An Imperative Programming Language for Spoken Language Translation

Alexander Franz, Keiko Horiguchi, Lei Duan

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

Abstract

This paper describes the Grammar Programming Language (GPL), a new formalism for feature-structure-based linguistic computation. GPL was designed to meet the needs of spoken language translation. GPL is easy to use, concise, and efficient, and it allows the direct expression of detailed linguistic algorithms. GPL was used successfully as the basis of Sony's machine translation project.

1 INTRODUCTION

Declarative grammar formalisms, such as the feature logics that are used with constraint-based linguistic theories like HPSG, "define associations between strings and informational elements in terms of what associations are permissible, not how they are computed" [Shieber 1986]. In contrast, in our work on spoken language translation, we have needed to develop linguistic algorithms that are most clearly and most directly described using the vocabulary of imperative programming -- in terms of series of steps, variables, assignments, tests, loops, branching control flow, etc. For this reason, we have designed and implemented a special-purpose language that is based on the well-established framework of manipulating feature structures that represent linguistic information.

2 GPL OVERVIEW

GPL (Grammar Programming Language) was designed to make it possible to spell out the series of steps that guide an NLP component to a solution in terms of the required tests and manipulations of linguistic information. GPL incorporates as many good ideas from related work as possible, but it remains neutral with respect to (and therefore not limited by) particular linguistic theories or analyses.

2.1 Rewrite Grammars

GPL follows the tradition of rewrite-rules that are annotated with feature structure information. A GPL program consists of a set of context-free rules that include a GPL "rule-body", which is a block of GPL statements. Individual GPL rules can succeed or fail, depending on the results of executing the rule-body. GPL Grammars are interpreted in the appropriate manner for the respective components, such as parsing or generation.

2.2 Feature Structures

GPL uses feature structures as the central data structure for representing linguistic information. Rather than attempting to use unification for all information manipulation, GPL includes separate operators for testing and copying (amongst others), since we find that this expresses our linguistic algorithms more clearly. GPL's feature structures are trees, not graphs. They are not re-entrant, and instead of structure-sharing we employ copying. This accords well with our imperative style of linguistic algorithm design, and it makes it much more efficient to manage the temporary feature structures that are created during processing.

In GPL, feature names are not required to be unique, but their order is significant. This means that it is valid to have two neighbouring features labelled NP, for example. As a corollary, the order of features is significant. This allows us to use certain types of linguistic representations that we have found desirable.

2.3 The GPL Compiler

GPL grammars are compiled by the GPL compiler into C code, which is then compiled with a C compiler and linked with system libraries and the required software engine (e.g. a parsing or generation engine). The GPL compiler generates code to handle nested tests and disjunction in feature structures in an efficient manner. The GPL run-time environment includes a memory manager with a garbage collection scheme that is fast enough to allow garbage collection of temporary feature structures after every GPL rule application. For more details, please see [Duan et al. 2000].

2.4 Previous Work

GPL echoes many ideas found in Tomita et al.’s parsing, transformation, and generation systems, and in the "pseudo-unification" formalism [Tomita 1988], [Tomita and Nyberg 1988], and [Tomita and Knight 1988]. That work, in turn, is related to LFG [Bresnan 1982], PATR-II [Shieber 1986], and earlier work on unification grammar. An extremely faithful implementation of LFG can be found in [Kaplan and Maxwell 1996]. Logic-programming-based HPSG implementations are described in [Carpenter and Penn 1999], [Makino et al. 1998], [Copestake 1999], and Goetz and Meurers 1997].
This section provides a gentle introduction to GPL by way of an example. Consider the (simplified) rule in Figure 1. It is used to generate a yes-no question, such as “Does the shoe fit?”

The input feature structure for this rule is shown below.

```
DEFINITE +
HEAD [ ROOT "shoe"]
PERSON THIRD
NUMBER SINGULAR
VP

DEFINITE +
HEAD [ ROOT "fit"]
TENSE PRES
TYPE VERBAL-VP
S-TYPE
TYPE YN-QUESTION
SEND
```

This feature structure corresponds to the non-terminal YN_SENT, and it can be referenced from within the rule using the symbol $m. The rule should only apply if the input feature structure has a modal auxiliary (such as “can” or “will”), or if the input is not a copula sentence. This is checked in line 2 of the rule. In line 3, the sub-feature-structure at the path “VP SUBJ” is assigned as the value of $d2, and removed from $m. This means the non-terminal SUBJ takes on the following value:

```
DEFINITE +
HEAD [ ROOT "shoe"]
PERSON THIRD
NUMBER SINGULAR
TYPE SIMPLE-NP
```

Similarly, line 4 assigns the value of $m's VP feature to $d3, corresponding to the non-terminal VP. Line 5 adds the feature-value pair (VFORM BASE), so the resulting feature structure for $d3 becomes the following:

```
DEFINITE +
HEAD [ ROOT "fit"]
TENSE PRES
TYPE VERBAL-VP
VFORM BASE
```

The final section of the rule-body determines which auxiliary should be used. Line 6 checks whether the input feature structure ($m) carries a MODAL-AUX slot. If so, and if the value of the TENSE feature in the VP were equal to PAST, then line 8 would add the feature-value pair (ASPECT PERFECTIVE) to $d3, the feature structure for the non-terminal VP.

Line 11 applies if there was no modal auxiliary. In this case, if the VP was in the past tense, then the auxiliary “did” would be used. Else, lines 13-14 test for the case where the PERSON of the SUBJ is something other than FIRST and SECOND, and where the NUMBER is not equal to PLURAL. This test succeeds for our example, so line 15 sets the value of the path “HEAD ROOT” to the string “does” in the feature structure for $d1, and the rule application succeeds.

### 4 GPL DETAILS

This section provides a detailed listing of the various features of GPL. All examples represent actual GPL code.

#### 4.1 Data Types

GPL supports a number of atomic and other simple data types, including strings, symbols, numbers, feature structures, feature structure paths, and sub-feature-structure specifications. Atomic types and feature structures can be combined into derived data types using operators for disjunction and
conjunction. In our applications, disjunction is used widely to represent linguistic ambiguity or choice. Conjunction, on the other hand, is used more rarely for multiple appropriate representations.

4.2 Variables and Tests

In a GPL rule, each non-terminal from the context-free part represents a feature structure, and the GPL statements refer to these feature structures using $m$ for the left-hand side (mother) feature structure, and $d1$, $d2$, ... for the right-hand-side (daughter) feature structures. For example, given the rule $S --> NP VP$, $d1$ refers to the feature structure for the NP non-terminal. GPL supports global variables and local variables, as well as local pointers that can be used to automatically modify sub-feature-structures without explicit pointer manipulations.

Since GPL does not require slot names to be unique, it provides a mechanism to use tests to select appropriate sub-feature-structures. A local pointer can be initialised to the first sub-feature-structure at a given path that satisfies a test. Similarly, the find-substruct and find-slot operators can be used to locate sub-feature-structures that satisfy a test, regardless of the path.

4.3 Predicates and Expressions

GPL includes a wide variety of predicates for testing various aspects of feature structures. Predicates return True or False. Predicates include test-path and test-path-not to check for the presence/absence of a path; test-feature and test-feature-not to check for atomic feature values; and test-feature-oneof, test-feature-not-oneof, test-feature-member, and test-feature-not-member to check for set membership of atomic values.

Additional test operators include test-feature-equal-feature and test-feature-not-equal-feature to test whether two atomic sub-feature-structures are equal; and test-function, test-feature-equal-function, and test-feature-not-equal-function to test whether an external function returns True, or returns a value that is equal to a specified atomic sub-feature structure. Overloading is used to allow grammar writers to designate these operators using only a few intuitively clear keywords, including ==, !==, ?member, !member, etc.

GPL expressions are used to place restrictions on the application of a rule, and to determine control flow within the rule-body of an individual rule. An expression consists of a simple predicate (listed above), or of a boolean combination of predicates with OR and AND.

4.4 Manipulation Operators

GPL includes a wide range of operators for manipulating feature structures. This includes operators for copying, moving, removing, and assigning feature values and sub-feature-structures.

Assignment operators, which are mostly designated by the overloaded keyword =, include assign-feature, assign-feature-front (this adds any new features to the front of the feature structure – remember that feature order is significant), add-feature-conjunction, add-feature-conjunction-front, assign-feature-function and assign-feature-function-front to assign the value returned by a function; and copy, copy-front, copy-new-slot, and copy-new-slot-front.

Other manipulation operators include remove-sub-feature-structure, move, rename-slot, and rename-slot-test (which uses an embedded test to identify the slot that should be renamed).

4.5 Control Flow

GPL offers a number of statements for control flow, including a conditional if-then-else construct, a switch statement, and two loop statements. Explicit succeed and fail statements can be used anywhere in a rule to cause rule application success or failure.

if-then-else. This construct accepts any GPL expression (see Section 4.3) for the test.

switch. The switch statement can operate on any atomic type (string, symbol, or number).
**loop-slot.** This statement causes a loop over all the slots at the given level of the feature structure. The following example uses the show statement, which is used during debugging to print out a sub-feature-structure. At each iteration, the variable SLOT is set to point to the next slot.

```plaintext
loop-slot $SLOT in $INPUT {
    if (?exist [$SLOT VALUE UNALIGNED]) {
        show [$SLOT VALUE] "Slot:\n";
    }
}
```

**loop-subfstruct.** This causes a loop over all nested sub-feature-structures in the given feature structure. In the example, at each iteration, the variable SUB is set to point to the next sub-feature-structure.

```plaintext
loop-subfstruct $SUB in $SOURCE {
    show [$SUB VALUE]  "Found sub-fs:\n";
}
```

**continue.** This statement causes either loop to proceed to the next iteration of the loop.

## 5 EVALUATION AND CONCLUSIONS

We have implemented an English-Japanese spoken language translation system with a vocabulary of 7,500 words covering the “overseas travel” domain. This domain is characterized by a high degree of linguistic variability, which necessitates detailed linguistic representations and processing. The system uses GPL in four components: (1) Lexical ambiguity resolution in the input word-graph, (2) parsing, (3) transfer, and (4) generation. Overall, the system includes over 40,000 lines of GPL instructions.

Since GPL is easy to use and allows the direct expression of the required linguistic algorithms, we were able to develop and refine the system rather quickly: Between May 1999 and April 2000, we were able to raise the rate of "acceptable" English-to-Japanese translations from 17% to 84%. The results from our latest evaluation on unseen test data are summarized below.

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flawless</td>
<td>60%</td>
</tr>
<tr>
<td>Stylistic Problem</td>
<td>9%</td>
</tr>
<tr>
<td>Minor Problem</td>
<td>14%</td>
</tr>
<tr>
<td>Acceptable with OOV</td>
<td>1%</td>
</tr>
<tr>
<td>Major Problem</td>
<td>9%</td>
</tr>
<tr>
<td>Wrong Translation</td>
<td>5%</td>
</tr>
<tr>
<td>Translation Failure</td>
<td>3%</td>
</tr>
</tbody>
</table>

One goal for future work is the creation of an integrated development environment for GPL, including a debugger and a graphical data inspector. Another goal is to integrate the feature structure type definition more closely into the system.

## 6 REFERENCES


