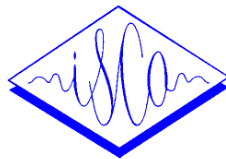
ISCA Archive
http://www.isca-speech.org/archive



6th International Conference on Spoken Language Processing (ICSLP 2000)
Beijing, China
October 16-20, 2000

Practical Spoken Language Translation Using Compiled Feature Structure Grammars

Lei Duan, Alexander Franz, Keiko Horiguchi

Spoken Language Technology, Sony US Research Laboratories
{lei, amf, keiko}@slt.sel.sony.com

10.21437/ICSLP.2000-230

Abstract

Practical work on spoken language translation must pursue two types of efficiency: computational efficiency, and "language engineering" efficiency. This paper describes the design, implementation, and evaluation of the GPL-based framework for spoken language translation that addresses both of these goals. In this framework, computational grammars are written in GPL, an easy-to-use imperative programming language that allows the direct expression of linguistic algorithms in terms of rewrite-grammars with feature structure tests and manipulations. Computational efficiency is achieved with the GPL compiler, which converts GPL grammars into efficient C routines, and with the GPL runtime environment, which provides services for linguistic representations, manipulation, and memory management. An evaluation of an English-Japanese spoken language translation system based on GPL shows that it is linguistically powerful, yet only requires reasonable computational resources.

1 INTRODUCTION

Our approach to practical spoken language translation is characterized by three main features. First, we defined the Grammar Programming Language (GPL), an imperative programming language for feature-structure-based rewrite-grammars that meets the goal of language engineering efficiency. Second, in order to achieve run-time efficiency, we implemented a compiler for GPL that converts GPL grammars into C routines, and we created a memory management scheme that allows very quick garbage collection. Third, by separating the GPL compiler from the software engines that provide the control structure for executing compiled GPL grammars, we created a modular framework that makes it easy to create new GPL-based NLP components.

1.1 Previous Work

Some ideas in our work can be traced back to the "pseudo-unification" approach described in [Tomita and Knight 1988], which in turn is related to LFG [Bresnan 1992] and to PATR-II [Shieber 1986]. In Tomita's system, the feature structure annotations to grammar rules undergo macro substitution, which results in Lisp functions that perform the run-time feature structure tests and manipulations.

Recent work on compiling feature structure-based grammars has focused on constraint-based grammar formalisms centred on HPSG. One example is the ConTroll system [Goetz and Meurers 1997]. In ConTroll, HPSG grammars are compiled into definite clause grammars, which are executed on a Prolog interpreter. In Carpenter and Penn's ALE system [Carpenter and Penn 1999], phrase structure grammars based on typed feature structures, as well as definite clause programs, are compiled into instructions for an abstract machine. The abstract machine instructions are then executed on a Prolog-based emulator compiled from the feature structure type specifications. In [Makino et al. 1998], a Prolog-like programming language for typed feature structures is compiled into instructions for an abstract machine, including the off-line generation of specialized unification routines for known typed feature structures. The abstract machine for typed feature structures (as well as an abstract machine for definite clause programs) is currently emulated by a C++-based emulator, but a native-code compiler is under development. [Wintner and Francez 1999] describe another approach involving an abstract machine tailored to ALE-style grammars, but excluding definite clauses as well as disjunction and negation in descriptions.

2 THE GPL COMPILER

GPL grammars are compiled into C language routines by the GPL compiler. This section describes the key features of this process.

2.1 Compiler Input and Output

GPL is an imperative programming language designed for feature-structure-based linguistic computation. The language includes various special data types, means to handle disjunction and conjunction, variables, simple and complex expressions, if-then-else and switch statements, loops, and many other features. For more details, please see [Franz et al. 2000]. The GPL compiler is a program that reads a grammar written in GPL and generates C language routines that carry out the tests and manipulations on the feature structures. The output of the GPL compiler is C source code that is compiled by a C compiler, and then linked with a software engine and other system modules.

2.2 Lexical Analysis, Syntactic Analysis and Semantic Actions

GPL is formally defined with a BNF grammar that covers the syntax of the language. Based on the BNF definition we

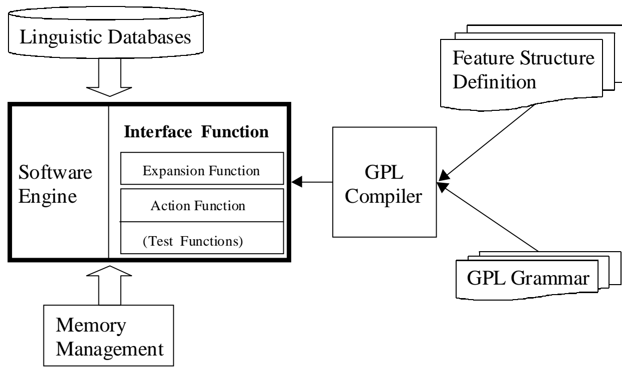


Figure 1 GPL Run-time Environment

created a `lex` grammar to recognize all tokens and keywords in GPL, and a `yacc` grammar to perform syntactic analysis of GPL grammars. The `yacc` rules include semantic actions that build up a successively larger context for the analyzed GPL code. This includes keeping track of global and local variables, maintaining a history of sub-feature-structure references for each rule, keeping track of separate test functions, mapping GPL constructions into corresponding calls to functions from a system library, and other steps. The final output from the compiler is a C language file that includes a section for variable declarations; a section for function prototypes; sections for pointer arrays for expansion and action functions; the definition of the interface function; and definitions of the expansion, action, and test functions.

2.3 Variables, Expressions, and Control Flow

GPL includes local variables and global variables, and both types may be declared anywhere in a GPL grammar. Variables are often used to store temporary results, but in certain contexts they actually function as pointers into other feature structures. The GPL compiler maintains a variable table that is used to keep track of variable names and properties, and to generate C variables and pointers in the compiler output.

Expressions in GPL are converted to C-style logical expressions. Complex GPL expressions involving the Boolean operators `AND` and `OR` are converted to C expressions involving `&&` and `||`. GPL statements for control flow, such as `switch`, `if-then-else`, and `loop`, are also converted into C-language constructs. For certain GPL constructions, the GPL compiler uses a type definition for features and their possible values to check correctness, and to determine which data type is involved. Currently, the GPL compiler does not use the feature structure type definition to generate fixed-width data structures in the style of [Carpenter and Penn 1999] or [Makino et al. 1998], but that is a possible future extension.

2.4 Compiling Nested Expressions

Some GPL constructions involve an expression that is nested inside a statement. For nested expressions of this sort, the compiler generates a separate “test” function, and includes a pointer to the test function in the appropriate place in the code for the action function. Other GPL statements allow similar constructions. This includes an operator to rename a slot if its value satisfies a test, an operator to initialise a local variable to point to a sub-feature-structure if it satisfies a test, and an operator to find a sub-feature-structure anywhere inside a feature structure if it satisfies a test.

2.5 Disjunctive Feature Structures

Disjunctive feature structures are used widely in GPL to represent natural language ambiguity. The semantics of the GPL predicates and manipulation operators can be extended naturally to cover disjunctive feature structures, and run-time efficiency is achieved with the following scheme.

During analysis of a GPL rule-body, the GPL compiler keeps track of all references to sub-feature-structures. Based on this information, the GPL compiler generates a so-called “expansion” function that, at runtime, expands any disjunction along any referenced path. When a GPL rule is applied, the expansion function is called first. In this way, if any of the feature structures contains a disjunction along a path that will be examined by the actions from the rule-body, then the disjunction is expanded once at the beginning of the rule application.

3 GPL-BASED COMPONENTS

We have implemented GPL-based components for parsing, transfer, and generation, and we have combined them into a bi-directional Machine Translation (MT) system between English and Japanese. This section describes some more details about GPL-based components, their run-time environment, and the architecture for the MT system.

3.1 Creating a GPL-based Component

The GPL compiler output follows certain object-oriented conventions that make it easier to create GPL-based NLP components for different tasks (such as parsing and generation) as well as for different languages (such as English analysis and Japanese analysis). For every rule in a GPL grammar, the GPL compiler generates an action function, as well as an expansion function, and any necessary test functions. The GPL compiler generates a public “interface function” that serves as the link between the software engine and the compiled GPL grammar. The software engine provides the global data structures and the control structure that is necessary for executing a GPL grammar in the intended manner. For example, in a GPL-based chart parser, the software engine would provide the data structure for the chart, as well as the control structure for deciding which rules should be applied at what point, and with what input. At every rule application, the software engine would marshal the input feature structures from the chart, it would call the appropriate GPL routines via the interface function, and it would take the appropriate actions if the GPL rule failed or succeeded.

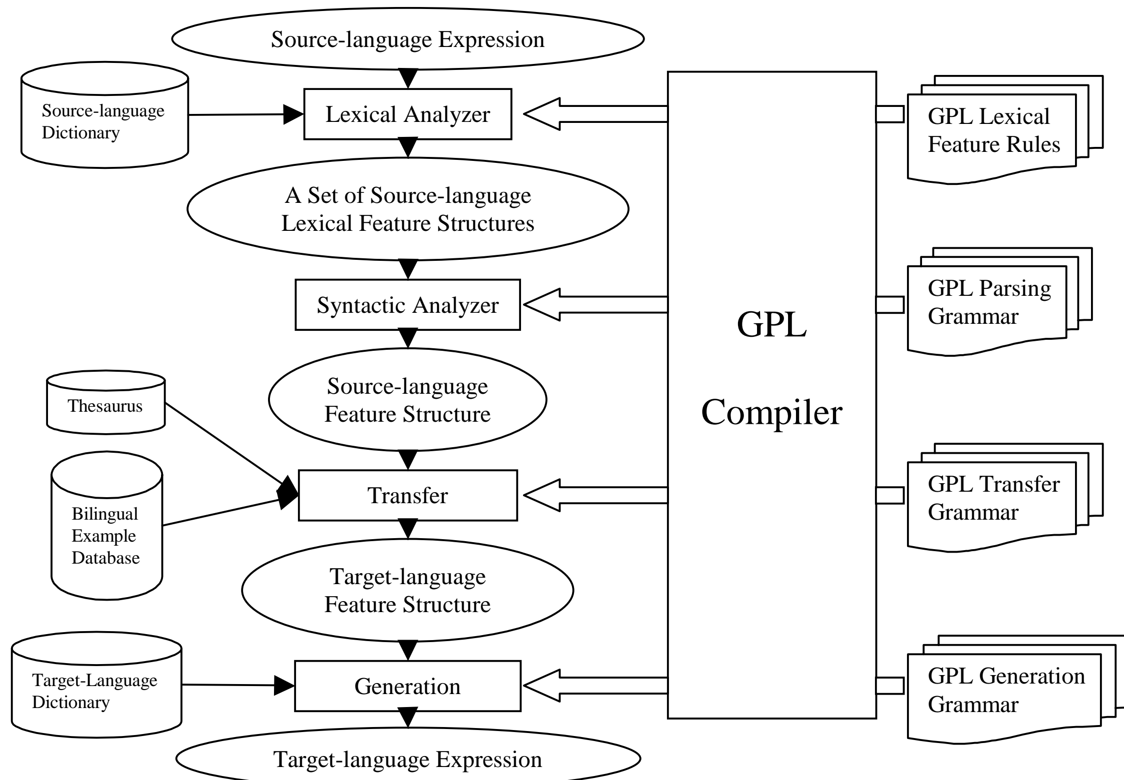


Figure 2 GPL-based Machine Translation Architecture

3.2 GPL Run-time Environment

In addition to the compiled GPL grammar and the software engine, the GPL run-time environment includes the feature structure library, a system library that provides services for representing, testing, and manipulating feature structures. Another component of the run-time environment is the memory management system for feature structures. This component manages stacks of memory pages that are used by the feature structure library to represent feature structures. This scheme provides a mechanism for garbage collection that is so fast that garbage collection can be performed after every attempted GPL rule application.

Per-rule garbage collection means that all temporary read/write memory used by a failed attempt to apply a rule is recycled immediately. If a rule application is successful, the only read/write memory that is used "permanently" is the memory required to store the feature structure(s) that represent the result of the rule, and that memory too can be recycled as soon as e.g. the relevant component has finished processing. In our experiments with English and Japanese parsing, we found that per-rule garbage collection reduced the overall read/write memory requirements by as much as a factor of four to six.

One more aspect of the GPL run-time environment concerns the "linguistic ware" or "lingware" that is available. This includes dictionaries, a semantic hierarchy, an example database (for transfer), and so on. All of this information is represented as feature structures, and GPL includes operators to query or retrieve information from the various lingware components. Figure 1 summarizes the GPL run-time environment.

3.3 GPL-based Machine Translation

We have constructed an English-Japanese Machine Translation system using a number of GPL-based components. The system has a vocabulary of 7,500 words, and it covers the "overseas travel" domain. GPL is used in four on-line components: Japanese lexical disambiguation, parsing, transfer, and generation. In addition, GPL is also used in an off-line component for generating a corpus from rules and semantic classes, and in another off-line component for performing systematic changes in collections of feature structure files (such as dictionary and example database files).

In the Japanese lexical disambiguation component, GPL is used to implement the logic that uses features from lexical entries to determine whether certain Japanese morphemes may follow each other. The component starts with a lattice that contains all possible Japanese morphemes given the input and the lexicon entries, and it reduces the size of the lattice by eliminating all paths that violate the morpheme sequencing rules.

In the English and Japanese parsing components, GPL is used for the parsing grammars, and a Generalized LR parsing software engine is used to provide the control structure.

The transfer component uses a hybrid transfer algorithm that combines feature structure rules with a large example database, where the example pairs are also represented as feature structures. The GPL transfer grammars specify restrictions on the example-matching engine, and they carry out tests and manipulations on the feature structures during the transfer process.

Finally, the generation component includes a software engine that applies all possible rules from the syntactic GPL generation grammars to derive a "generation tree". The target language expression is generated from the generation tree by applying morphological GPL rules. The architecture for the system is outlined in Figure 2.

Overall, the system includes over 40,000 lines of GPL instructions. Due to the expressive power and ease of use of GPL for language engineering, we were able to improve the various linguistic GPL-based components to raise the rate of "acceptable" translations from 17% to 84% within 12 months. At the same time, the GPL-based architecture resulted in very low memory requirements.

Read-only Memory for Code and Data	6MB
Read-only Memory for Dictionary, Examples, etc.	20MB
Read/Write Memory for Feature Structures	14MB
Read/Write Memory for Software Engines	4MB

In addition, as the following measurements from a Pentium III at 500 MHz show, the computational requirements are also very low:

Average time to translate one sentence	1.0 secs
Average amount of memory allocated and freed from the OS for one utterance	138 KBytes

4 CONCLUSIONS

We have discussed a powerful framework for constructing NLP components that are based on a common feature structure manipulation language, and we have described an MT system constructed from such components. One goal for future work is the closer integration of the feature structure type definition into the system, in order to improve error checking during compilation, and to improve run-time performance. In addition, we are planning to create an integrated development environment with a debugger and a graphical data inspector to improve the GPL grammar development process.

5 REFERENCES

[Bresnan 1992] Bresnan, J. W. (1982). *The Mental Representation of Grammatical Relations.* MIT Press, Cambridge, MA.

[Carpenter and Penn 1999] Carpenter, B. and Penn, G. (1999). *ALE: The Attribute Logic Engine.* User's Guide. Lucent Bell Laboratories and University of Tuebingen.

[Goetz and Meurers 1997] Goetz, T. and Meurers, W. D. (1997). The ConTroll system as large grammar development platform. In *Proceedings of the Workshop on Computational Environments for Grammar Development and Linguistic Engineering (ENVGRAM),* ACL/EACL-97, Madrid, Spain.

[Franz et al. 2000] Franz, A. and Horiguchi, K. and Duan, L. (2000). An Imperative Programming Language for Spoken Language Translation. In *Proceedings of the International Conference on Spoken Language Translation (ICSLP-2000),* Beijing.

[Jensen et al. 1993] Jensen, K., Heidorn, G. E., and Richardson, S. D., editors. *Natural Language Processing: The PLNLP Approach.* Kluwer Academic Publishers, Boston, MA.

[Makino et al. 1998] Makino, T., Yoshida, M., Torisawa, K., and Tsujii, J. (1998). LiLFeS – towards a practical HPSG parser. In *Proceedings of COLING-ACL '98*, pages 807-811, Montreal, Canada.

[Shieber 1986] Shieber, S. (1986). An Introduction to Unification-based Approaches to Grammar, volume 4 of CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, CA.

[Tomita 1988] Tomita, M. (1988). The generalized LR parser/compiler (Version 8.1): User's Guide. Technical Memorandum CMU-CMT-88-MEMO, Center for Machine Translation, Carnegie Mellon University, Pittsburgh, PA.

[Tomita and Knight 1988] Tomita, M. and Knight, K. (1988). Pseudo-unification and full-unification.. Technical Memorandum CMU-CMT-88-MEMO, Center for Machine Translation, Carnegie Mellon University, Pittsburgh, PA.

[Wintner and Francez 1999] Wintner, S. and Francez, N. (1999). Efficient implementation of unification-based grammars. *Journal of Language and Computation*, Volume 1, Number 1, pages 53-92.