# The Generalized LR Parser/Compiler V8-4: A Software Package for Practical NL Projects

**Masaru Tomita**
School of Computer Science and
Center for Machine Translation
Carnegie Mellon University
Pittsburgh, PA 15213, USA
mt@cs.cmu.edu

## 1. Introduction

This paper[1]describes a software package designed for practical projects which involve natural language parsing. The Generalized LR Parser/Compiler V8-4 is based on Tomita's Generalized LR Parsing Algorithm [7, 6], augmented by pseudo/full unification modules. While the parser/compiler is not a commercial product, it has been thoroughly tested and heavily used by many projects inside and outside CMU last three years. It is publicly available with some restrictions for profit-making industries[2]. It is written entirely in CommonLisp, and no system-dependent functions, such as window graphics, are used for the sake of portability. Thus, it should run on any systems that run CommonLisp in principle[3], including IBM RT/PC, Mac II, Symbolics and HP Bobcats.

Each rule consists of a context-free phrase structure description and a cluster of *pseudo equations* as in figure 1-1. The non-terminals in the phrase structure part of the rule are referenced in the equations as `x0...xn`, where `x0` is the non-terminal in the left hand side (here, <DEC>) and `xn` is the n-th non-terminal in the right hand side (here, `x1` represents <NP> and `x2` represents <VP>). The pseudo equations are used to check certain attribute values, such as verb form and person agreement, and to construct a f-structure. In the example, the first equation in the example states that the case of <NP> must be nominative, and the second equation states that the form of <VP> must be finite. Then one of the following two must be true: (1) the time of <VP> is present and agreements of <NP> and <VP> agree, OR (2) the time of <VP> is past. If all of the conditions hold, let the f-structure of <DEC> be that of <VP>, create a slot called "subj" and put the f-structure of <NP> there, and create a slot called "passive" and put "-" there. Pseudo equations are described in detail in section 3.

```
(<DEC> <==> (<NP> <VP>)
        (((x1 case) = nom)
         ((x2 form) =c finite)
         (*OR*
           (((x2 :time) = present)
            ((x1 agr) = (x2 agr)))
           (((x2 :time = past)))
         (x0 = x2)
         ((x0 subj) = x1)
         ((x0 passive) = -)))
```

**Figure 1-1:** A Grammar Rule for Parsing

Grammar compilation is the key to this efficient parsing system. A grammar written in the correct format is to be compiled before being used to parse sentences. The context-free phrase structure rules are compiled into an *Augmented LR Parsing Table*, and the equations are compiled into CommonLisp functions. The runtime parser then does the shift-reduce parsing guided by the parsing table, and each time a grammar rule is applied, its CommonLisp function compiled from equations is evaluated.

[1]Many members of CMU Center for Machine Translation have made contributions to the development of the system. People who implemented parts of the system, besides the author, are: Hideto Kagamida, Kevin Knight, Hiroyuki Musha and Kazuhiro Toyoshima. People who made contributions in maintaining the system include: Steve Morrisson, Eric Nyberg, Hiroaki Saito and Hideto Tomabechi. People who provided valuable comments/bug reports in writing and debugging grammars include: Donna Gates, Lori Levin, Toru Matsuda and Teruko Mitamura. Other members who made indirect contributions in many ways include: Ralph Brown, Jaime Carbonell, Marion Kee, Sergei Nirenburg and Koichi Takeda.

[2]For those interested in obtaining the software, contact Radha Rao, Business Manager, Center for Machine Translation, Carnegie Mellon University, Pittsburgh, PA 15213 (rdr@nl.cs.cmu.edu).

[3]In practice, however, we usually face one or two problems when we transport it to another CommonLisp system, due to bugs in CommonLisp and/or file I/O complications.

In the subsequence sections, features of the Generalized LR Parser/Compiler v8-4 are briefly described.

## 2. Top-Level Functions

There are three top-level functions:

```
        ; to compile a grammar
   (compgra grammar-file-name)

        ; to load a compiled grammar
   (loadgra grammar-file-name)

        ; to parse a sentence string
   (p sentence)
```

## 3. Pseudo Equations

This section describes pseudo equations for the Generalized LR Parser/Compiler V8-4.

### 3.1. Pseudo Unification, =

*path = val*

Get a value from *path*, unify it with *val*, and assign the unified value back to *path*. If the unification fails, this equation fails. If the value of *path* is undefined, this equation behaves like a simple assignment. If *path* has a value, then this equation behaves like a test statement.

*path1 = path2*

Get values from *path1* and *path2*, unify them, and assign the unified value back to *path1* and *path2*. If the unification fails, this equation fails. If both *path1* and *path2* have a value, then this equation behaves like a test statement. If the value of *path1* is not defined, this equation behaves like a simple assignment.

### 3.2. Overwrite Assignment, <=

*path <= val*

Assign *val* to the slot *path*. If *path1* is already defined, the old value is simply overwritten.

*path1 <= path2*

Get a value from *path2*, and assign the value to *path1*. If *path1* is already defined, the old value is simply overwritten.

*path <= lisp-function-call*

Evaluate *lisp-function-call*, and assign the returned value to *path*. If *path1* is already defined, the old value is simply overwritten. *lisp-function-call* can be an arbitrary lisp code, as long as all functions called in *lisp-function-call* are defined. A path can be used as a special function that returns a value of the slot.

### 3.3. Removal Assignment, ==

*path1 == path2*

Get a value from *path2*, assign the value to *path1*, and remove the value of *path2* (assign nil to *path2*). If a value already exists in *path1*, then the new value is unified with the old value. If the unification fails, then this equation fails.

### 3.4. Append Multiple Value, >

*path1 > path2*

Get a value from *path2*, and assign the value to *path1*. If a value already exists in *path1*, the new value is appended to the old value. The resulting value of *path1* is a multiple value.

### 3.5. Pop Multiple Value, <

*path1 < path2*

The value of *path2* should be a multiple value. The first element of the multiple value is popped off, and assign the value to *path1*. If *path1* already has a value, unify the new value with the old value. If *path2* is undefined, this equation fails.

### 3.6. *DEFINED* and *UNDEFINED*

*path* = `*DEFINED*`

Check if the value of *path* is defined. If undefined, then this equation fails. If defined, do nothing.

### 3.7. Constraint Equations, =c

*path =c val*

This equation is the same as an equation

*path = val*

except if *path* is not already defined, it fails.

### 3.8. Removing Values, *REMOVE*

*path* = `*REMOVE*`

This equation removes the value in *path*, and the path becomes undefined.

### 3.9. Disjunctive Equations, *OR*

```
(*OR* list-of-equations
      list-of-equations ....)
```

All lists of equations are evaluated disjunctively. This is an inclusive OR, as oppose to exclusive OR; Even if one of the lists of equations is evaluated successfully, the rest of lists will be also evaluated anyway.

### 3.10. Exclusive OR, *EOR*

```
(*EOR* list-of-equations
       list-of-equations ....)
```

This is the same as disjunctive equations `*OR*`, except an exclusive OR is used. That is, as soon as one of the element is evaluated successfully, the rest of elements will be ignored.

### 3.11. Case Statement, *CASE*

```
(*CASE* path
   (key1 equation1-1 equation1-2 ...)
   (Key2 equation2-1 ...)
   (Key3 equation3-1) ....)
```

The *CASE* statement first gets the value in *path*. The value is then compared with Key1, Key2, ...., and as soon as the value is `eq` to some key, its rest of equations are evaluated.

### 3.12. Test with an User-defined LISP Function, *TEST*

```
(*TEST* lisp-function-call)
```

The *lisp-function-call* is evaluated, and if the function returns nil, it fails. If the function returns a non-nil value, do nothing. A path can be used as special function that returns a value of the slot.

### 3.13. Recursive Evaluation of Equations, *INTERPRET*

```
(*INTERPRET path)
```

The *INTERPRET* statement first gets a value from *path*. The value of *path* must be a valid list of equations. Those equations are then recursively evaluated. This *INTERPRET* statement resembles the "eval" function in Lisp.

### 3.14. Disjunctive Value, *OR*

```
(*OR* val val ...)
```

Unification of two disjunctive values is set interaction. For example, `(unify '(*OR* a b c d) '(*OR* b d e f))` is `(*OR* b d)`.

### 3.15. Negative Value, *NOT*

```
(*NOT* val val ...)
```

Unification of two negative values is set union. For example, `(unify '(*NOT* a b c d) '(*NOT* b d e f))` is `(*NOT* a b c d e f)`.

### 3.16. Multiple Values, *MULTIPLE*

```
(*MULTIPLE* val val ...)
```

Unification of two multiple values is append. When unified with a value, each element is unified with a value. For example, `(unify '(*MULTIPLE* a b c d b d e f) 'd)` is `(*MULTIPLE* d d)`.

### 3.17. User Defined special Values, **user-defined*

The user can define his own special values. An unification function with the name `UNIFY*`*user-defined*`*` must be defined. The function should take two arguments, and returns a new value or *FAIL* if the unification fails.

## 4. Standard Unification Mode

The pseudo equations described in the previous section are different from what functional grammarians call "unification". The user can, however, select "full (standard) unification mode" by setting the global variable `*UNIFICATION-NODE*` from `PSEUDO` to

`FULL`. In the full unification mode, equations are interpreted as standard equations in a standard functional unification grammar [5], although some of the features such as user-defined function calls cannot be used. However, most users of the parser/compiler find it more convenient to use PSEUDO unification than FULL unification, bot only because it is more efficient, but also because it has more practical features including user-defined function calls and user-defined special values. Those practical features are crucial to handle low-level non-linguistic phenomena such as time and date expressions [8] and/or to incorporate semantic and pragmatic processing of the user's choice. More discussions on PSEUDO and FULL unifications can be found in [10].

## 5. Other Important Features

### 5.1. Character Basis Parsing

The user has a choice to make his grammar "character basis" or standard "word basis". When "character basis mode" is chosen, terminal symbols in the grammar are characters, not words. There are at least two possible reasons to make it character basis:

1. Some languages, such as Japanese, do not have a space between words. If a grammar is written in character basis, the user does not have to worry about word segmentation of unsegmented sentences.
2. Some languages have much more complex morphology than English. With the character basis mode, the user can write morphological rules in the very same formalism as syntactic rules.

### 5.2. Wild Card Character

In pseudo unification mode, the user can use a wild card character "%" in his grammar to match any character (if character basis) or any word (if word basis). This feature is especially useful to handle proper nouns and/or unknown words.

### 5.3. Grammar Debugging Tools

The Generalized LR Parser/Compiler V8-4 includes some debugging functions. They include:

- `dmode` --- debugging mode; to show a trace of rule applications by the parser.
- `trace` --- to trace a particular rule.
- `disp-trees, disp-nodes, etc.` --- to display partial trees or values of nodes in a tree.

All of the debugging tools do not use any fancy graphic interface for the sake of system portability.

### 5.4. Interpretive Parser

The Generalized LR Parser/Compiler V8-4 includes another parser based on chart parsing which can parse a sentence without ever compiling a grammar:

; to load a grammar
(`i-loadgra` *grammar-file-name*)

; to run the interpretive parser
(`i-p` *sentence*)

While its run time speed is significantly slower than that of the GLR parser, many users find it useful for debugging because grammar does not need to be compiled each time a small change is made.

### 5.5. Grammar Macros

The user can define and use macros in a grammar. This is especially useful in case there are many similar rules in the grammar. A macro can be defined in the same way as CommonLisp macros. Those macros are expanded before the grammar is compiled.

## 6. Concluding Remarks

Some of the important features of the Generalized LR Parser/Compiler have been highlighted. More detailed descriptions can be found in its user's manual [9]. Unlike most other available software [1, 2, 4], the Generalized LR Parser/Compiler v8-4 is designed specifically to be used in practical natural language systems, sacrificing perhaps some of the linguistic and theoretical elegancy. The system has been thoroughly tested and heavily used by many users in many projects inside and outside CMU last three years. Center for Machine Translation of CMU has developed rather extensive grammars for English and Japanese for their translation projects, and some experimental grammars for French, Spanish, Turkish and Chinese. We also find the system very suitable to write and parse task-dependent semantic grammars. Finally, a project is going on at CMU to integrate the parser/compiler with a speech recognition system (SPHINX [3]).

## 7. References

[1] Karttunen, L.
D-PATR: A Development Environment for Unification-Based Grammars.
In *12th International Conference on Computational Linguistics.* Bonn, 1986.

[2] Kiparsky, C.
*LFG Manual.*
Technical Report, Xerox Palo Alto Research Center, 1985.

[3] Lee, K. F. and Hon, H. W.
Large-Vocabulary Speaker-Independent Continuous Speech Recognition.
*Proceedings of IEEE Int'l Conf. on Acoustics, Speech and Signal Processing* , 1988.

[4] Shieber, S. M.
The Design of a Computer Language for Linguistic Information.
In *10th International Conference on Computational Linguistics*, pages 362-366. Stanford, July, 1984.

[5] Shieber, S. M.
*CSLI Lecture Notes: An Introduction to Unification Approaches to Grammar.*
Center for the Study of Language and Information, 1986.

[6] Tomita, M.
*Efficient Parsing for Natural Language.*
Kluwer Academic Publishers, Boston, MA, 1985.

[7] Tomita, M.
An Efficient Augmented-Context-Free Parsing Algorithm.
*Computational Linguistics* 13(1-2):31-46, January-June, 1987.

[8] Tomita, M.
Linguistic Sentences and Real Sentences.
*12th International Conference on Computational Linguistics* , 1988.

[9] Tomita, M., Mitamura, T. and Kee, M.
*The Generalized LR Parser/Compiler: User's Guide.*
Technical Report, Center for Machine Translation, Carnegie-Mellon University, 1988.

[10] Tomita, M. and Knight, K.
*Pseudo Unification and Full Unification.*
Technical Report unpublished, Center for Machine Translation, Carnegie-Mellon University, 1988.