'past
                      'in-pschema PS-Robbery)))
```

The pattern is the surface word "robbed" and the concept is the CD action **atrans** in the plan schema for generic robbery PS-Robbery.[1] The **?variable-name** notation is used to denote variables in the conceptual objects. Patterns can be composed of sequences of conceptual objects to match syntactic patterns. For example, the entry for the pattern human-robs-bank using the subject-verb-object syntax form looks like:

```
(phrase:define 'ph-human-rob-bank
     (comment "<human> <action:rob> <financial-institution>")
     (pattern ?*robber+&human
              (action 'type   'atrans
                      'actor  ?robber
                      'object ?money
                      'from   ?victim
                      'to     ?robber
                      'time   ?tense
                      'in-pschema PS-Robbery
              ?*bank+&financial-institution))
     (concept (action 'type   'atrans
                      'actor  ?robber
                      'object ?money
                      'from   (human 'employee-of ?bank)
                      'to     ?robber
                      'time   ?tense
                      'in-pschema PS-Bank-Robbery)))
```

The pattern is the three ordered constituents **?robber**, the **atrans** action, and **?bank**. The **?*robber+&human** notation is used for phrasal variables. The phrasal variable **?*robber+&human**

[1]The representation system for conceptual objects used in the examples is based on Conceptual Dependency (CD)[Schank, 1973; Schank, 1975]. **Atrans** is an action primitive standing for abstract transfer of possession. Plan Schemas (PS) are used to represent intentional information in THUNDER and are based on Memory Organization Packets (MOPS)[Schank, 1982].

will match a conceptual object of the class `&human` and will bind the variable `?robber` to that conceptual object. Variable matching is done using unification, so variables of the same name only match conceptual objects that are the same. When the human-rob-bank pattern is matched, the plan schema is specialized from generic robbery (PS-ROBBERY) to bank robbery (PS-BANK-ROBBERY).

PPARSE and PGEN both use the same set of phrases. The phrases are stored in discrimination nets (d-nets)[Charniak et al., 1980, pp. 162–176] to quickly access candidate phrases. For parsing, the phrases are indexed in a d-net by the pattern, and for generation by the concept. Since patterns and concepts contain variables, the d-net returns a candidate set of phrases which are then tested for variable constraints and user specified tests before being applied.

During parsing, words are read and matched against the patterns in phrases. If no pattern matches, the words are kept in an list in order to match patterns with more than one constituent. When a pattern matches, its constituents in the list are rewritten to the concept in the phrase. This process results in the bottom-up construction of a parse tree where each node is the concept of a matched pattern.

Figure 2 shows how words and constituents are successively re-written for the sentence "John robbed a bank." The word "John" matches the pattern for the conceptual representation of a human named John, and "robbed" is represented as an action in the robbery plan schema PS-Robbery. The word "a" doesn't have its own phrase, so the parser reads the next word "bank", and matches the pattern "a bank" which has the representation object for `financial-institution` as its concept. The human-rob-bank pattern then matches the top nodes of the tree, yielding the action in the bank robbery plan schema as the parsed representation of the sentence.

PGEN generates an output list of natural language words from a given conceptual object. PGEN constructs a generation tree beginning with the input concept at the root node. This object is matched against the concept section of phrases, and new nodes are added to tree for each of the pattern elements of the matching phrase. This process is repeated depth-first for each of the new nodes until words (lisp atoms) are reached at the leaf nodes of the tree. Since phrase patterns generally break apart the concept into constituent parts, this process is called *recursive descent* generation.

The generation process can be illustrated by reading Figure 2 from the bottom up: PGEN takes as input the **atrans** action with the robber and bank variables instantiated. This concept matches the human-rob-bank phrases, so new nodes are constructed for the robber, the rob action, and the bank, with the variable bindings passed to the new nodes. This process is repeated for the human, generating his first name "John", and the action is realized with the word "robbed". The bank concept re-writes to the article+institution pattern, which in turn re-writes to "a bank".

3 Issues in Phrasal Parsing and Lexicon Construction

There are three general problem areas involved in constructing a phrasal natural language parser/generator: (1) the domain of the parser, (2) phrase definition and lexical construction, (3) and how the parser handles structural ambiguity. In this section, each of these issues are

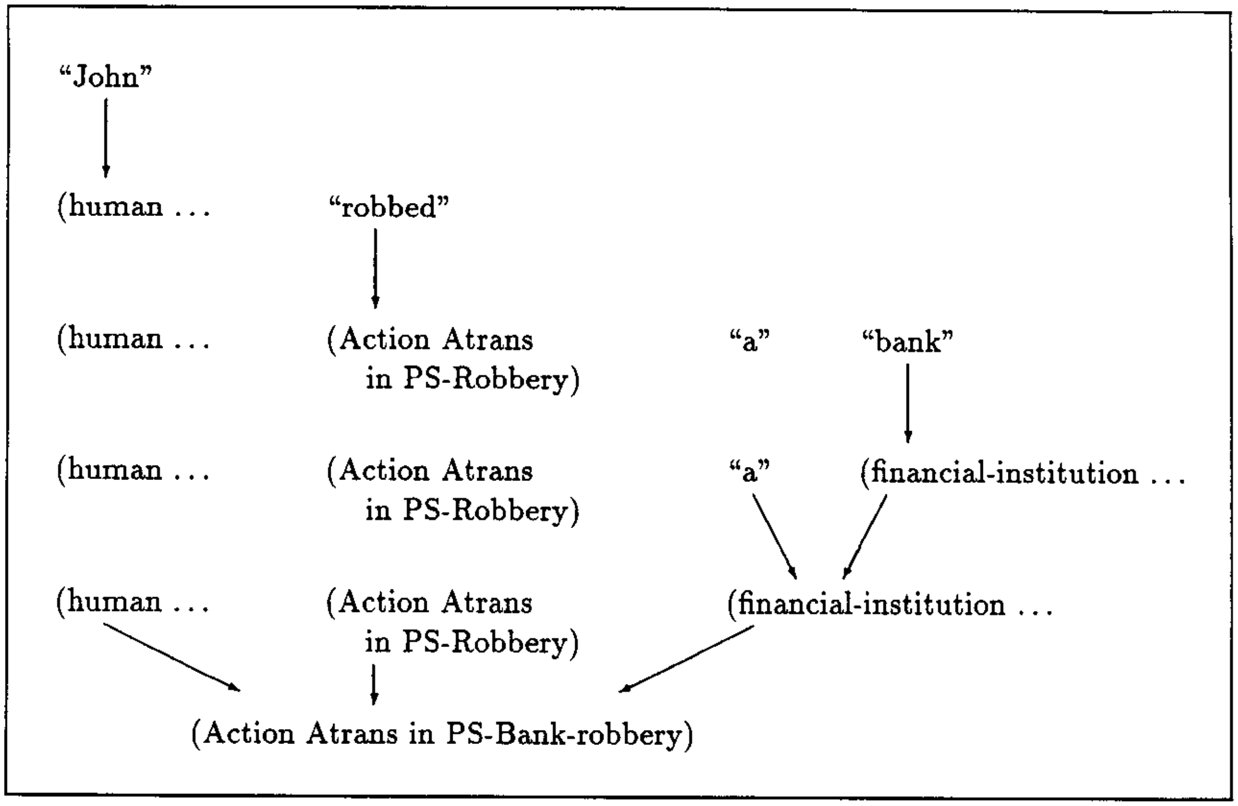
"John"
(human ...
"robbed"
(human ...
(Action Atrans in PS-Robbery)
"a"
"bank"
(human ...
(Action Atrans in PS-Robbery)
"a"
(financial-institution ...
(human ...
(Action Atrans in PS-Robbery)
(financial-institution ...
(Action Atrans in PS-Bank-robbery)

Figure 2: Example parse tree constructed by rewriting patterns

discussed, and how they are addressed in PPARSE/PGEN is presented.

3.1 The Domain of the Parser

What is the separation point between parsing, and inference and reasoning? For example, a program that reads the sentence:

> John wanted a new car, so he decided to rob a bank.

should be able to infer that John is going to get the money from the bank, and use it to buy a new car.[2] A high level phrase for for this sentence might be:

> Pattern: $\ll$goal$\gg$ *comma* so $\ll$action$\gg$
> Concept: $\ll$action$\gg$ achieves $\ll$goal$\gg$

The concept represents that John is robbing the bank to achieve his goal of acquiring a new car. However, there is an intervening step between the bank robbery and John getting the car: John has to take the money and buy the car. The issue is whether the parser should find the causal chain from bank-robbery $\Rightarrow$ get-money $\Rightarrow$ buy-car $\Rightarrow$ possess-car, or just recognize that there is an unspecified causal relation between the clauses "John wanted a car" and "John decided to rob a bank" and pass it on to plan recognition routines.

[2]A more intelligent program might wonder why John just doesn't steal a new car, and skip the middleman.

The two extreme viewpoints on this issue are taken by (1) *syntactic* parsers (e.g. [Woods, 1970; Marcus, 1980]), where the output of the parser is a syntactic parse tree which is then taken as input by a semantic interpreter or inference engine, and (2) *integrated* parsers (e.g. [Dyer, 1983]), where the parser has full access to the conceptual structure being produced, and augments this structure as the text is read. The advantages of syntactic parsing are that the parser can be modularized, and knowledge about language can be represented and manipulated independently of the task domain of the program. The disadvantages are that the parser doesn't have access to potentially relevant information that makes the parsing task easier, such as information for reference and disambiguation.

The strategy taken in THUNDER is that parser extracts all of the language specific information from the input sentences: who the actor is, the action, and the relationships of the clauses. Phrases that recognize relationships between phrase constituents fire demons which use semantic information to figure out the and represent the exact relationship.

3.2 Phrase Representation and Lexical Construction

Researchers in phrasal natural language systems have identified the following desiderata for phrase representation:

- Parsimony: each phrase should represent a specific piece of linguistic knowledge, and duplication of knowledge should be avoided.
- Uniformity: a canonical representation should be used for all linguistic knowledge.
- Processing Independence: the representation of linguistic knowledge should be independent of the language tasks that use the knowledge[Arens, 1986].
- Extensibility: it should be possible to incrementally add new phrases to the system[Jacobs, 1985a].
- Adaptability: the representation of linguistic knowledge should be applicable to new domains[Jacobs, 1985a].
- Learnability: the phrase representation should, in principle, support mechanisms for new phrases to be acquired[Zernik, 1987].

Uniformity of representation and processing independence are both provided by PPARSE/PGEN, but lead to problems meeting the other criteria. PPARSE/PGEN do not make a distinction between linguistic and semantic data in the representation of phrases, so the knowledge representation has to support syntactic class distinctions. This problem is particularly apparent during generation, as there is no mechanism for specifying the syntatic environment ("casting"[Hovy, 1987]) for how an object should be generated, except by changing the representation.

Parsimony is primarily a function of the underlying knowledge representation, and is difficult to achieve totally. In PPARSE/PGEN some duplication cannot be avoided because of the frame-based nature of the representation. The rule "The subject of an active verb is the actor" is expressed in each clause to action-frame pattern as a variable binding, and thus is the knowledge is duplicated in each phrase. If the phrases are organized in a generalization hierarchy, general information can be represented in the high level phrases, and more specific phrases would inherit most of the general information while representing idiosyncratic exceptions[Jacobs, 1985a; Zernik, 1987]. PPARSE/PGEN use d-nets for phrase organization and access, which automatically creates some of this hierarchy. Patterns with variables are indexed higher in the net, while patterns with specific content are indexed lower. However, other pattern constraints in the phrases are not used by the d-net, such as the class restrictions on phrasal variables and applicablity test functions.

Extensibility, adaptability, and learnability of phrases are accomplished by systems that design the phrase representation for integration with the knowledge representation and processing tasks[Jacobs, 1985a; Zernik, 1987]. These programs tightly couple the phrase representation with phrase organization, defeating the processing independence principle. Since PPARSE/PGEN are designed to be independent of a specific knowledge representation, in addition to being independent of processing tasks, phrase organization is not made dependent on the knowledge representation. PPARSE/PGEN are adaptable and extensible, but only in the sense that the user can incrementally add phrases.

3.3 Structural Ambiguity

Structural ambiguity is a property of sentence that have multiple syntactic parse trees. For example, in the sentence:

I saw the man with the telescope.

the prepositional phrase "with the telescope" can be attached to the main clause (the telescope was used to see the man) or the object (the man possessed the telescope). In some cases semantics can be used to resolve the ambiguity. For example, the prepositional phrase attachment in:

I saw the Grand Canyon flying to New York .

can be done by knowing that "the Grand Canyon" isn't something that can fly.

Structural ambiguity is a particular problem for phrasal parsers. To handle structural ambiguity the parser either (1) has to delay phrase application until potentially ambiguous constructs have been parsed, or (2) has to be able to backtrack when a structural ambiguity is recognized.

PPARSE/PGEN do not backtrack. Once a phrase is selected and applied there is no way to "undo" the effects of phrase application. There is also no way to produce multiple parses of structurally ambiguous sentences. For example, using the phrases:

Pattern: ≪ human1 ≫ ≪verb to-see≫ ≪human2≫
Concept: (action type:attend actor:human1 object:human2)

Pattern: ≪action≫ ≪prep-phrase prep:with object:?object≫
Concept: (action instrument:?object)

to parse:

I saw the man with the telescope.

will result in the conceptualization:

```
(action nil
     'type       'attend
     'actor      reader
     'object     the man
     'instrument telescope)
```

With these phrases, the prepositional phrase "with the telescope" is attached to the action, instead of to "the man". Even with the following phrase added to the lexicon:

Pattern: ≪human≫ ≪prep-phrase prep:with object:?object≫
Concept: (human possess:?object)

the human-saw-human pattern would match first, and the result would be the same.

One possible solution would be to add the phrase:

Pattern: ≪action≫ ≪prep-phrase prep:with object:?object≫
Concept: (action object: (human possess:?object))

and then use semantic routines called from the applicability tests of the phrase to choose between the two competing ≪action≫ ≪prep-phrase≫ patterns.

The design decision not to allow the parser to backtrack was made for the following reasons:

1. Backtracking points are difficult to recognize. Since phrases have multiple constituents, PPARSE has to keep a list of top-level nodes in case the next word or phrase matched completes a phrase. Also, since PPARSE can be used to produce a list of concepts (a sequence of actions, for example), it can't recognize a backtrack point when no phrase prefixes match.

2. Parsing problems based on structural ambiguity are sometimes the result of too little semantics to distinguish between phrases. Syntactically ambiguous sentences usually fall into two classes: (1) where semantics can be used to resolve the ambiguity (e.g. "I saw the Grand Canyon flying to New York.", or (2) where there is an ambiguity that *can't* be resolved (e.g. "I saw the man with the telescope"). Backtracking doesn't work generally for either of these cases.

3. Having to backtrack during parsing is the symptom of a deeper problem with symbolic understanding systems. Human readers have the ability to dynamically switch between interpretations as more information becomes available, whereas symbolic natural language systems have to commit to variable bindings and stick with them. For example, in:

John kicked the bucket.
His wife was very upset.
He started to clean up the mess.

The interpretation of the phrase "kicked the bucket" goes from "John is dead" after the second sentence, to the literal interpretation after sentence three.

4. Real backtracking by human readers is fairly rare. The only class of sentences where human reader consciously backtrack are *garden path* sentences, such as "The horse raced past the barn fell." For most applications, sentence of this type don't occur, or can be re-written.

4 Technical Description of PPARSE/PGEN

PPARSE/PGEN were originally written in T[Rees et al., 1984; Slade, 1987], and recently ported to Common Lisp[Steele Jr., 1984]. PPARSE/PGEN make use of Rhapsody representation objects, hash tables, and discrimination nets, so Rhapsody has to be loaded before PPARSE/PGEN.

The file `phrasal_load.lisp` loads the entire PPARSE/PGEN system. There are three main packages:

phrase The phrase package contains the phrase definition and indexing functions. The source is in the file `phrase.lisp`.

pparse The pparse package contains the PPARSE functions and auxiliary routines for parsing with the phrases. The source is in `pparse.lisp`.

pgen The pgen package contains the PGEN functions and auxiliary routines for generation from the phrases. The source is in `pgen.lisp`.

To use PPARSE/PGEN the user has to define a set of phrases use the macro `phrase:define`. The two main functions in PPARSE/PGEN are:

```
(pparse list-of-atoms)
(pgen representation-object)
```

`Pparse` returns the representation objects at the root of the parse tree. `Pgen` returns a list of output words.

4.1 Phrase Definitions

The macro `phrase:define` is used to define the database of lexical and linguistic knowledge that PPARSE/PGEN use. The format of `phrase:define` is:

```
(phrase:define phrase-name
      (comment     comment-string)
      (pattern     list of items)
      (concept     representation object)
      (flags       list of flags)
      (parse-test  list of functions)
      (parse-proc  list of functions)
      (gen-test    list of functions)
      (gen-proc    list of functions))
```

The `comment`, `pattern`, and `concept` sections are required, and all of the other sections are optional. The `phrase-name` part of the pattern is an atom that is used to uniquely identify the phrase. Each section of the phrase definition is described below.

`comment` *comment-string*

The *comment-string* is used to describe the pattern in parsing and generation traces.

`pattern` *list of items*

Each *item* is one of the following: (1) an atom, (2) a representation object, (3) a simple variable, (4) or a phrasal variable. During parsing, the items are matched against the values of the top level nodes in the parse tree. Atoms in the pattern match atoms (using `eq`). Representation objects, simple variables and phrasal variables are matched using unification, so that variables with the same name match the same object. If the phrase is used in generation, each *item* is used to build a new leaf node in the generation tree. Atoms are put into the output buffer as surface words. Representation objects, simple variables, and phrasal variables are de-referenced, and their values are used as the values of the new nodes.

`concept` *representation-object*

The representation object in the concept section is the object that a phrase will re-write to during parsing, and the object that will be matched during generation. Variables in the concept are matched and unified with variables of the same name in the concept and pattern.

`flags` *list of flags*

The flags sections is a list of atoms that control where the concept and the pattern are stored. The atoms that can be used in the **`flags`** section are:

dont-gen Don't use this pattern during generation. If this flag is used, the phrase is not indexed into `*pgen-dnet*`.

dont-parse Don't use this pattern during parsing. If this flag is used, the phrase is not indexed into `*pparse-dnet*`.

`parse-test` *list of functions*
`parse-proc` *list of functions*
`gen-test` *list of functions*
`gen-proc` *list of functions*

The `-test` and `-proc` sections provide a way for arbitrary functions to be called during parsing (sections with the `parse-` prefix) and generation (sections with the `gen-` prefix). The `-test` functions are used to determine phrase applicability; for the phrase to be used, all functions in the `-test` function list must return non-nil. The `-proc` functions are called after the phrase has been selected, and can be used to change values in the tree, re-order nodes in the tree, or to set variable bindings. Each entry in the list of functions is a lambda with no arguments that is evaluated when the phrase is defined. For some purposes it is useful to write macros with static parameters for commonly used `-test` and `-proc` functions.

4.2 Simple Variables, Phrasal Variables and Binding Lists

To find appropriate phrases during parsing, PPARSE matches phrasal patterns to the top level values in the parse tree using *unification*. (For a description of a unification algorithm see [Charniak et al., 1980, pp. 146–149]). The notation used for variables is `?`*variable-name* (e.g. `?human`, `?actor`). Variable of this type are called *simple variables*.

The second kind of variable used in phrasal patterns are called *phrasal variables*. Phrasal variables are used to specify variables that will only match conceptual objects in a set of named classes. The format of a phrasal variable is `?*`*variable-name*`+`*class* or `?*`*variable-name*`+(`*class-list*`)`. Phrasal variables work like simple variables during matching, but are also restricted to only match representation objects of the classes specified. For example, `?*hum+&human` only matches objects of class `&human`, and `?*animate+(&human &cat &dog)` matches objects of class `&human`, `&cat`, or `&dog`.

Phrasal and simple variables that have the same variable name unify with each other. Variables are scoped over the entire phrase, so that `?actor` in the concept section and `?*actor+(&human &animal)` in the pattern must match the same representation object, which must be of class `&human` or `&animate` (from the phrasal variable restrictions).

The variable bindings for each phrase are kept in a global hash table called `*pparse-bindings*`.[3] The key of each entry in the hash table is the variable name, and the value of the entry is the variable's binding. This hash table is available to be used by phrase's `-test` and `-proc` procedures. The hash table is cleared before each phrase match, so it only holds the bindings for the current phrase. In addition to variable bindings, each representation object is put into the hash table keyed by its name, so that representation objects in the concept and pattern can be accessed by `-proc` and `-test` functions.

4.3 Global Variables

During parsing and generation, PPARSE/PGEN build a tree as patterns are matched and incorporated. Each node in the tree is an input or output word, an element of a phrase pattern, or a phrase concept. For some words or patterns, the user may want to reorganize the tree, or change node values using the `-test` or `-proc` sections of a phrase. The following global variables are accessible by the user:

pparse-dnet Discrimination net for indexing phrases by their patterns.

pgen-dnet Discrimination net for indexing phrases by their concepts.

pparse-trace Trace object for PPARSE.

pparse-trace+ Trace object for extended, debugging trace of PPARSE.

pgen-trace Trace object for PGEN.

pgen-trace+ Trace object for extended, debugging trace of PGEN.

pparse-root The root node of the parse tree.

[3]`*pparse-bindings*` is a Rhapsody hash table, not a Common Lisp hash table.

pgen-root Root node of the generation tree.

pparse-bindings A hash table of the variable bindings used in matching concepts or patterns.

pgen-node The current pgenode being generated.

pgen-output-buffer List containing the words output by PGEN.

4.4 PPARSE Nodes

Each node in the phrasal parse tree is a structure called a *ppnode*. Ppnodes have the following components:

name Unique identifier of the node

value The conceptual content of the node.

prev Pointer to the previous node. The value of `ppnode:prev` will either be the node that precedes the node in a phrase, or the parent node for nodes that begin a phrase.

constits For phrases, a list of the constituent nodes in the phrase. In other words, the children of the node in the parse tree.

lex The entry in the phrasal lexicon used to build the node.

Each component is accessible and settable by the function (`ppnode:`*componentname node*). The function (`ppnode:create :rest` *name*) creates new ppnodes. The optional argument *name* will be used as a prefix on the name component of the node.

4.5 PPARSE's Parsing Algorithm

For each word in the input list:

1. Construct a new node for the word.
2. Find and incorporate new phrases into the parse tree:
    (a) Make a list of the values of the top level nodes in the parse tree.
    (b) Make a list of candidate patterns to match by making a pattern out of the list of top level values and the successive cdrs of the list. These candidate patterns are tried in order, so that the parser matches the longest pattern first. The last pattern tried will be the value of the most recently created node.
    (c) For each candidate pattern:
        i. Search `*pparse-dnet*` for candidate phrases. The d-net returns a list of phrases in order of the specificness of their match with the search pattern.
        ii. For each candidate phrase:

A. Check that phrasal variable restrictions are met.

B. Run the candidate phrases `parse-tests`.

C. If the restrictions are met and the `parse-tests` return non-nil, build a new node for the matched phrase, set the father and next pointers of the nodes in the pattern matched, and set `*pparse-root*` to point at the new node. Repeat step 2 until no new patterns are found.

4.6 PGEN Nodes

Each node in the phrasal generation tree is a structure called a *pgenode*. Pgenodes have the following components:

name Unique identifier of the node

value The conceptual content of the node.

father Pointer to the parent node.

constits List of the children of the node, in their lexical order.

next Pointer to the next node to generate. For internal nodes, `pgenode:next` points to first child of the node to generate. For leaf nodes, `pgenode:next` points to the next leaf node to generate.

lex The entry in the phrasal lexicon used to build the node.

Each component is accessible and settable by the function (`pgenode:`*componentname node*). The function (`pgenode:create :rest` *name*) creates new pgenodes. The optional argument *name* will be used as a prefix on the name component of the node.

4.7 PGEN Generation Algorithm

Set `*pgen-output-buffer*` to nil, and build a new pgen node for the input concept and set `*pgen-node*` to point at the node.

For the current `*pgen-node*`:

1. If the value of `*pgen-node*` is `nil`, and reverse and return `*pgen-output-buffer*`.

2. If the concept of `*pgen-node*` is an atom, cons it on to `*pgen-output-buffer*`, set `*pgen-node*` from the next pointer of the current `*pgen-node*` and repeat, else

3. Search `*pgen-dnet*` for candidate phrases. The d-net returns a list of phrases in order of the specificness of their match with the search concept.

4. For each candidate phrase:

   (a) Check that phrasal variable restrictions are met.

   (b) Run the candidate phrases `gen-tests`.

(c) If the restrictions are met and the **`gen-tests`** return non-nil, for each element of the pattern section of the phrase:

   i. Build a new node for the pattern element.

   ii. Set the father pointer of the new node to **`*pgen-node*`**

(d) Set the constits component of pgen node to the list of newly created nodes.

(e) Set the next component of **`*pgen-node*`** to the first constituent node, and the next component of each constituent node to the next node in the list.

(f) Set **`*pgen-node*`** from the next component of the current **`*pgen-node*`**, and repeat.

4.8 Testing and Procedure Functions

The functions described in this section are built into PPARSE/PGEN for use in **`-test`** and `-proc` sections of phrases. Each element in the **`-test`** and `-proc` sections of phrases is a function with no arguments. The **`-test`** and `-proc` sections of a phrase are evaluated when the phrase is read. All of the routines described in this section return a procedure (lambda closure) of no arguments that is evaluated when the phrase is applied. The arguments to these functions are used in the routines to access run-time variables and global data structures.

The following functions can be used in **`pparse-`** or **`gen-test`** sections of phrases:

(**`pparse:check-var`** *variable*) Returns true if *variable* is bound.

(**`pparse:check-null-var`** *variable*) Complement of **`pparse:check-var`**.

(**`pparse:check-class`** *variable class-list*) Returns true if the class of the binding of *variable* is a member of *class-list.*

(**`pparse:check-link-on-var`** *variable link*) Returns true if the instance bound to *variable* has a non null *link.*

The following functions can be used in the **`parse-proc`** section of phrases:

(**`pparse:add-node-bef`** *variable*) Add a new node before the **`*pparse-root*`** node, and set its value to the binding of *variable.*

(**`pparse:add-node-aft`** *variable*) Add a new node after the **`*pparse-root*`** node, and set its value to the binding of *variable.*

(**`pparse:set-slot`** *variable slot-name value*) Set *slot-name* on the binding of *variable* to *value.*

(**`pparse:set-slot-from-var`** *variable slot-name variable2*) Set *slot-name* on the binding of *variable* to the binding of *variable2.*

(**`pparse:set-slot-from-proc`** *variable slot-name procedure*) Set *slot-name* on the binding of *variable* to the the value obtained by evaluating *procedure. Procedure* is a lambda with no arguments.

(`pparse:add-link` *link variable*) Add a *link-name* link from the value of `*pparse-root*` to the binding of *variable*.

(`pparse:replace-root-val` *procedure*) Replace the value of `*pparse-root*` with the value of evaluating *procedure*. *Procedure* is a lambda with no arguments.

The following functions can be used in the **`gen-test`** section of phrases:

(`pgen:prev-in-class` *class-list*) Returns true if the class of the value of the previous node in the generation tree is a member of *class-list*.

(`pgen:prev-not-in-class` *class-list*) Complement of `pgen:prev-in-class`.

(`pgen:prev-eq` *variable*) Returns true if the value of the previous node is `eq` to the binding of *variable*.

(`pgen:check-var-slot-val` *variable slot test-value*) Returns true if the *slot* value of the binding of *variable* is `eq` to *test-value*.

The following function can be used in the **`gen-proc`** section of phrases:

(`pgen:replace-place-holder` *tag value*) Replace occurrences of the atom *tag* in the pattern with *value*.

4.9 Tracing and Debugging Features

There are four user-settable flags for parsing and generation tracing:

pparse-trace PPARSE trace showing phrase application and the resulting conceptual object for each phrase used.

pparse-trace+ Shows candidate phrases that are returned from the d-net. Primarily used for debugging.

pgen-trace PGEN trace showing phrase application and the resulting concepts.

pgen-trace+ For extended debugging trace of PGEN.

Tracing is turned on using the call:

```
(setf (trace:on trace-flag) t)
```

And turned off with:

```
(setf (trace:on trace-flag) nil)
```

PPARSE/PGEN trace output defaults to `standard-output`, but can be directed to another output stream with the call:

```
(setf (trace:stream trace-flag) new-output-stream)
```

The indent level can be set with:

```
(setf (trace:indent trace-flag) indent-column)
```

The default is column 0.

The following functions are useful for PPARSE debugging:

`(pparse:top-level-nodes)` Returns a list of pparse node at the top level of the parse tree.

`(pparse:top-level-vals)` Like `pparse:top-level-nodes`, but returns a list of the values of the top level nodes.

`(pparse:dump-tree)` Prints the content of each node in the parse tree by traversing the tree depth-first.

`(pparse:map-tree` *function*`)` Returns a list of the values of applying *function* to each of the nodes in the parse tree depth-first.

`(pparse:walk-tree` *function*`)` Like `pparse:map-tree`, but doesn't return anything useful.

The following functions are useful for PGEN debugging:

`(pgen:dump-tree)` Prints the content of each node in the generation tree by traversing the tree depth-first.

`(pgen:map-tree` *function*`)` Returns a list of the values of applying *function* to each of the nodes in the generation tree depth-first.

`(pgen:walk-tree` *function*`)` Like `pgen:map-tree`, but doesn't return anything useful.

4.10 Packages for Common Linguistic Problems

While PPARSE/PGEN doesn't presume to have the final word on the proper way to integrate linguistic and semantic knowledge during parsing and generation, there are two problem areas that every natural language program has to deal with: (1) *pronoun reference*, using pronouns to reference items that have previously been mentioned, and (2) *lexical disambiguation*, selecting correct word senses from words that have multiple meanings. Since these problems occur in almost every natural language sentence, PPARSE/PGEN provides built-in mechanisms to for the user who wants to work around these problems. This section describes both problems, and the solutions provided by PPARSE/PGEN.

4.10.1 Pronoun Reference

Pronouns are used to refer to people and things that have been previously mentioned in a text. There are three types of problems in parsing pronouns:

1. When should the reference be resolved? Should the reference be resolved when the pronoun is read, or should the parse wait until a sentence or clause boundary is reached in order to exploit semantic constraints? For example, in:

John hit Bill, and he ...
... hurt his knuckles.
... dropped to the floor.

the contrasting completions of the sentence show that the pronoun *he* cannot be resolved until the end of the sentence. A related question is whether the parser should commit to a particular resolvent, or find a potential resolvents and wait for more evidence. For example, in:

John looked at Bill, and he said "Hello."

the most probable speaker is Bill, but it could be John (this would more likely be written "John looked at Bill and said 'Hello' ").

2. What strategies should be used to find the reference? Some sources of constraints on the reference of a pronoun are (1) the pronoun itself, i.e. *he* can only refer to male humans, (2) the range of possible referents that have been mentioned or are currently in focus[Sidner, 1983], (3) semantic constraints, such as that the person hitting will hurt his knuckles, and that the person hit will drop to the floor.

3. How should forward or null referents be recognized and handled? Some pronouns are used as initial mentions of people. For example, if a text began:

He was tall, dark, and handsome.

the parser should produce the concept for a tall, dark, handsome, male human, even though there is no referent for *he*. And for:

It was a dark and stormy night.

the parser has to recognize that *it* refers to the time of day, and the sentence is being used to convey the setting.

To handle pronoun reference, PPARSE/PGEN provide a set of `parse-proc` and `gen-test` functions in the `lexref` package. These functions save potential referents in global data structures, fire demons during parsing to resolve pronoun references, and test for pronoun applicability during generation. To illustrate how the functions work, consider the definitions of the phrases "john" and "he":

```
(phrase:define 'ph-john
    (comment "John")
    (pattern 'john)
    (concept (human 'proper-name1
        'first-name 'john
        'gender     'male))
    (parse-proc (lexref:parse-save-ref *lexref-people*))
    (gen-test (lexref:not-most-recent-ref
        ?proper-name1
        &proper-name1
        *lexref-people*))
    (gen-proc (lexref:gen-save-ref *lexref-people*)))
```

```
(phrase:define 'ph-he
     (comment "he")
     (pattern 'he)
     (concept (human 'male-pronoun1
                  'gender 'male
                  'number ?num))
     (parse-proc (lexref:spawn-resolver-demon
                     ?male-pronoun1
                     'nomative-pronoun))
     (gen-test (lexref:most-recent-ref
                  ?male-pronoun1
                  &male-pronoun1
                  *lexref-people*)))
```

The `parse-proc` in the phrase for "John" uses the function `lexref:save-ref` to save the conceptualization for John in the global list `*lexref-people*`. The `parse-proc` for "he" uses `lexref:spawn-resolver-demon` to search `*lexref-people*` for a concept matching the concept of the "he" phrase. The reason that a demon is spawned, instead of doing the search when "he" is parsed, is so that the search function can use the context conceptualization in which "he" was used.

During generation, `gen-test` functions are used to decide when to use a pronoun. The function `lexref:most-recent-ref` tests `*lexref-people*` to see if the object being generated is the most recent object that matches the conceptualization in the phrase. So, if the conceptual object for "John" is being generated, and "John" is the most recent male human who has been generated, the pronoun "he" will be used.

The `lexref` package used the following global variables:

lexref-people List of person references, in order of use.

lexref-things List of "thing" references, in order of use.

The following functions are used in `parse-` and `gen-proc` functions:

(`lexref:spawn-resolver-demon` *var type*) spawns resolver demons defined in the file `pparse_demon`. The binding of *var* is used as a argument to the demon, and *type* is used to identify the demon.

(`lexref:parse-save-ref` *list*) pushes the content of `*pparse-root*` in the global variable *list*.

(`lexref:gen-save-ref` *list*) pushes the content of `*pgen-node*` in the global variable *list*.

(`lexref:parse-search-for-ref` *list*) searches global variable *list* for a concept matching the contents of `*pparse-root*`, and replaces `*pparse-root*` with what it finds.

(`lexref:gen-search-for-ref` *list*) searches global variable *list* for a concept matching the contents of `*pgen-node*`, and replaces `*pgen-node*` with what it finds.

The following functions are used in **`gen-test`** functions:

(**`lexref:most-recent-ref`** *var matcher list*) returns true if the binding for *var* is `eq` to the most recent object in *list* that matches *matcher*,

(**`lexref:not-most-recent-ref`** *var matcher list*) is the complement of `lexref:most-recent-ref`.

(**`lexref:not-mentioned`** *var list*) returns true if the binding for *var* is not contained in *list*. This function is useful for using phrases that generate full descriptions of objects the first time that they are generated (e.g. "a large white cockatoo" vs. "the bird").

(**`lexref:mentioned`** *var list*) is the complement of **`lexref:not-mentioned`**.

Example pronoun definitions are given in the file `pptest_pronoun.lisp`, and sample pronoun resolution demons are defined in the file `parse_demon.lisp`. The function `pparse:run-demons` is used to run demons spawned by the **`lexref`** package.

4.10.2 Lexical Disambiguation

To integrate top-down information into the parsing process, PPARSE uses special processing to match ambiguous words. For example, the word "bank" can be a river bank (as in "the west bank of the Mississippi") or a financial institution (as in the kind of bank John robs). This ambiguity is represented by a special representation class called `ambiguous-word` which holds the competing senses of the lexical item. The entry for "bank" is:

```
(phrase:define
     (comment "bank (ambiguous)")
     (pattern 'bank)
     (concept (ambiguous-word
                    'sense0 (location
                                   'of 'river
                                   'prep 'next-to)
                    'sense1 (financial-institution
                                   'name ?name))))
```

When an attempt is made to match an element of a constituent pattern against an ambiguous word, each of the senses of the word a matched. If any of the senses matches, that sense is selected and the pattern is used. For example, the parse tree for "John robbed a bank" before the application of the human-rob-bank pattern above is shown in figure 3. Since the human-rob-bank pattern matches if the financial-institution sense of bank is used, the human-rob-bank pattern is used to construct the meaning of the sentence. The resulting parse tree after the human-rob-bank is matched is shown in figure 4.

The representation class `ambiguous-word` can hold up to 10 word senses, labeled `sense0` – `sense9`. Phrases and concepts can treat concepts of the `ambiguous-word` class just as any other representation class; they can be included in patterns, concepts, and phrasal variables.

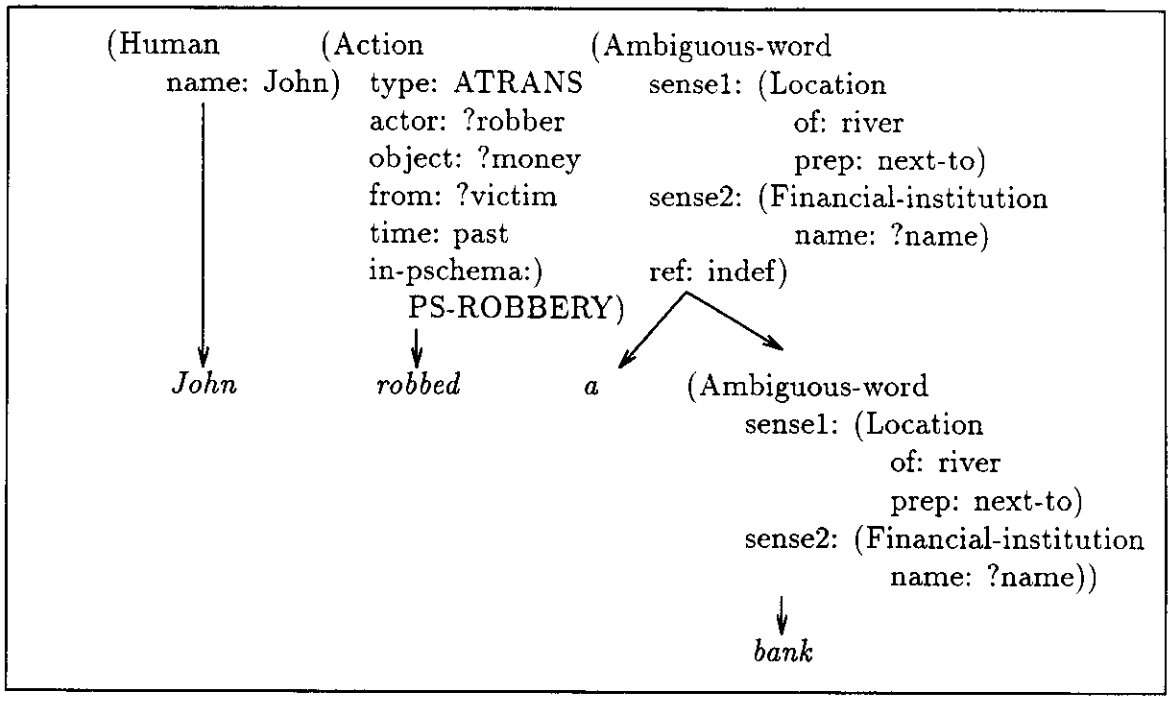

Figure 3: Example Parse Tree Before Matching the Syntax Pattern

5 Example Lexicon and Trace

This section contains sample lexical entries for the sentence "John picked up the ball and put it in the box", and a trace of PPARSE/PGEN parsing and generating the sentence.

5.1 Lexical Entries

Basic noun patterns map the words to their associated concepts. The **gen-tests** for regular nouns test for an unbound `?ref` variable to make sure that the article has been generated.

```
;;; john

(phrase:define 'ph-1
   (comment "John")
   (pattern 'john)
   (concept (human 'ph-1-con
               'name 'john
               'gender 'male)))

;;; ball

(phrase:define 'ph-5
   (comment "ball")
   (pattern 'ball)
```

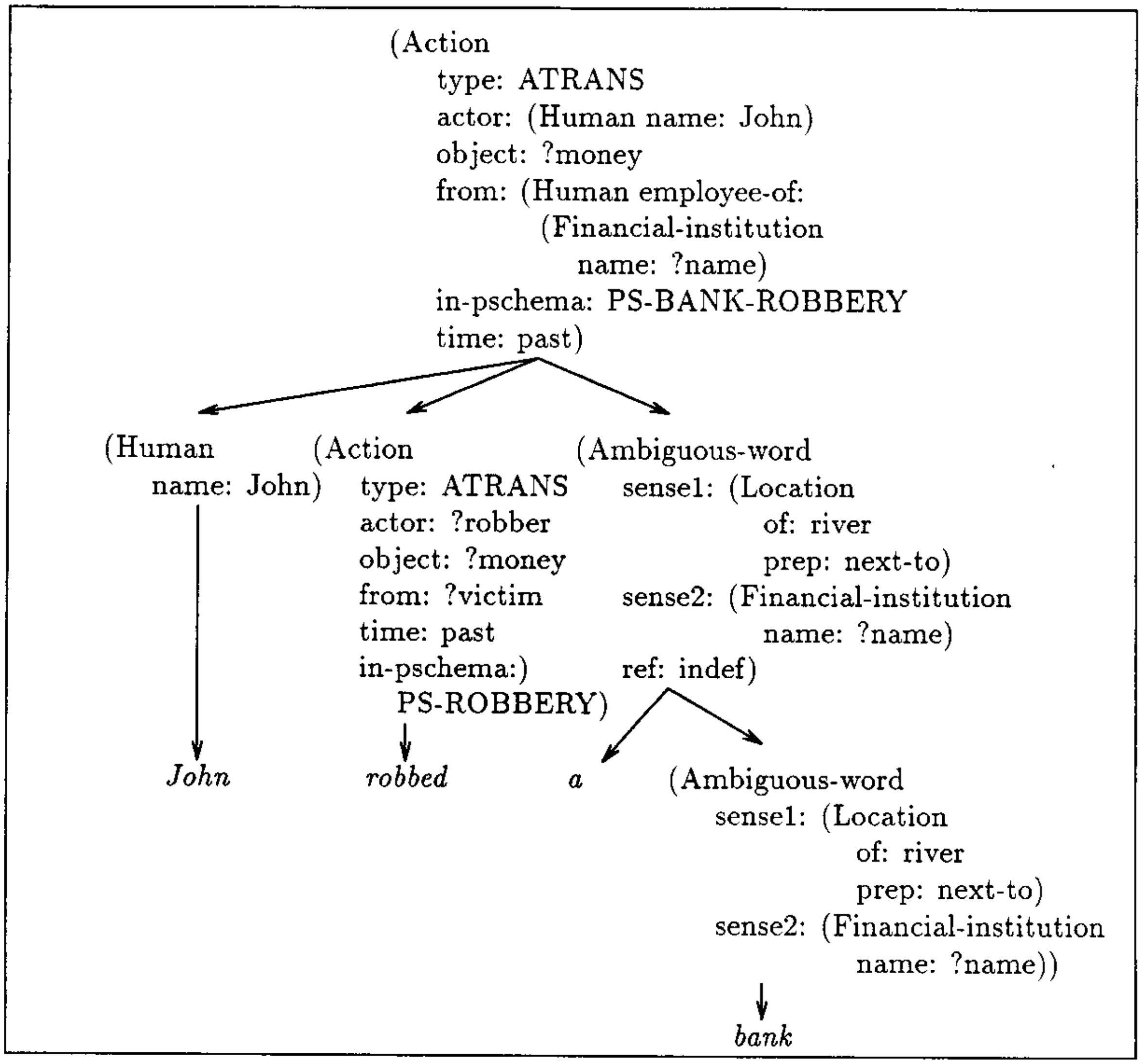

Figure 4: Example Parse Tree After Matching the Syntax Pattern

```
   (gen-test (pparse:check-null-var ?ref))
   (concept (phys-obj 'ph-5-con
                'ref  ?ref
                'type 'game-obj
                'name 'ball)))

;;; box

(phrase:define 'ph-12
   (comment "box")
   (pattern 'box)
   (gen-test (pparse:check-null-var ?ref))
   (concept
      (phys-obj 'ph-12-con
         'ref  ?ref
         'type 'container
         'name 'box)))
```

The pattern for "it" doesn't have a concept. Instead, the `parse-proc` searches the parse tree for the most recently seen `phys-obj`, and uses it as the value of the phrase.

```
;;; it

(phrase:define 'ph-it
  (comment "it")
  (flags 'dont-gen)
  (pattern 'it)
  (parse-proc
    #'(lambda ()
        (let ((node (pparse:pronoun-search (list &phys-obj))))
          (and node
               (setf (ppnode:value *pparse-root*)
                     (ppnode:value node)))))))
```

The representation class `verb` is used to hold the type of verb and associated modalities (tense, number, etc.)

```
;;; picked

(phrase:define 'ph-2
   (comment "picked")
   (pattern 'picked)
   (concept (verb 'ph-2-con
               'name 'to-pick
               'tense 'past)))
```

```
;;; put

(phrase:define 'ph-9
  (comment "put")
  (pattern 'put)
  (concept
      (verb 'ph-9-con
         'name 'to-put)))
```

The pattern for the verbal "picked up" is the verb to-pick with a variable tense followed by the word "up". If there were patterns for various verb modalities, this pattern would also be used for "pick up", "had been picked up", etc.

```
;;; <verb:to-pick> up

(phrase:define 'ph-3
   (comment "<Verb:to-pick> up")
   (pattern (verb 'ph-3-verb
                'name 'to-pick
                'tense ?tense)
           'up)
   (concept (verb 'ph-3-con
                'name 'to-pick-up
                'tense ?tense)))
```

The articles "a" and "the" are represented as with different types with different types, so that the `article` class can be used to recognize the beginning of noun phrases.

```
;;; the

(phrase:define 'ph-4
   (comment "the")
   (pattern 'the)
   (concept (article 'ph-4-art
                'type 'definite)))

;;; a

(phrase:define 'ph-4a
   (comment "a")
   (pattern 'a)
   (concept (article 'ph-4-arta
                'type 'indefinite)))
```

The next two patterns put together noun and prepositional phrases, respectively. The pattern for the "in object" prepositional phrase uses a phrasal variable to restrict "object" to members of the representational class `phys-obj`.

```
;;; <article> <phys-obj>

(phrase:define 'ph-6
   (comment "<article> <phys-obj>")
   (pattern (article 'ph-6-art
               'type ?ref)
            (phys-obj 'ph-6-obj
               'name ?name
               'type ?type))
   (concept (phys-obj 'ph-6-con
               'ref  ?ref
               'type ?type
               'name ?name)))

;;; in <obj>

(phrase:define 'ph-11
   (comment "in <phys-obj>")
   (pattern 'in ?*obj+&phys-obj)
   (concept
      (prep 'ph-11-con
         'name 'in
         'obj  ?obj)))
```

The following two patterns put subject-verb-object clauses together into actions.

```
;;; <?human> <verbal:pick-up> <?obj>

(phrase:define 'ph-7
   (comment "<human> <pick-up> <phys-obj>")
   (pattern ?*human+&human
            (verb 'ph-7-verb
               'name 'to-pick-up)
            ?*obj+&phys-obj)
   (concept
      (action 'ph-7-con1
         'type   'grasp
         'actor  ?human
         'object ?obj
         'instr (action 'ph-7-con2
                   'type   'move
                   'actor  ?human
                   'object 'fingers
                   'to     ?obj))))
```

```
;;; human <to-put> obj1 <in obj2>

(phrase:define 'ph-13
    (comment "<human> <put> <obj1> <in obj2>")
    (pattern ?*human+&human
             (verb 'ph-13-verb
                'name 'to-put)
             ?*obj1+&phys-obj
             (prep 'ph-13-prep
                'name 'in
                'obj  ?obj2))
   (concept
      (action 'ph-13-con
         'type 'ptrans
         'actor ?human
         'object ?obj1
         'to    ?obj2
         'instr ?instr-act))
   (gen-test (pparse:check-null-var ?instr-act)))
```

The next pattern fills in the actor in the second clause of the sentence. When this pattern is matched, "John picked up the ball" has been parsed into an action, and "put" has matched the verb concept. The parse proc adds a node before the ***pparse-root***, so act-and-verb is transformed into act-and-actor-verb.

```
;;; <act> and <verb>

(phrase:define 'ph-8
   (comment "<act> and <verb>"
            "Fill in the elliptical actor of the second act")
   (pattern
      (action 'ph-8-act
         'actor ?human)
      'and
      ?*verb1+&verb)
   (parse-proc (pparse:add-node-bef ?human)))
```

The next pattern recognizes the sequence of acts, and puts them together in an instrumental relation.

```
(phrase:define 'ph-14
   (comment "<grasp-act> and <ptrans-act>"
            "The Grasp is inferred to be instrumental to the Ptrans")
   (pattern (action 'ph-14-act1
               'type   'grasp
```

```
                'actor  ?human
                'object ?obj1
                'instr  ?act1)
             'and
             (action 'ph-14-act2
                'type   'ptrans
                'actor  ?human
                'object ?obj1
                'to     ?obj2))
   (concept
      (action 'ph-14-con
         'type 'ptrans
         'actor ?human
         'object ?obj1
         'to     ?obj2
         'instr  (action 'ph-14-con2
                     'type   'grasp
                     'actor  ?human
                     'object ?obj1
                     'instr  ?act1))))
```

5.2 Program Trace

The call to pparse will create a concept from the list of words, returning the concept. The call to pgen will take that concept, and return the generated list of atoms.

```
> (pgen (pparse '(john picked up the ball and put it in the box))

---------------------------------
Processing word: john
---------------------------------
Trying phrase #{ph-1}

Found Phrase: #{ph-1}
-----> John <-----

Created Concept:
(human &human.3
   name   john
   gender male)
```

PPARSE works in a cycle adding words and applying phrases. The Trying phrase... message is printed for each phrase that is returned from the *pparse-dnet*.

```
---------------------------------
Processing word: picked
```

```
---------------------------------
Trying phrase #{ph-2}

Found Phrase: #{ph-2}
-----> Picked <-----

Created Concept:
(verb &verb.7
   name  to-pick
   tense past)

---------------------------------
Processing word: up
---------------------------------
Trying phrase #{ph-3}

Found Phrase: #{ph-3}
-----> <Verb:to-pick> up <-----

Created Concept:
(verb &verb.8
   name  to-pick-up
   tense past)
Trying phrase #{ph-7}
```

Once human-pick-up has been recognized, calls to the dnet returns the potential pattern human-pick-up-object (ph-7). Since the object is not a `phys-obj`, the pattern in not applied.

```
---------------------------------
Processing word: the
---------------------------------
Trying phrase #{ph-7}
Trying phrase #{ph-4}

Found Phrase: #{ph-4}
-----> the <-----

Created Concept:
(article &article.5
   type definite)
Trying phrase #{ph-7}

---------------------------------
Processing word: ball
---------------------------------
```

```
Trying phrase #{ph-5}

Found Phrase: #{ph-5}
-----> ball <-----

Created Concept:
(phys-obj &phys-obj.5
   ref  ?ref
   type game-obj
   name ball)
Trying phrase #{ph-6}

Found Phrase: #{ph-6}
-----> <article> <phys-obj> <-----

Created Concept:
(phys-obj &phys-obj.6
   ref  definite
   type game-obj
   name ball)
Trying phrase #{ph-7}
```

When "the ball" is parsed into a `phys-obj`, the clause pattern is applied.

```
Found Phrase: #{ph-7}
-----> <human> <pick-up> <phys-obj> <-----

Created Concept:
(action &action.12
   type   grasp
   actor  &human.3
   object &phys-obj.6
   instr  &action.13)

---------------------------------
Processing word: and
---------------------------------
Trying phrase #{ph-8}

---------------------------------
Processing word: put
---------------------------------
Trying phrase #{ph-8}
Trying phrase #{ph-9}
```

Since PPARSE tries to find the longest phrase it can at every point in the parse, it tries the action-and-verb pattern before the "put" pattern. The phrasal variable `?*verb1+&verb`

in the pattern of ph-8 is indexed as a simple variable in the d-net, which is why ph-8 is returned. The class constraint from the phrasal variable is tested after the pattern is returned, and ph-8 is rejected because the atom "put" is not of class **verb**.

```
Found Phrase: #{ph-9}
-----> put <-----

Created Concept:
(verb &verb.9
   name to-put)
Trying phrase #{ph-8}

Found Phrase: #{ph-8}
-----> <act> and <verb> <-----
-----> Fill in the elliptical actor of the second act <-----

Created Concept:
(verb &verb.9
   name to-put)

--------------------------------
Processing word: it
--------------------------------
Trying phrase #{ph-it}

Found Phrase: #{ph-it}
-----> it <-----

Created Concept:
(phys-obj &phys-obj.6
   ref  definite
   type game-obj
   name ball)

--------------------------------
Processing word: in
--------------------------------
Trying phrase #{ph-11}
```

Processing of "the box" is the same as for "the ball", so the trace is deleted here, up the where the prepostional phrase pattern matches.

```
Trying phrase #{ph-11}

Found Phrase: #{ph-11}
-----> in <phys-obj> <-----
```

```
Created Concept:
(prep &prep.3
   name in
   obj  &phys-obj.8)
Trying phrase #{ph-13}

Found Phrase: #{ph-13}
-----> <human> <put> <obj1> <in obj2> <-----

Created Concept:
(action &action.14
   type   ptrans
   actor  &human.3
   object &phys-obj.6
   to     &phys-obj.8
   instr  ?instr-act)
Trying phrase #{ph-14}

Found Phrase: #{ph-14}
-----> <grasp-act> and <ptrans-act> <-----
-----> The Grasp is inferred to be instrumental to the Ptrans <-----

Created Concept:
(action &action.15
   type   ptrans
   actor  &human.3
   object &phys-obj.6
   to     &phys-obj.8
   instr  &action.16)

Processing Complete

Result of Parse

(action &action.15
   type   ptrans
   actor  &human.3
   object &phys-obj.6
   to     &phys-obj.8
   instr  &action.16)
```

The result of the parse is the action `&action.15`, which is then the argument to PGEN.

```
Generating:
```

```
(action &action.15
   type   ptrans
   actor  &human.3
   object &phys-obj.6
   to     &phys-obj.8
   instr  &action.16)
trying phrase #{ph-13}
trying phrase #{ph-14}
```

The phrase ph-13 is rejected on generation by the **gen-test**, which checks for a null instrumental action. When ph-14 is applied, the generated is recursively called on the constituents in the phrase pattern.

```
Applying phrase: #{ph-14}
-----> <grasp-act> and <ptrans-act> <-----
-----> The Grasp is inferred to be instrumental to the Ptrans <-----
Generating:
(action &action.18
   type   grasp
   actor  &human.3
   object &phys-obj.6
   instr  &action.13)
trying phrase #{ph-7}

Applying phrase: #{ph-7}
-----> <human> <pick-up> <phys-obj> <-----
Generating:
(human &human.3
   name   john
   gender male)
trying phrase #{ph-1}

Applying phrase: #{ph-1}
-----> John <-----

---------------------------------
Generating word: john
---------------------------------
Generating:
(verb &verb.10
   name to-pick-up)
trying phrase #{ph-3}

Applying phrase: #{ph-3}
-----> <Verb:to-pick> up <-----
```

```
Generating:
(verb &verb.11
   name  to-pick
   tense ?tense)
trying phrase #{ph-2}

Applying phrase: #{ph-2}
-----> Picked <-----

---------------------------------
Generating word: picked
---------------------------------

---------------------------------
Generating word: up
---------------------------------
Generating:
(phys-obj &phys-obj.6
   ref  definite
   type game-obj
   name ball)
trying phrase #{ph-5}
trying phrase #{ph-6}

Applying phrase: #{ph-6}
-----> <article> <phys-obj> <-----
Generating:
(article &article.7
   type definite)
trying phrase #{ph-4}

Applying phrase: #{ph-4}
-----> the <-----

---------------------------------
Generating word: the
---------------------------------
Generating:
(phys-obj &phys-obj.9
   name ball
   type game-obj)
trying phrase #{ph-5}

Applying phrase: #{ph-5}
-----> ball <-----
```

```
----------------------------------
Generating word: ball
----------------------------------
```

After recursively descending through the grasp act, PGEN generates the second constituent of ph-14, which is the atom "and". After that, the ptrans action is generated.

```
----------------------------------
Generating word: and
----------------------------------
Generating:
(action &action.17
   type   ptrans
   actor  &human.3
   object &phys-obj.6
   to     &phys-obj.8)
trying phrase #{ph-13}

Applying phrase: #{ph-13}
-----> <human> <put> <obj1> <in obj2> <-----
Generating:
(human &human.3
   name   john
   gender male)
trying phrase #{ph-1}

Applying phrase: #{ph-1}
-----> John <-----

----------------------------------
Generating word: john
----------------------------------
Generating:
(verb &verb.12
   name to-put)
trying phrase #{ph-9}

Applying phrase: #{ph-9}
-----> put <-----

----------------------------------
Generating word: put
----------------------------------
Generating:
```

```
(phys-obj &phys-obj.6
   ref  definite
   type game-obj
   name ball)
trying phrase #{ph-5}
trying phrase #{ph-6}
```

The text "the ball" is generated in the same way as before. For brevity, trace is deleted here.

```
Generating:
(prep &prep.4
   name in
   obj  &phys-obj.8)
trying phrase #{ph-11}

Applying phrase: #{ph-11}
-----> in <phys-obj> <-----

---------------------------------
Generating word: in
---------------------------------
Generating:
(phys-obj &phys-obj.8
   ref  definite
   type container
   name box)
trying phrase #{ph-12}
trying phrase #{ph-6}
```

The text "the box" is generated in the same way as "the ball".

```
Processing Complete
(john picked up the ball and john put the ball in the box)
```

References

Arens, Y. (1986). *Cluster: An Approach to Contextual Language Understanding.* PhD thesis, Computer Science Division, University of California, Berkeley. Report UCB/CSD 86/293.

August, S. E. and Dyer, M. G. (1985a). Analogy recognition and comprehension in editorials. In *Proceedings of the Seventh Annual Conference of the Cognitive Science Society (CogSci-85)*, pages 228–235, Irvine, CA.

August, S. E. and Dyer, M. G. (1985b). Understanding analogies in editorials. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 845–847, Los Angeles, CA.

Bresnan, J. and Kaplan, R. M. (1982). Lexical-functional grammar: A formal system for grammatical representation. In Bresnan, J., editor, *The Mental Representation of Grammatical Relations.* Cambridge University Press, Cambridge, MA.

Charniak, E., Riesbeck, C. K., and McDermott, D. V. (1980). *Artificial Intelligence Programming.* Lawrence Erlbaum, Hillsdale, NJ.

Dyer, M. G. (1983). *In-Depth Understanding: A Computer Model of Integrated Processing for Narrative Comprehension.* MIT Press, Cambridge, MA.

Fillmore, C. (1968). The case for case. In Bach, E. and Harns, R. T., editors, *Universals in Linguistic Theory*, pages 1–90. Holt, Reinhart and Winston, Chicago, IL.

Goldman, S. R., Dyer, M. G., and Flowers, M. (1987). Precedent-based legal reasoning and knowledge acquisition in contract law: A process model. In *Proceedings of the First International Conference on Artificial Intelligence and Law*, pages 210–221, Boston, MA.

Hovy, E. H. (1987). *Generating Natural Language Under Pragmatic Constraints.* PhD thesis, Computer Science Department, Yale University, New Haven, CT. Research Report 521.

Jacobs, P. S. (1985a). *A Knowledge-Based Approach to Language Production.* PhD thesis, Computer Science Division, University of California, Berkeley. Report UCB/CSD 86/254.

Jacobs, P. S. (1985b). Phred: A generator for natural language interfaces. Technical Report Report UCB/CSD 85/198, Computer Science Division, University of California, Berkeley.

Kay, M. (1979). Functional grammar. In *Proceedings of the Fifth Annual Meeting of the Berkeley Linguistics Society*, pages 142–158.

Marcus, M. P. (1980). *Theory of Syntactic Recognition for Natural Languaguage.* MIT Press, Cambridge, MA.

Pereira, F. C. N. and Warren, D. H. (1980). Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13(3):231–278.

Rees, J. A., Adams, N. I., and Meehan, J. R. (1984). *The T manual.* Computer Science Department, Yale University, New Haven, CT., fourth edition.

Reeves, J. F. (1988). Ethical understanding: Recognizing and using belief conflict in narrative understanding. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-88)*, St Paul, MN.

Reeves, J. F. (1989). Computing value judgements during story understanding. In *Proceedings of the Eleventh Annual Conference of the Cognitive Science Society (CogSci-89)*. Ann Arbor, MI.

Reeves, J. F. (Forthcoming). *Computational Ethics: A Process Model of Belief Conflict and Resolution for Narrative Understanding.* PhD thesis, UCLA Artificial Intelligence Laboratory, University of California, Los Angeles.

Schank, R. C. (1973). Identification of the conceptualizations underlying natural language. In Schank, R. C. and Colby, K. M., editors, *Computer Models of Thought and Language.* W. H. Freeman, San Francisco.

Schank, R. C., editor (1975). *Conceptual Information Processing.* American Elsevier, New York.

Schank, R. C. (1982). *Dynamic Memory: A Theory of Learning in Computers and People.* Cambridge University Press, Cambridge, MA.

Sidner, C. (1983). Focusing in the comprehension of definite anaphora. In Brady, M. and Berwick, R., editors, *Computational Models of Discourse*, pages 267–330. MIT Press, Cambridge, MA.

Slade, S. (1987). *The T Programming Language: A Dialect of Lisp.* Prentice-Hall, Inc, Englewood Cliffs, NJ.

Steele Jr., G. L. (1984). *Common Lisp: The Language.* Digital Press, Bedford, MA.

Turner, S. R. and Reeves, J. F. (1987). Rhapsody user's guide. Technical Report UCLA-AI-87-3, UCLA Artificial Intelligence Laboratory, University of California, Los Angeles.

Wilensky, R. (1981). A knowledge-based approach to natural language understanding: A progress report. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence (IJCAI-80)*, Vancouver, British Columbia.

Wilensky, R. and Arens, Y. (1980). Phran: A knowledge-based natural language understander. In *Proceedings of the 18th annual meeting of the Association for Computational Linguistics (ACL-80)*, Philadelphia, PA.

Winograd, T. (1983). *Language as a Cognitive Process, Volume 1: Syntax.* Addison-Wesley, Reading, MA.

Woods, W. T. (1970). Transition network grammars for natural language analysis. *Communications of the ACM*, 13(10):591–606.

Zernik, U. (1987). *Strategies in Language Acquistion: Learning Phrases from Language in Context.* PhD thesis, UCLA Artificial Intelligence Laboratory, University of California, Los Angeles. Technical report UCLA-AI-87-1.

From: Stephanie Sandberg <Stephanie.Sandberg@efi.com>
To: "'bramirez@cs.ucla.edu'" <bramirez@cs.ucla.edu>
Subject: CS Technical Report
Date: Mon, 25 Oct 1999 10:52:43 -0700
X-Mailer: Internet Mail Service (5.5.2650.21)

Hi,

I'm a graduate student at San Francisco State, I'm looking for a
copy of CSD-890064 "The Rhapsody Phrasal Parser and Generator" by
John F. Reeves.

Is there any way to get a copy of this paper electronically?

Thanks,

Stephanie Sandberg