Aarno Lehtola
Sitra Foundation
P.O.Box 329, SF-00121 Helsinki

DPL - A COMPUTATIONAL METHOD FOR DESCRIBING GRAMMARS AND MODELLING PARSERS

ABSTRACT DPL, Dependency Parser Language, is a special definition language for describing natural language grammars. It is based on functional dependency. A DPL-grammar consists of (1) definition of used metrics ie. property names and values (2) definition of binary dependency relations and grammatical functions between constituent-pairs (ie. words or recognized phrase constructs) and (3) description of constituent surroundings in the form of two-way automata. The compilation of DPL-grammars results in executable codes of corresponding parsers.

To ease the modelling of grammars there exists a linguistically oriented programming environment, which contains e.g. tracing facility for the parsing process, grammar-sensitive lexical maintenance programs, and routines for the interactive graphic display of parse trees and grammar definitions. Translator routines are also available for the transport of compiled code between various LISP-dialects. The DPL-compiler and associated tools can be used under INTERLISP and FRANZLISP. This paper focuses on knowledge engineering issues. Linguistic argumentation we have presented in /3/ and /4/. The detailed syntax of DPL with examples can be found in /2/.

I INTRODUCTION

Our initial objective was to build a parser for Finnish to work in real production applications. We were faced with both linguistic and computational problems: (1) so far there was no formal description of the Finnish grammar, (2) there was no parser formalism that seemed specially suitable for highly inflectional and relatively free word order languages, (3) our computational solutions should be portable, efficient and general. In addition we wanted to have linguistic knowledge and processing mechanisms separated in our system. It is important that linguists may ignore the computational details of the parsing processes while the computer professionals may purely concentrate on computational issues.

The parser system we have developped is based on functional dependency. Grammar is specified by a family of two-way finite automata and by dependency function and relation definitions. These are expressed via DPL and compiled automatically to executable parsers. The flexible programming environment makes it easy to tune up parsers. The architecture of DPL-environment is described schematically in Figure 1. The main parts are highlighted by heavy lines. Single arrows represent data transfer; double arrows indicae the production of data structures. The realisations do not rely on specifics of underlying LISP-environments.

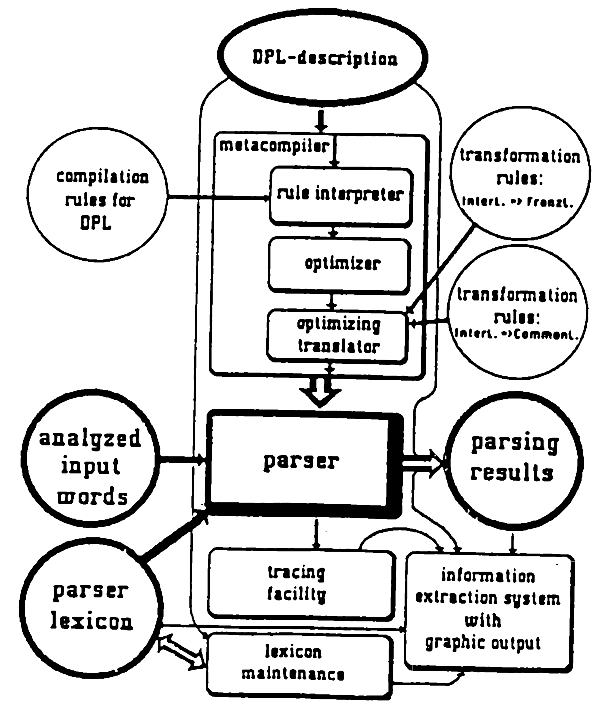


Figure 1. The architecture of the DPL-environment

II DPL-DESCRIPTIONS

The main data object in DPL is a constituent. A grammar specification opens with the structural descriptions of constituents and the allowed property names and property values. User may specify simple properties, features or categories. The structures of the lexical entries are also defined at the bėginning. All properties of constituents may be referred in a uniform manner using their values straight. The system automatically takes into account the computational details associated to property types. For example, the system is automatically tuned to notice the inheritance of properties in their hierarchies. Extensive support to multidimensional analysis has been one of the central objectives in the design of the DPL-formalism. Patterning can be done in multiple dimensions and the property set associated to constituents can easily be extended.

The binary grammatical functions and relations are defined as special and-or-expressions which contain both property predications and search directing information. A DPL-function returns as its value the binary construct built from the so called current constituent and its dependent candidate, or it returns NIL. DPL-relations return as their values the pairs of constituents that have passed the predicate filter. A user may vary a predication between simple equality and equality with ambiguity elimination. As their side effects predications may also replace and insert properties.

In the level of two-way automata the working storage consists of two constituent stacks and of a register which holds the current constituent. The two stacks hold the right and left contexts of the current constituent. The basic decision for the automaton associated with the current constituent is to accept or reject a neighbor via a valid syntactico-semantic subordinate relation. Successfully called DPL-function subordinates the neighbor, and it disappears from the stack. The structure of an input sentence will be the outcome of a series of such binary constructions. Dynamic local control is realized by permitting the automata activate one another.

III THE DPL-COMPILER

A compilation results in executable code of a parser. The compiler produces highly optimized lisp code /1/. In the generated code only a small set of basic lisp functions is used. In bench marking was found that specialized higher level functions often consume more time than corresponding functions composed of open compilable basic functions. For instance many type checks can be avoided when the actual situation of use is known. In addition the chosen set makes it easier to transfer parsers to other computers. The low-level lisp code may be compiled to machine language level by normal lisp compilers.

Internally data structures are only partly dynamic for the reason of fast information fetch. Ambiguities are expressed locally to minimize redundant search. The principle of structure sharing is followed whenever new data structures are built. In the manipulation of constituent structures there exists a special service routine for each combination of property and predication types. These routines take special care of time and memory consumption. For instance with regard to replacements and insertions the copying includes physically only the path from the root of the list structure to the changed sublist. The logically shared parts will be shared also physically. This principle of structure sharing minimizes memory usage.

In the state transition network level the search is done depth first. To handle ambiguities DPL-functions and -relations process all alternative interpretations in parallel. In fact the alternatives are stored in the stacks and in the current constituent register as trees of alternants.

As a general time consuming strategy iteration is preferred to recursion whenever possible. Boolean expressions are optimized to avoid unnecessary nesting. The same affects also nested 'conds' and 'ifs'. Local memory reservation is minimized by taking into account control paths.

In the first version of the DPL-compiler the generation rules were intermixed with the compiler code. The maintenance of the compiler grew harder when we experimented with new computational characteristics. We therefore developed a metacompiler in which compilation is defined by rules.

Our parsers were aimed to be practical tools in real production applications. It was hence important to make the produced programs transferable. As of now we have a rule-based translator which converts parsers between LISP-dialects. The translator accepts currently INTERLISP, FRANZLISP and COMMON LISP.

IV LEXICON AND ITS MAINTENANCE

The environment has a special maintenance program for lexicons. The program uses video graphichs to ease updating and it performs various checks to guarantee the consistency of the lexical entries. It also co-operates with the information extraction facility to help the user in the selection of properties.

V THE TRACING FACILITY

The tracing facility is a convenient tool for grammar debugging. For example, in Figure 2 appears the trace of the parsing of the sentence "Poikani tuli illalla kentältä heittämästä kiekkoa." (i.e. "My son came back in the evening from the stadium where he had been throwing the discus."). Each row represents a state of the parser before the control enters the state mentioned on the right-hand column. The thus-far found constituents are shown by the parenthesis. An arrow head points from a dependent candidate (one which is subjected to dependency tests) towards the current constituent.

```
_(T POIKANI TULI ILLALLA KENTÄLTÄ HEITTÄMÄSTÄ KIEKKOA .)

383 conses
.03 seconds
0.0 seconds, garbage collection time
PARSED
_PATH()

=>  (POIKA)  (TULLA)  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?N
(POIKA)  <=  (TULLA)  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  N?
=>  (POIKA)  (TULLA)  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?NFinal
(##)  (POIKA)  (TULLA)  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  NIL
(POIKA)  =>  (TULLA)  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?V
=>  ((POIKA) TULLA)  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?VS
((POIKA) TULLA)  <=  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  VS?
((POIKA) TULLA)  =>  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?N
((POIKA) TULLA)  (ILTA)  <=  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  N?
((POIKA) TULLA)  =>  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?NFinal
((POIKA) TULLA)  <=  (ILTA)  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  VS?
((POIKA) TULLA (ILTA))  <=  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  VS?
((POIKA) TULLA (ILTA))  =>  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?N
((POIKA) TULLA (ILTA))  (KENTTÄ)  <=  (HEITTÄÄ)  (KIEKKO)  N?
((POIKA) TULLA (ILTA))  =>  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  ?NFinal
((POIKA) TULLA (ILTA))  <=  (KENTTÄ)  (HEITTÄÄ)  (KIEKKO)  VS?
((POIKA) TULLA (ILTA) (KENTTÄ))  <=  (HEITTÄÄ)  (KIEKKO)  VS?
((POIKA) TULLA (ILTA) (KENTTÄ))  =>  (HEITTÄÄ)  (KIEKKO)  ?V
((POIKA) TULLA (ILTA) (KENTTÄ))  (HEITTÄÄ)  <=  (KIEKKO)  V?
((POIKA) TULLA (ILTA) (KENTTÄ))  (HEITTÄÄ)  =>  (KIEKKO)  ?N
((POIKA) TULLA (ILTA) (KENTTÄ))  (HEITTÄÄ)  (KIEKKO)  <=  N?
((POIKA) TULLA (ILTA) (KENTTÄ))  (HEITTÄÄ)  =>  (KIEKKO)  ?NFinal
((POIKA) TULLA (ILTA) (KENTTÄ))  (HEITTÄÄ)  <=  (KIEKKO)  V?
((POIKA) TULLA (ILTA) (KENTTÄ))  (HEITTÄÄ (KIEKKO))  <=  VO?
((POIKA) TULLA (ILTA) (KENTTÄ))  =>  (HEITTÄÄ (KIEKKO))  ?VFinal
((POIKA) TULLA (ILTA) (KENTTÄ))  <=  (HEITTÄÄ (KIEKKO))  VS?
((POIKA) TULLA (ILTA) (KENTTÄ) (HEITTÄÄ (KIEKKO)))  <=  VS?
=>  ((POIKA) TULLA (ILTA) (KENTTÄ) (HEITTÄÄ (KIEKKO)))  ?VFinal
((POIKA) TULLA (ILTA) (KENTTÄ) (HEITTÄÄ (KIEKKO)))  <=  MainSent?
((POIKA) TULLA (ILTA) (KENTTÄ) (HEITTÄÄ (KIEKKO)))  <=  MainSent? OK
DONE
```

Figure 2. A trace of parsing process

The tracing facility gives also the consumed CPU-time and two quality indicators: search efficiency and connection efficiency. Search efficiency is 100%, if no useless state transitions took place in the search. This figure is meaningless when the system is parameterized to full search because then all transitions are tried. Connection efficiency is the ratio of the number connections remaining in a result to the total number of connections attempted for it during the search.

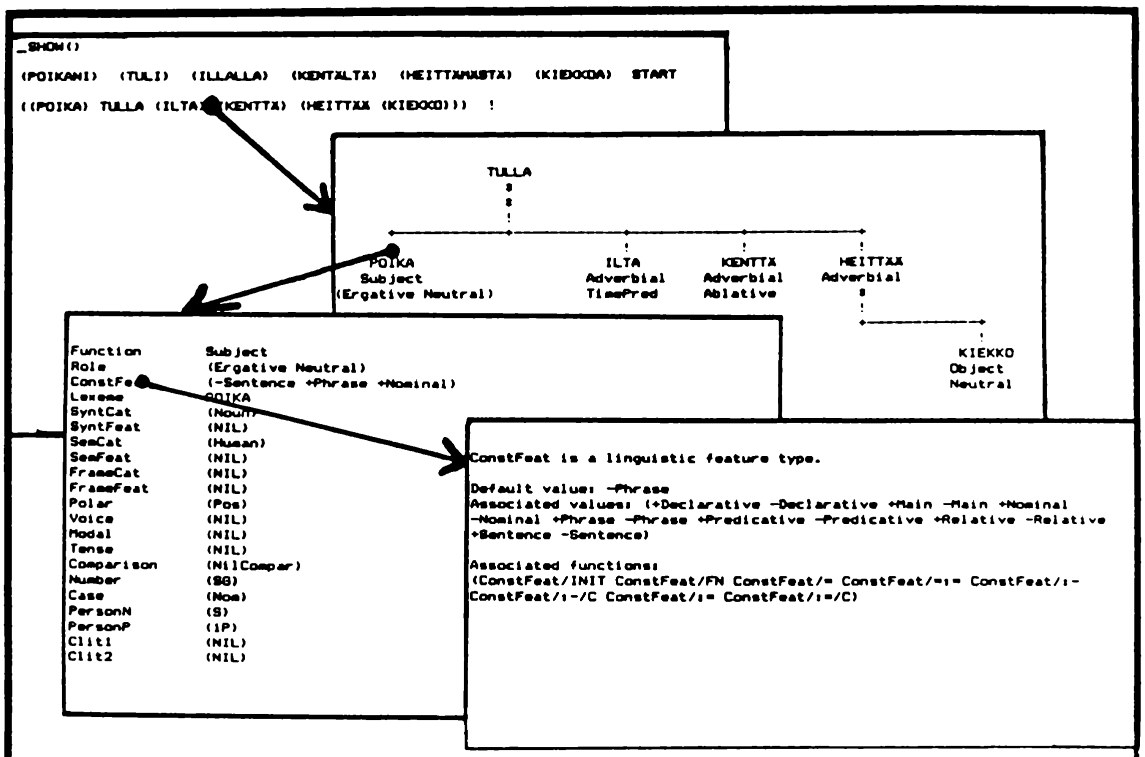
There exists also automatic book-keeping of all sentences input to the system. These are divided to into two groups: parsed and not parsed. The first group constitutes growing test material to ensure monotonic improvement of grammars.

VI <u>THE INFORMATION EXTRACTION FACILITY</u>

In an actual working situation there may be thousands of linguistic symbols in the work space. To make such a complex manageable, we have implemented an information system that for a

-156-

given symbol pretty-prints all information associated with it.



```
_SHOW()
(POIKANI)  (TULI)  (ILLALLA)  (KENTÄLTÄ)  (HEITTÄMÄSTÄ)  (KIEKKOA)  START
((POIKA) TULLA (ILTA) (KENTTÄ) (HEITTÄÄ (KIEKKO)))  !
```

```
              TULLA
                :
                :
                !
   +------------+------------+------------+------------+
                             !            !            !
   POIKA                    ILTA        KENTTÄ       HEITTÄÄ
  Subject                 Adverbial    Adverbial    Adverbial
(Ergative Neutral)        TimePred     Ablative         :
                                                        !
                                                        +-----------+
                                                                    !
                                                                  KIEKKO
                                                                  Object
                                                                  Neutral
```

```
Function       Subject
Role           (Ergative Neutral)
ConstFeat      (-Sentence +Phrase +Nominal)
Lexeme         POIKA
SyntCat        (Noun)
SyntFeat       (NIL)
SemCat         (Human)
SemFeat        (NIL)
FrameCat       (NIL)
FrameFeat      (NIL)
Polar          (Pos)
Voice          (NIL)
Modal          (NIL)
Tense          (NIL)
Comparison     (NilCompar)
Number         (SG)
Case           (Nom)
PersonN        (S)
PersonP        (1P)
Clit1          (NIL)
Clit2          (NIL)
```

```
ConstFeat is a linguistic feature type.

Default value: -Phrase
Associated values: (+Declarative -Declarative +Main -Main +Nominal
-Nominal +Phrase -Phrase +Predicative -Predicative +Relative -Relative
+Sentence -Sentence)

Associated functions:
(ConstFeat/INIT ConstFeat/FN ConstFeat/= ConstFeat/=:= ConstFeat/:-
ConstFeat/:-/C ConstFeat/:= ConstFeat/:=/C)
```

Figure 3. An example of information extraction utilities

The environment has routines for the graphic display of parsing results. A user can select information by pointing with the cursor. The example in Figure 3 demonstrates the use of this facility. The command SHOW() inquires the results of the parsing process described in Figure 2. The system replies by first printing the start state and then the found result(s) in compressed form. The cursor has been moved on top of this parse and CTRL-G has been typed. The system now draws the picture of the tree structure. Subsequently one of the nodes has been opened. The properties of the node POIKA appear pretty-printed. The user has furthermore asked information about the property type ConstFeat.

VII CONCLUSION

The parsing strategy applied for the DPL-formalism was originally viewed as a cognitive model . It has proved to result practical and efficient parsers as well. Experiments with a non-trivial set of Finnish sentence structures have been performed both on DEC-2060 and on VAX-11/780 systems. Experiments with a non-trivial set of Finnish sentence structures have been performed both on DEC-2060 and on VAX-11/780 systems. The time behaviour on DEC-2060 has been described in Figure 4. In those test runs only main sentences were used. The analysis of an eight word sentence, for instance, takes between 20 to 600 ms of DEC CPU-time in the INTERLISP-version depending on whether one wants only the first or, through complete search, all parses for structurally ambiguous sentences. The MACLISP-version of the parser runs about 20% faster on the same computer. The NIL-version (COMMON LISP compatible) is about 5 times slower on VAX.

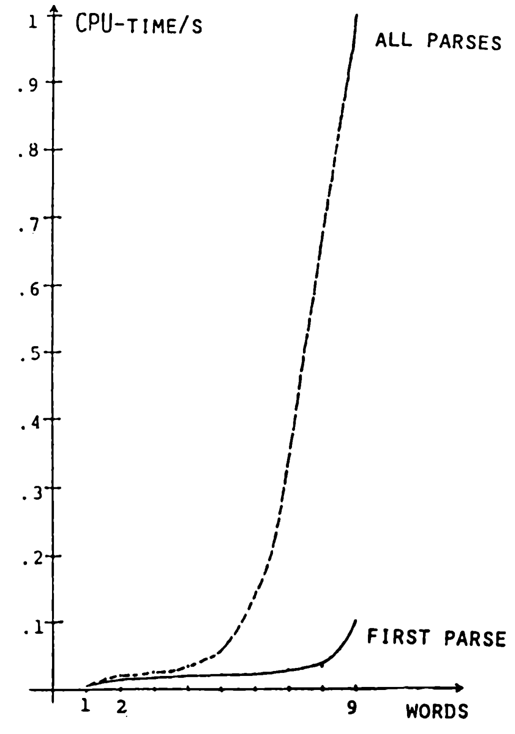


Figure 4. Efficiency test on DEC-2060 under INTERLISP

The whole environment has been transferred also to FRANZLISP on VAX. We have not yet focused on optimality issues in grammar descriptions. We believe that by reordering expectations in the automata and by introducing more heuristics to reduce parallelism improvement in efficiency ensues.

REFERENCES

1. Lehtola, A., Compilation and Implementation of 2-way Tree Automata for the Parsing of Finnish. M.S. Thesis, Helsinki University of Technology, Department of Technical Physics, 1984, 120 p. (in Finnish)

2. Lehtola, A., Jäppinen, H. and Nelimarkka, E., Language-based Environment for Natural Language Parsing. Proc. of the 2nd Conf. of the European Chapter of the Association for Computational Linguistics, Geneva, 1985, pp. 98-106.

3. Nelimarkka, E., Jäppinen, H. and Lehtola, A., Two-way Finite Automata and Dependency Theory: A Parsing Method for Inflectional Free Word Order Languages. Proc. COLING84/ACL, Stanford, 1984, pp. 389-392.

4. Nelimarkka, E., Jäppinen, H. and Lehtola, A., Parsing an Inflectional Free Word Order Language with Two-way Finite Automata. Proc. of the 6th European Conference on Artificial Intelligence, Pisa, 1984, pp. 167-176.

5. Winograd, T., Language as a Cognitive Process. Volume I: Syntax. Addison-Wesley Publishing Company, Reading, 1983, 640 p.