Efficiency Considerations for LFG-Parsers – Incremental and Table-Lookup Techniques

István Bátori and Stefan Marok
FB 2 Linguistics EWH Rheinland-Pfalz, Rheinau 3-4,
D-5400 Koblenz, FRG

Abstract

The paper presents a concise description of the LFG-Parser-Generator developed at the EWH in Koblenz. Special attention is paid to efficiency considerations to speed up the system in the execution phase. Lexicon is separated, ll(k)-parsing tables are used and some preliminary unifications are carried out before the actual execution. The run time system follows the single path strategy and produces the f-structures simultaneously with the processing of the c-structures.

1 General Considerations of Parsing Efficiency

Basic parsing techniques (both shift reduce and recursive descent) seem to be inherently inefficient inasmuch as they proceed strictly according to the sequence of the rules in the grammar and they are not able to exploit the surrounding (preceeding and following) syntactic information. Their scope is limited to a single rule and they jump mechanically to the sequentially next rule, even if such a move is obviously abortive and must be inmediately abandoned (Winograd 1983, 108-115; Phillips 1984; Hellwig 1988).

Parsing tables – as they are conceived in current compiler construction devices for LR(k) and LL(k) languages – make 1. the information provided by the grammar accessible throughout the entire processing and not just at the point where they happen to occur, and 2. they can be constructed algorithmically (Aho/Ullman 1979).

2 The LFG-Model of the EWH: General Design

The Koblenzer LFG-Parser-Generator is an interactive system, designed to create and to test grammars for natural languages according to the linguistic philosophy of the LFG as conceived in Bresnan und Kaplan (1982). Both lexicon and syntax follow closely the original format specifications. The system can be divided into two main phases: *preprocessing* and actual *execution*).

1. Preprocessing of the input grammar (including lexicon) generates the executable code, which in turn involves two logically distinct steps:
   - Generating the PROLOG code and
   - Optimizing the PROLOG code, – and
2. the actual execution phase analyses the input string and produces the f-structures.

2.1 Code-Generation

In the preprocessing phase the grammar rules are entered into the system and translated into an executable PROLOG Code. This part of the system is written in PASCAL. The implementation includes facilities for the treatment of the metavariables $\Uparrow$ and $\Downarrow$ needed for the treatment of the *long distance dependencies* (Weisweber 1986). The grammar may contain both optional categories and multiply reoccurring categories (marked by the Kleene-star $\star$-operator).

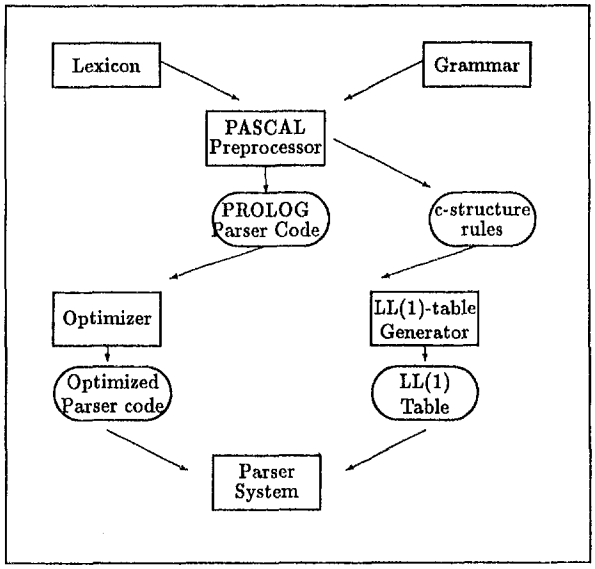

Figure 1: The LFG-System in Koblenz

In order to facilitate the generation of the tables with the *reach relations* the phrase-structure portion of the rules of the grammar (c-structure rules) is extracted and stored as an additional, separate data set.

2.2 Code Optimization

The second task of the preprocessor is to produce a more efficient PROLOG code. Optimization covers construction of parsing table and code revision.

In order to speed up the actual analysis in the execution phase the preprocessor constructs a *table of reach relations* on the basis of *first and follow sets*, connecting nonterminal and preterminal nodes with a lookahead of 1.

The definition of the *first and follow sets* is based on context free grammar (Aho/Ullman (1979, 186-192, 429-30)):

$$G = (N, \Sigma, \mathcal{P}, S).$$

$$\alpha, \beta, \in (N \in \Sigma)^* \text{ and } A \in N.$$

The *first sets* are defined for a non terminal symbol A over a string α of preterminals as the potential preterminal symbols which can occur in the leftmost position of the string:

$$\mathbf{FIRST}(\alpha) = \{a \in \Sigma \mid \alpha \stackrel{*}{\Longrightarrow} a\beta\}$$

$$\cup \{\in \mid \alpha \stackrel{*}{\Longrightarrow} \epsilon\}$$

The *follow sets* of a nonterminal A are defined as the *first sets* of the preterminals which may occur after the nonterminal A:

$$\mathbf{FOLLOW}(A) := \{a \in \Sigma \mid S \stackrel{*}{\Longrightarrow} \alpha A \beta \wedge a \in \mathbf{FIRST}(\beta)\}$$

$$\cup \{\$ \mid S \stackrel{*}{\Longrightarrow} \alpha A \$\}.$$

Contrary to the standard definition of the terms (op. cit.) the Koblenzer system does not exclude the application to left recursive constructions. The reach relations are build up uniformly both for left recursive and for all other constructions.

The *first and follow sets* allow to define the *reach relations*, which provide the information for a nonterminals A (in the stack) and for a preterminal symbol (located in the input string a) by which production rule(s) the preterminal can be accessed:

$$\mathbf{REACH}(A, a, P) \Longleftrightarrow \exists P \in \mathcal{P} \ \textit{with} \ P = A \rightarrow \alpha :$$

$$a \in \mathbf{FIRST}(\alpha) \wedge \neg(a = \epsilon)$$

$$\vee a \in \mathbf{FOLLOW(A)} \wedge a \stackrel{*}{\Longrightarrow} \epsilon$$

The *reach relations* are valid for all context free languages and extend the applicability of LL(1)-tables for them in general. They are calculeted over the *first and follow sets* and stored in tables for the execution phase. The practical construction of the table of reach relations is based on the systematic separation of dictionary and grammar rules, without which the construction of the table would not be feasible.

There are a number of grammatically predefined f-descriptions, which can be preprocessed in advance independently of the actual input, reducing the number of unifications at run time. Preliminary unification of f-structures can be carried out in the following configurations:

- If *an f-description subsumes another f-description*, the subsumed f-structure can be regarded as already unified and dropped. In the execution phase the system will use only the subsuming (i.e. larger) f-description. E.g. if a dictionary entry in the PROLOG code, produced in the preprocessing phase, has the specifications as (↑SUBJ NUM) = SG, and simultaneously: !(↑SUBJ), the later can be safely dropped in order to avoid the vacuous unification of the explicit subject in the execution phase.
- If *an f-description is unified with new attributes*, hitherto not used in the grammar, the operation will always succeed, regardless of the actual value of the attributes. Unifications of this type can be carried out safely in advance regardless of later possible changes of the attribute value.
- There are further minor possible f-structure configurations which can be simplified before the actual unification in the execution phase. The current optimization will recognize some of these special cases and replace the general unification procedures by specialized and hence more restricted procedures already at the time of code generation. The general broad unification procedures (merge functions) will be substituted here by more specific and computationally less expensive procedures.

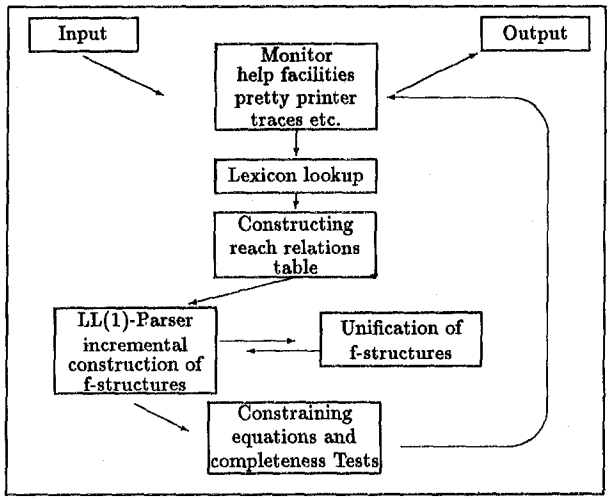

Figure 2: The run time system

3 The Run Time System

Firstly, the run time system can be characterized by the *basic separation of lexicon lookup and actual parsing*. The separation of lexicon rules and syntactic rules is based on the linguistic insight that the two components (lexicon and grammar) reflect entirely different language properties. The division can be supported also by consideration of processing efficiency.

The lexicon lookup is carried out at the beginning of the processing and it immediately allows the rejection of input in case of missing entries in the lexicon. The user can enter another word on the spot and proceed with the processing of the same sentence.

The next step is the inspection of the LL(1) tables by means of which the *reach relations are established*. The table of reach relations provides the optimal subset of grammatical symbols and connects them to the lexical entries occurring in the actual input sentence.

Secondly, the run time system is characterized by the *single-pass strategy* of processing, i.e. the input is read in only once, merging two fundamental tasks of the LFG: 1. the constructing of the c-structures and 2. the unification of the f-structures in a single step.

A special treatment is necessary for the left recursive constructions. The entries in the LL(1)-table for potential left recursions may be used only as long as the repetion is not spurious, otherwise their further application is suspended. At the time of the processing of phrase structure rules, the associated functional description is processed immediately. At this point the nodes relevant to the functional assignments are easily accessible as the left hand side symbol (for the metavariable ↑) and the right hand side symbols (for the metavariables ↓) in the rules.

As the input is processed the f-structure is constructed step by step *incrementally*. All available attributes and values are merged together as soon as they emerge, which is efficient for at least two reasons: 1. There is no need to store and reprocess the cumulated f-equations in an additional step and 2. merging the f-descriptions incrementally step by step operates with smaller chunks, which implies faster unification.

The incremental processing means that at the end of the input sentence the analysis is complete and solved and does not need to be scanned again in order to solve a series of f-equations. There is only one single control operation at the end of the sentence checking the wellformedness (completeness and exhaustiveness) of the output.

The single-pass model differs therefore from the Kaplan-Bresnan-model by lacking a separate processing phase for the cumulated f-structures following the generation of c-structures. In fact there is no explicit need for retaining the c-structures, except for their possible display in tutorials and in tracing errenous production, while testing the rules of the input grammar.

The current implementation delivers both the c-structure as well as the f-structure of the input sentence. In case of multiple interpretations all c-structures and all valid f-structures are displayed in succession.

4 Adequacy and Efficiency of Grammars

LFG-Grammars have been mostly studied from the point of view of linguistic adequacy, i.e. they have been developed in order to cover substantial aspects of natural language syntax phenomena. The parser should help the working linguist to find the optimal grammar for a particular language, to test the individual rules of the grammar as well as the general formalism.

Parsing efficiency can be studied at least at three different levels:

1. in view of the efficiency of developing grammars (concerns the work of the linguist).

2. in view of optimizing processing of the input grammar (concerns the preprocessing phase)

3. in view of optimizing the processing of the input sentence (concerns the execution phase and the run time system)

The present study is concerned mostly with the third type of efficiency i.e. with improving parsing efficiency, presupposing linguistic adequacy of the model. Practical efficiency of standard parsing techniques depends on the amount of back tracking and reprocessing needed in cases of errenous analysis. While using standard recursive descent parsing techniques guarantees the comprehensive coverage of the whole language, it does not exploit available information in an optimal manner. Parsing tables with REACH relations entail more information than single phrase structure rules, they accumulate information on the distribution of symbols in the whole grammar and enable the parser to build up the correct c-structure at the first attempt. If for some reason no valid structure is produced, ordinary back tracking applies and the input string is reprocessed.

In the final version of the paper further details of current improvements will be reported.

5 Literature used

Aho, Alfred V. and Ullman, Jeffrey D. (1979) Principles of Compiler Design. Reading, Massachusetts et al., Addison-Wesley Publishing Company.

Bresnan, Joan (1982) (ed.) The mental representation of grammatical relations. Cambridge, Massachusetts, The MIT Press.

Hellwig, Peter (in print) Parsing natürlicher Sprachen: Grundlagen und Realisierungen. In: Bátori, I., Lenders, W. and Putschke, W. (eds): Computational Linguistics – An international Handbook on Computer Oriented Language Research and Applications. Berlin, Walter de Gruyter

Kindermann, Jörg and Meier, Justus (1986) An extension of LR-Parsing for Lexical-Functional Grammar. Universität Bielefeld, Fakultät LiLi, Forschungsschwerpunkt Sprach- und Textverarbeitun. (To be published in: Reyle, U. (ed.): Word Order and Parsing in Unification Grammars).

Phillips, Brian (1984) An object-oriented parser. In: Bara, Bruno G. and Guida, Giovanni (eds.) Computational Models of Natural Language Processing. Amsterdam et al. North-Holland, 297-321.

Tomita, Masaru (1987) An Efficient Augmented-Context-Free Parsing Algorithm. CL 13:31-46.

Weisweber, Wilhelm (1986) Ein Parsergenerator für die lexical functional grammar (LFG). EWH Rheinland-Pfalz – Abteilung Koblenz – Fachberichte Informatik 4/86.

Winograd, Terry (1983) Language as a Cognitive Process – Syntax. Reading, Massachusetts et al., Addison-Wesley Publishing Company.