A Parser for HPSG

Alex Pranz

Jvdy 1990

Report No. CMU-LCL-90-3

Laboratory for Computational Linguistics

139 Baker Hall
Department of Philosophy
Carnegie Mellon University
Pittsburgh, PA 15213

A Parser for HPSG

Alex Franz[1]

[1]I would like to thank Carl Pollard and Bob Carpenter for helpful comments and discussions. This work was supported by the NSF under grant number IRI-8806913.

Contents

1 Overview

This document describes a parser for Head-driven Phrase Structure Grammar (HPSG). The system was implemented in Lucid Common Lisp (Version 2.1.1) in the Laboratory for Computational Linguistics on an HP 9000/370 workstation.

Following [Carpenter and Pollard, 1989], I view the HPSG formalism as a set of constraints over a type scheme for sorted feature structures. The function of the HPSG interpreter is to solve these constraint equations.

The interpreter has three conceptual components: The grammar conversion routines transform an HPSG grammar into the appropriate internal format. The **grammar processor** performs parsing, and output routines print sorted feature structures either in ASCII format, or directly generate I^TgX instructions.

The next section provides a brief guide to running the parser. Subsequent sections describe HPSG from a grammar-writer's point of view, and provide formal input specifications that define the version of the HPSG formalism adopted here.

Following this, there are sections that describe grammar conversion and parsing in detail.

A sample grammar, a grammar conversion and parsing trace, some output from the program, and further details on printing feature structures are contained in the appendices.

2 Running the Parser

Following is a brief guide to running the parser that should provide enough information to allow the reader to start experimenting with the system.

2.1 Preparing a Grammar

First of all, an input grammar must be provided. Appendix A contains a sample grammar that could be modified for this purpose. Sections 4 and 5 describe HPSG grammars in more detail, and should be referred to during more serious grammar-writing efforts.

2.2 Loading the Parser

The parser has been compiled for Lucid CL versions 2.1.1 and 3.0, but the Lisp source should load in most Common Lisp implementations. When compiling the program on other machines care must be taken to first load the Lisp source files in order to properly compile macros.

The source code has been divided into the following files:

[e **startup.lisp (global declarations)**
aux.lisp (auxiliary functions)
graph-rep.lisp (functions for representation and manipulation of graphs)
parse.lisp (parser functions)
convert-input-gramxnar.lisp (functions to read and convert input grammars)
latex-output.lisp (functions to generate &TEX instructions for printing graphs)
debug.lisp (auxiliary grammar and parser debugging functions)

Typing (load "startup") at the Lisp prompt causes all parser files to be loaded:

```
> (load "startup")
;;; Loading binary file "startup.hbin"
;;; Loading binary file "aux.hbin"
;;; Loading binary file "graph-rep.hbin"
;;; Loading binary file "parse.hbin"
;;;; Loading binary file "convert-input-grammar.hbin"
;;; ; Loading binary file "latex-output.hbin"
;;;; Loading binary file "debug.hbin"
#P"7users/alex/Courses/Nlp3/Code/startup.hbin"
>
```

If the parser files have not been compiled for the machine in use, *first* load the source files, then type

(compile-parser)

to cause all the files to be compiled. Finally, load the binaries by typing (load "startup") again.

2.3 Reading an Input Grammar

Type (run) to start the parser. The greeting will be printed, and you will be asked whether you wish to keep a trace in the log file; typing <retura> causes this to default to "no*. The trace, which is written to a file called parser.log, contains a lot of information (see Section 6.2) and keeping a trace will slow parsing down considerably. However, it is essential for grammar-debugging, and examining selected parts of the trace will provide insights on the working of the parser.

```
> (run)
                    HPSG Parser
??? Do you want to keep a trace in the log file ?: ["No"] yes
**• Trace will be written to file parser.log
```

Next, you will be prompted to type the name of the file containing the input grammar. Type <retura> to read the default file, grammarl.text:

```
??? Enter file name of input grammar: ["grammarl.text[11]]
*** Initializing Converter Variables
*** Converting grammar in file "grammarl.text[11]
*** Parsing signature specification
*** There are 131 sorts
**• Reading sort and relation definitions
*•• Performing inheritance in sort hierarchy
*** Constructing minimal satisfiers
*•* Constructing datatypes
*** Constructing satisfiers of definite constraints
*** Constructing satisfiers of conjuncts in definitions
*•* Constructing satisfiers of disjunctive constraints
*** Expanding minimal satisfiers
*•* Grammar conversion was successful.
**• Initializing parser
```

The parser now reads the input grammar and converts it into a number of internal data structures. Messages indicating itsj>rogress are printed throughout, as shown above.

2.4 Parsing Sentences

After a little while you will reach the parsing component of the system, indicated by the following message and prompt:

```
<parser> Enter type of expression and words terminated by ';>:
 -->
```

Input to the parser consists of two parts. The first indicates the type of expression you wish to parse; possible expressions include sentence, phrase, sign. This is followed by the words that constitute the expression. Commands to the parser must be ended by a semicolon.

sentence may be abbreviated as s. In addition, if the keyword :all is included the parser will retTirn all possible parses; otherwise, it will deliver only the first parse found. Below are some examples of possible input to the parser if the grammar in **grammar1.text** has been loaded:

```
—> sentence sandy laughs;
—> :all s sandy gives cindy tea;
—> :all phrase likes cindy;
```

Once an expression has been entered, the parser starts working on the input. It attempts to resolve sort and relation definitions in the grammar by searching through the space of non-disjunctive sorted feature structures created by disjunctions in the definitions. At every state, a message of the following format is printed out:

```
*** 0 states in queue: NIL
 Moving from graph #1
 Resolving disjuncts of — sentence —
 Mode: DEPTH-TIL-SPLIT Generated 3 new states:
  (#2 sentence #3 sentence #4 sentence)
```

The first line indicates how many unresolved states are currently waiting in the queue. Every time a new graph is created during search it is labelled with a number. The above message was printed at the initial state; thus, there are no states in the queue, and we are moving from the start state, corresponding to "graph #1*. The third line tells us that the parser is attempting to resolve the definition of the sort *sentence,* a special sort created to act as a template for parsing sentences.

Finally, the last line informs us that we are in search mode DEPTH-TIL-SPLIT and that we have generated three new states. It also tells us the numbers of the new graphs (#2, #3, and #4). The term [a]Depth-til-split" means breadth-first search that proceeds down paths until they split. That is, at any point where only one new state is generated we continue search from the new state, but whenever two or more new states are generated the new states are adjoined to the back of the queue, and search proceeds in the usual breadth-first manner.

If all moves from some non-final state fail, a message like the following is printed:

```
        ⋮
** 2 states in queue:  (#3 #4)
Moving from graph #2
Resolving disjuncts of — vord —
All moves failed (backtracking)...
*• 1 states in queue:(#4)
Moving from graph #3
⋮
```

Graph #2, while not bdng final, turned out to be a dead end since a "word" constraint could not be satisfied. Thus, the graph was deleted, and search continued from the next state in the queue (#3).

If you axe running in all-paths-paxsing mode (having included the :all keyword), the parser prints this message every time it has found a parse:

```
*•* Found one parse — looking for more...
```

The parser may be interrupted at any time; the feature structures corresponding to the parses found so fax will be accessible in a list as the value of the global variable *all-parses*.

Eventually, one of the following messages will be printed out:

```
*** Exiting; first parse found is in *parse*
*** Exiting; there was no successful parse
*** There are no more parses; all parses found are in *all-parses*
```

This will be followed by the parser's listener prompt

```
<parser> Enter type of expression and words terminated by ';':
 -->
```

The parser has finished, and you are ready to inspect the result.

2.5 Displaying Output

The parser listener is a simple loop that accepts expressions to be parsed (as shown above), the command quit, or a Lisp form to be evaluated. The latter is achieved by typing the following:

```
—> eval <Lispform>;
```

The function (print-graph <graph>) can be used to print a feature structure in ASCII format, as shown on the next page.

```
-> eval (print-graph "parse");
- Graph number "@2#" -

   PHOK <»«&st-pao»-word>
-ir-!:  FIRST |I]<MBdy>
    ?    BEST W<arfUt-phon-word>
         FIRST [6] <laughs>
         REST <clist-phon-word>
SYKSEM l»] <synsem>
        LOCOoc»l>
            CAT <c*t>
                MARKING <«nm»rked>
                BEAD [5] <**T*>>
                        VPOBlf  <fin>
                        AUX <minw>
                        IKV <nlın«>
                        PRD <mi»tt*>
                        SPEC <»3rn»exn>
              C"  SUBCAT <elbt.«yB»cm>
              CONT [3] <U«fB-M>*>
            xEf. . ". LAUGHER |2] <apro>
                        IKDEX <iadex>
                            GEKD <feadcr>
                            KUM <«ng>
                            PER <third>
DTRS <he*d.coxnp-«truc>
    HEAD-DTR <phr»*«>
            PHOK 17]
            SYKSEM <syn*«m>
                LOC Ooc*l>
                    CAT <c»t>
                        MARKING <unm*rked>
                        SUBCAT <ueli»t.«yn»en.>
                            FIRST [4] <iyn»ena>
                                LOC <loc»]>
                                    CAT <c*t>
                                        MARKIKG <«nm»rked>
                                        SUBCAT <«li«t-tyn*em>
                                        HEAD <BO«B>
                                            SPEC I«|
                                            PRD <boolean >
                                            CASE <nom>
                                    COKT <rei-obj>
                                        RESTR <n»mMO»>
                                            KAME II]
                                            NAMED 12]
                                        PARA 12]
                        REST <eli*t-»yn*em>
                    BEAD \b)
                CONT f3]
            DTRS <he*d-comp.«tmc>
                HEAD-DTR <U«ffl»-w©rd>
                    PBON <BdUt.pboB.word>
                        FIRST l«]
                        REST <clUt.pboB.word>
                    SYNSEM <tyB*em>
                        LOC <loc*l>
                            CAT <c*t>
                                MARKING <unm*rked>
                                SUBCAT <neltwt.«yn»eni>
                                    FIRST [4]
                                    REST <eli*t.«yn#«m>
                                BEAD [5]
                            CONT 13]
                COMP-DTRS <cfi*t.MfB>
  - COMP-DTRS <ndi«u«fB>
            FIRST <»*Bdy-word>
                PBON <neli»t.pbon-word>
                    FIRST [I]
                    REST <clwt-phoB-word>
                SYNSEM [4]
            REST <eli»t-»if»>
<parser> Eater typ« of cxprcMitoB »ad word* terminated by ';':
->
```

2.6 Using UTjpC to Display Output

Since HPSG's feature structures quickly grow beyond a manageable size, output functions have been provided that generate MfcX instructions directly from graphs. There axe three macros for

generating LaTeX instructions to typeset a feature structure:

(la-s <graph>) to typeset a feature structure in a miniscule typeface

(la-m <graph>) to typeset a feature structure in a minute typeface

(la-l <graph>) to typeset a feature structure in a small typeface

Let's assume that a sentence has just been parsed, and that the command quit has been issued to the parser, placing us back at Lisp top level with the global variable *parse* holding the result of parsing.

```
*** Exiting; first parse found is in *parse*
<parser> Enter type of expression and words terminated by ';':
 --> quit;
NIL
> (la-s *parse*)
??? Enter file name for LaTeX output: ["graphtex.txt"]
```

You are prompted for the name of the file to which the LaTeX instructions will be written. Unless you want to prevent it from being overwritten the next time LaTeX instructions are generated, just hit return to direct output to the default file:

```
*** Setting LaTeX parameters for style TABULAR in size SMALL
*** Writing LaTeX source to file "graphtex.txt"
*** Done writing LaTeX output
NIL
>
```

Next, type (latex) to start LaTeX on the newly written output, and to produce a dvi-file:[1]

[1]This function is operating system dependent, and might not work in a different implementation of Lisp.

```
> (latex)
*** LaTeXing root file crammedoutput.tex
This is TeX, Version 2.93 (eas «(#) 1.61 88/10/11 17:57:06)
(preloaded format=lplain 88.10.11)
(crammedoutput.tex
LaTeX Version 2.09 <4 Aug 1988>
(/usr/lib/tex/macros/article.sty
Document Style 'article[1] <16 Mar 88>,
(/usr/lib/tex/macros/artll.sty))  (/usr/lib/tex/macros/fleqn.sty)
(/usr/lib/tex/macros/leqno.sty) (crammedoutput.aux) (graphtex.txt) [1]
(crammedoutput.aux)
Output written on crammedoutput.dvi (1 page, 5008 bytes).
Transcript written on crammedoutput.log.
**• Starting xdvi on file crammedoutput.dvi
??? Do you want to print the output ?: ["Ho"]
HIL
>
```

The function runs &T£X on the instructions generated earlier, starts up xdvi (an X-Windows based DVI previewer) on the output, and asks whether to print the dvi-file.[2] Figure 1 shows the final product of this procedure.

2.7 Changing Search Strategy and other Parameters

Table 1 summarizes general system parameters. (See appendix D for StTjjX-outpnt related parameters.)

Name	Default Value	Other Possible Values
•search*	DEPTH-TIL-SPLIT	BREADTH DEPTH
max-no-of -sorts	140	integer < 250
max-sentence-length	20	integer < 40
•circular-features*	("SPEC")	list of strings

Table 1: Some System Parameters

The value of *search* controls the search mode of the parser. The two numeric variables control the size of certain internal data structures that axe set up during grammar conversion and parsing; setting them to values much larger than necessary will cause Lisp to set up unneccessarily large arrays. It is important to add the names of features that introduce cycles in Feature Structures to the value of *circular-features*; otherwise, the parser might get stuck in a loop when trying to expand feature structures (see Section 7.8).

[2]Again, the latter two operations can not be expected to work on machines outside the LCL without changes to the Lisp code in the file latex-output.lisp.

Output from HPSG parser

July 10,1990

GraplT Mraiber #24

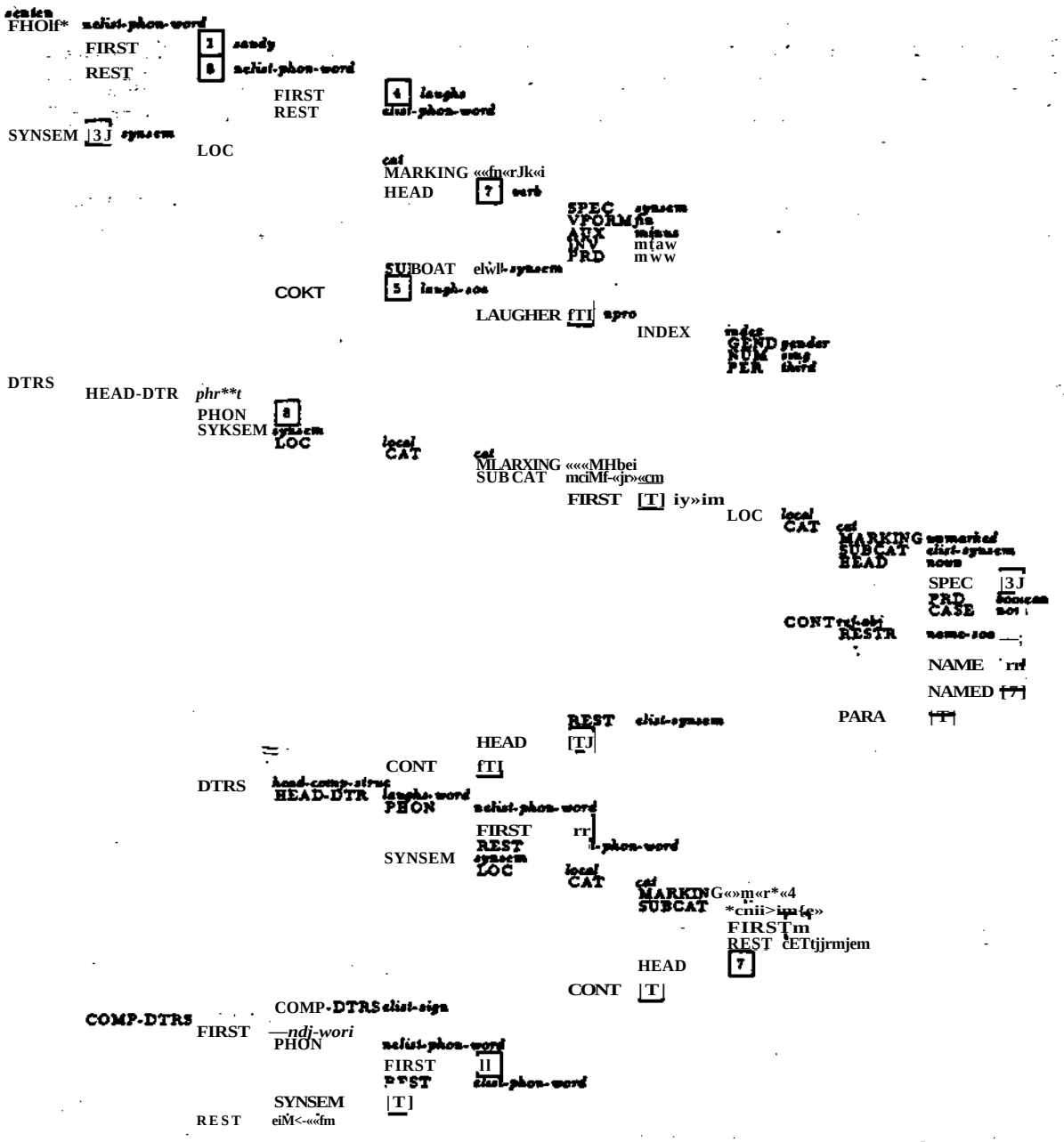

Figure 1: ^TjjX output generated by parser for "Sandy laughs'"

3 About HPSG

This section constitutes a brief review of some relevant aspects of the HPSG formalism. It is not intended to provide exhaustive coverage; for this, the reader is advised to turn to [Pollard, forthcoming, Pollard and Moshier, 1990].

HPSG uses *sorted feature structures* to model the information contained in linguistic objects. Natural Language grammars are written as constraints on feature structures, and "the linguistic entities which correspond to admissible feature structures constitute the predictions of the theory" ([Pollard and Sag, forthcoming]).

Sorted feature structures are a specific kind of graph structure that can be identified in a number of ways. In this report, as in much of the HPSG literature, I use diagrams of Attribute-Value Matrices (AVMs), and a logical feature structure description language is defined below to represent input to the parser. Inside the parser sorted feature structures are represented in yet another way, which is described in Section 7.5.

3.1 Organization of HPSG Grammars

I have divided HPSG grammars into a number of components:

Signature Four kinds of information are specified in the signature:

- The available sorts, features, and relations
- The subsumption ordering on the sorts.
- Which features are carried by the individual sorts ("feature appropriateness").
- Type restrictions on the values of individual features ("sortal restrictions") and on the arguments of relations

Lexicon The lexicon is the repository of all information carried by lexical sorts.

Principles of Grammar Universally valid well-formedness conditions such as the Head Feature Principle and the Subcategorization principle.

Constituent Structure Principles Well-formedness conditions on phrases related to constituent structure. This includes the Immediate Dominance Principle (IDP), and the Constituent Order Principle (COP).

The IDP and COP are not included with the other principles of grammar because they represent information about phrase structure. One possibility to avoid the combinatorial explosion of disjunctions would be to treat these constraints differently than other disjunctive constraints in the grammar; however, such a scheme has not been implemented here.

How does the above correspond to a system of constraints on models of linguistic objects? IVom a technical point of view, grammars written in the HPSG formalism consist of two components:

- The signature defines a system of types for feature structures, where each type of feature structure corresponds to a type of linguistic object that has been identified in the empirical domain.
- A system of constraints over this type scheme, including relational constraints, represents the constraints on admissible models.

Given a signature that specifies a type scheme of sorted feature structures, the principles of grammar make up a set of recursive constraint equations over the type scheme. For each sort in the sort hierarchy, the equations specify the constraints that have to be met by well-formed instances of the sort. These constraints are called the sort's *definition.*

3.2 The Signature

As described above, the signature defines a type scheme for sorted feature structures. Formally, it has the following parts:

- SORTS is the set of sorts.
- FEATURES is the set of features.
- $\sqsubseteq$ is an inheritance (or "subsumption") ordering over SORTS
- Appropriate: SORTS $\rightarrow$ (FEATURES -> SORTS) is a function that defines which features axe appropriate for individual sorts, and type restrictions on the values of those features.

Sorts are similar to record datatypes. Here is an example of an entry that might be found in a sort hierarchy:

sign
PHON: *list(phon-word)*