ICOT Technical Report: TR-521

TR-521

Language Tool Box (LTB) A Program Library of NLP Tools

by
K. Akasaka, Y. Kubo, F. Fukumoto,
H. Fukushima & K. Hagiwara

November, 1989

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Language Tool Box (LTB) A Program Library of NLP Tools

Kouji AKASAKA , Yukihiro KUBO , Fumiyo FUKUMOTO
Hideaki FUKUSHIMA , Kaoru HAGIWARA
Institute for New Generation Computer Technology (ICOT)

Abstract

LTB (Language Tool Box) is a program library containing indispensable NLP tools which can be used as common building blocks in developing a variety of NLP systems. The LTB provides many NLP researchers with good facilities such as efficient analysis and generation module, well-designed and rich language data, or friendly user interface to operate them. The current system consists of a lexical analyzer, a syntax analyzer, a sentence generator, dictionaries, a language for semantic description CIL, and an LTB-Shell. Most of the software modules are written in ESP (object oriented Prolog), and the rest of them in CIL (Prolog augmented by a record-like data structure). This paper is an introduction to the LTB, presenting its currently available software facilities. A number of window images of some tools are provided in order to give the readers a feeling for our system. The future research and development direction are also discussed.

1 Introduction

LTB (Language Tool Box) is a program library containing indispensable NLP tools which can be used as common building blocks in developing a variety of NLP systems. The **LTB**, (the **Language Tool Box**), is implemented on PSI[1], a logic-programming machine newly developed at ICOT.

The LTB is intended to provide many NLP researchers and NLS (Natural Language processing System) developers with good facilities, such as efficient analysis and generation modules, a well-designed and rich data in dictionaries, and a friendly user interface. Such facilities can be used in NLSs as parts of the systems and also provide NLP researchers with a workbench.

The LTB was originated from our experimental implementation of discourse understanding system (named **DUALS**) [2]. Through the implementation of DUALS, we keenly realized the need for efficient software modules that have a basic NLP functions environment which could support the development of sophisticated experimental NLSs. To meet this need, we started the development of the LTB by extracting the modules of DUALS which had basic NLP functions such as syntax analysis and sentence generation. Then we refined the modules from the point of an efficiency and a modularity. After that, we equipped all of them with some peripheral software tools, such as debugging tools or dedicated editors. In addition, we are designing a shell to coordinate those modules in a uniform manner so that the modules will form an integrated workbench. The effectiveness of the LTB has been evaluated in the development of the latest version of DUALS.

Generally, a desirable environment for the development of NLSs provides the following: 1) **excellent description formats** for linguistic data, 2) **interactive facilities** to support the user. Furthermore, it should have 3) **high modularity**, and be 4) an **integrated environment**. As for 1), in the LTB, each module has its own description format, suitable to its aspect of NLP, to encode linguistic information. However we feel the need to improve the flexibility and readability of theses formats. With regard to 2), each module of the LTB is equipped with various types of interactive facilities, such as debuggers or editors, each of which fully takes advantage of a multi-window and graphic environment provided by PSI. In regard to 3) and 4), two types of Prolog-based language, in either of which all the LTB modules were written, played important roles. One is ESP[3] (Extended Self-contained Prolog), a sort of object- oriented Prolog, which describes the underlying PSI/SIMPOS. Written in ESP, all modules have high modularity and compatibility with each other. The other is CIL[4], an extended Prolog with a record-like structure, called **PST (Partially Specified Term)**. PST, used by every module, plays the role of a common data structure for internal representations throughout the LTB's module: both analysis module and generation module.

The current version of the LTB is composed of the following modules:

CIL: The basic programming language of the the LTB. Every tool in the LTB has access to CIL and its programming environments.

DataBase: The **master dictionary** and the **thesaurus**, both of which are referred by both analysis modules and the generation module.

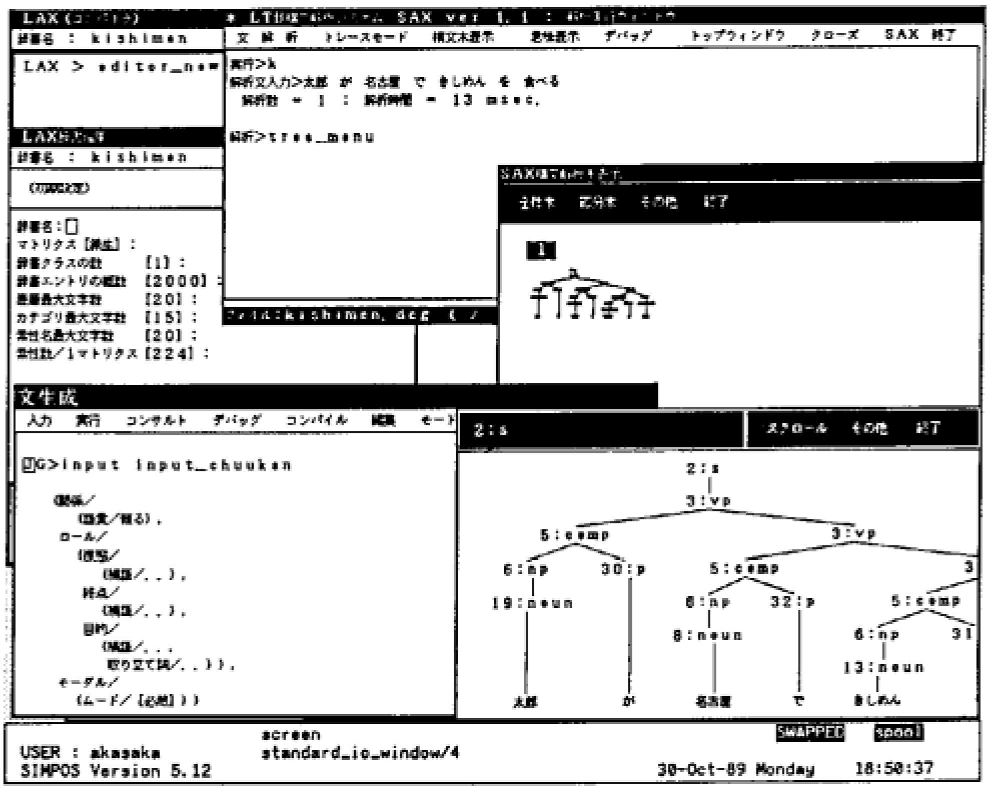
LAX: The morphological and semantic analyzer.

SAX: The syntactic and semantic analyzer. The grammar is written in DCG.

Generator: The Japanese sentence generator, whose input is a frame structure written in CIL, outputs Japanese surface sentences.

Shell: The LTB-shell coordinates the operation of every module in a uniform way, and it manages communications between any two modules.

In the following sections, the details of the above modules except for **DataBase** are described in that order.



```
辞書名 : kishimen
LAX > editor_new
辞書名 : kishimen
辞書名:
マトリクス [illegible] :
辞書クラスの数          [1] :
辞書エントリの総数      [2000] :
[illegible]最大文字数    [20] :
カテゴリ最大文字数      [15] :
素性名最大文字数        [20] :
素性数/1マトリクス      [224] :
```

```
実行>k
解析文入力>太郎 が 名古屋 で きしめん を 食べる
 解析数 = 1 : 解析時間 = 13 msec.

解析>tree_menu
```

```
ファイル:kishimen.dcg
```

```
G>input input_chuuken

    (関係/
       ([illegible]/[illegible]),
     ロール/
       ([illegible]/
          ([illegible]/..),
        [illegible]/
          ([illegible]/..),
        目的/
          ([illegible]/...
           取り立て詞/..)),
     モーダル/
       (ムード/[必然]))
```

```
                              screen
USER : akasaka                standard_io_window/4
SIMPOS Version 5.12                                         30-Oct-89 Monday   18:50:37
```

Figure 1: LTB on PSI machine

2 CIL

This section outlines the semantic processing language called **CIL** and its programming environment. See [5] for further information.

CIL, employed by every other module, plays the role of the basic programming language of the LTB.

2.1 Outline of CIL

CIL was developed to make description of natural language processing systems easy. It has two augmentations to Prolog. One is a record-like structure called a **partially specified term** (PST). The other is the **freeze mechanism**, originating in PrologII[6]. A PST is a set of attribute-value pairs in the form:

$$\{\mathtt{a}_1/\mathtt{b}_1, \ldots, \mathtt{a}_n/\mathtt{b}_n\}$$

where

$$n \geq 0, \quad \mathtt{a}_i \neq \mathtt{a}_j \quad (i \neq j)$$

A PST can be seen as an abstraction of the following data structure: a **category** of GPSG, a **functional structure** of LFG, and an assignment of Situation Theory. A PST is a simple but expressive data structure to describe various complicated objects, which appear in the NLP.

CIL provides various built-in-predicates and syntax-sugar to make the operation on PSTs easy.

2.2 Programming Environment

Figure 2 shows the configuration of the CIL system. The CIL programming environment consists of the following submodules.

Command Interpreter: The top level user interface, from which the user gives commands to other tools.

Compiler: The Compiler translates a given CIL program into an optimized ESP code which runs efficiently.

Interpreter: With the interpreter, users can run their CIL program immediately without having to wait for a long compile time.

Cil editor A text editor with a syntax-checker.

Inspector A utility for inspecting a nested PST.

Debug Aid A full-screen source image tracer which enables users to debug visually and interactively.

3 Lexical Analyzer/*LAX*

In this section, we illustrate a Japanese lexical analyzer called *LAX* (Lexical Analyzer for Syntax and Semantics) and its software environment called the *LAX system*.

First we present an outline of this system, then describe the flow of the lexical analysis, which consists of two modules: morphological analysis module and semantic construction module. Finally, we explain the configuration of the LAX system and its useful development tools.

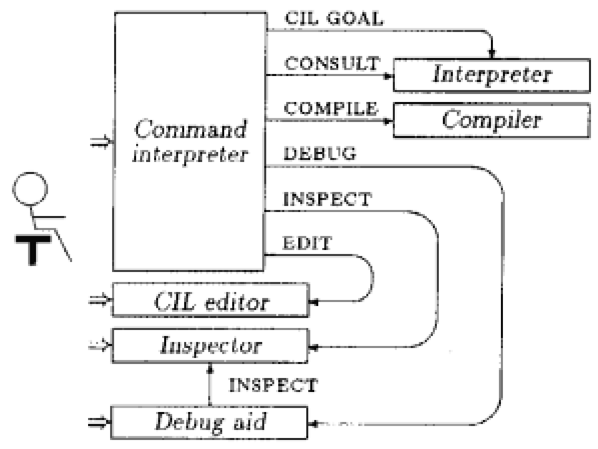


Figure 2: Configuration of the CIL system and user interface

3.1 Outline of the LAX system

In processing a Japanese sentence, morphological analysis is more important than is the case in other languages, mainly due to the following two reasons:

- Unlike European languages such as English, Japanese sentences are ordinarily written in a way that the words in the sentences are not separated by delimiters such as blanks. So a Japanese sentence analyzer must recognize the words which compose the sentence.

- The second reason arises from *Kanji* (Chinese characters). Kanji can form a compound word (Kanji compound word), which may be very complex. Therefore, a Japanese sentence analyzer must recognize the morphemes in a Kanji compound word, and derive the meaning of the word from the meaning of each morpheme of the word.

For the above reasons, we needs an efficient morphological analysis system, for the processing of Japanese sentences. *LAX* was developed to satisfy this demand.

The LAX system provides a software environment for the Japanese lexical analyzer (LAX), which consists of morphological analysis module and semantic construction module.

In using the LAX system, the user first develops a morphological dictionary called *LAX−dictionary*, in which a morphological information (*connective rules*) and semantic information (*semantic construction rules*) are described for each

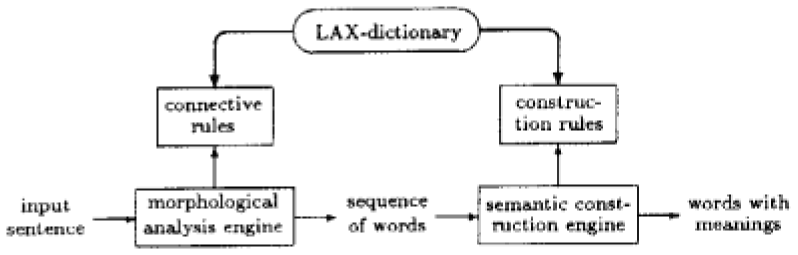


Figure 3: Flow of the Lexical Analysis in LAX

morpheme. Intuitively, connective rules indicate what kind of morphemes can be concatenated with the described morpheme, and semantic construction rules indicate what to do in order to construct meanings of words. Then LAX-dictionary is translated into a lexical analyzer LAX.

In order to edit the LAX-dictionary, we can use a special tool called a Dictionary editor. The Inspector provides a debugging environment, which enables us to check the result of morphological analysis and trace the semantic construction of each word. The ambiguities in the results of the morphological analysis are displayed graphically by the tool called Pretty printer.

LAX analyzes a sentence deterministically and outputs all possible interpretations in parallel if there are ambiguities. LAX can analyze a sentence consisting of 50 characters [1] in less than 100msec, even though there are many possible solutions. This result shows that LAX is efficient enough for practical purposes. LAX can also recognize unknown morphemes occurring in the input sentence.

The result of lexical analysis done by LAX can be used as the input for the syntax and semantic analyzer SAX.

3.2 Flow of the Lexical Analysis in LAX

Figure 3 shows the flow of the lexical analysis in LAX. The lexical analyzer LAX consists of two modules, each of which consists of an engine and rules. The first engine, called morphological analysis engine, transforms a Japanese sentence as a surface string into a sequence of words that consists of several morphemes referring connective rules. The second engine, called semantic construction engine, constructs each word's meaning referring construction rules or action rules. These two rules referred by each engine are transformed from the LAX-dictionary in terms of a translator.

[1] It corresponds to an English sentence consisting of about 100 letters.

The morphological analysis mechanism employed in the morphological analysis engine is based on the layered stream technique [7], and can analyze all the possible solutions in parallel, without backtracking. The connective rules referred by this engine are compiled into the TRIE structure [8] with respect to the headword for morphemes from the LAX-dictionary for quick retrieval. Ambiguities are packed as a graph stack structure. The result of the morphological analysis is sent to the next module, semantic construction engine.

For each word, the semantic construction engine builds up its meaning from the semantic construction rule of each morpheme that constructs the word. The engine applies the construction rule assigned the first morpheme of the word, and sends the result to the next morpheme, and so forth. Finally, after applying the last construction rule, corresponding to the last morpheme, the engine outputs the meaning for the word. In this way, the engine constructs the meaning for every word, that is the output of LAX.

3.3 Syntax and Semantics of the LAX-dictionary

In this section, the description format and the semantics of the LAX-dictionary are described. The grammar category defined in the LAX-dictionary belongs to the regular grammar or type-3 grammar. This description format is designed so that the Japanese morphological grammar is written easily. Specifically, this format is aimed to implement Morioka's morphological grammar [9].

The basic concept of the morphological analysis is to check whether a certain sequence of morphemes can be acceptable as a word. So, the judging process can be regard as a non-deterministic finite state automaton (NFA). The starting and ending features of a word correspond to the initial and final state of the NFA. The LAX-dictionary is a definition of the state transition function of the NFA. Namely, the surface string of the morpheme corresponds to the input symbol, $left-hand\ feature$ to the input state, $right-hand\ feature$ to the set of states to which the NFA can transit.

Figure 4 shows the syntax of the LAX-dictionary. The morphemes that belong to the same morphological category are described together as shown by the lines from 1 to 8 in Figure 4. Each definition consists of three parts. The first part specifies a surface string as a headword and a token as a return value described in line 2.

The second part specifies, from lines 3 to 5, connective rules which consist of a left-hand feature which stands for the identifier of the described morpheme, and a right-hand feature which is a set of identifiers of the morphemes that can follow the described morpheme. Every connective feature (Cf) belongs to a certain state transition table (Stt), and should be designated with the name of the related state transition table in the LAX-dictionary. Similar features can be packed into one group in the state transition table.

The last one, on lines 6 and 7, specifies a semantic construction rules. This part consists of a declaration of variables each of which mediates the comm-

$$
\begin{array}{llll}
\mathit{begin(Category)} & & & (1)\\
\quad \mathit{Surface} \quad \mathit{(Token)} & & & (2)\\
 & :: & \mathit{Stt}_f([\mathit{Cf}_{f1},\ldots]) & (3)\\
 & \&\& & \mathit{Stt}_{b1}([\mathit{Cf}_{b11},\ldots]), & (4)\\
 & & \vdots & \\
 & & \mathit{Stt}_{bn}([\mathit{Cf}_{bn1},\ldots]), & (5)\\
 & \$\$ & [[\mathit{declaration\ of\ variables}], & (6)\\
 & & \mathit{semantic\ construction\ rules}]. & (7)\\
\quad \ldots & & & \\
\mathit{end(Category)} & & & (8)
\end{array}
$$

Figure 4: Syntax of the LAX-dictionary

nication between semantic construction engine and semantic construction rules or action rules. The variables that can be used are shown in Table 1.

The special state transition table named *end*, for which the system provides, indicates the end of word when the adjoining morphemes are connected by a connective feature which belongs to *end Stt*.

Declaration	In/out	Meaning
in(In)	in	get the previous semantic structure
no(No)	in	get the word number
sur(Sur)	in	get the surface string of the morpheme
cat(Cat)	in	get the category of the morpheme
tok(Tok)	in	get the token of the morpheme
out(Out)	out	put the semantics structure to the next morpheme

Table 1: Declaration of Variables

3.4 Configuration of the LAX System

Figure 5 shows the configuration of the LAX development system. The LAX-dictionary written in the description format as shown in Figure 4 is transformed into the intermediate files by the translator. Most of the tools provided by the LAX system operate through these intermediate files.

The LAX-dictionary can be transformed from the intermediate files by reverse-translator at any time. The aim of this function is to back up the intermediate files and confirm the current state of the morphological dictionary.

During the development of the morphological dictionary, the analysis engine accesses the intermediate files by way of an interpreter. We can therefore check the effect of modifying the dictionary immediately. This enables us to develop the dictionary effectively. It takes about ten times longer to analyze a sentence

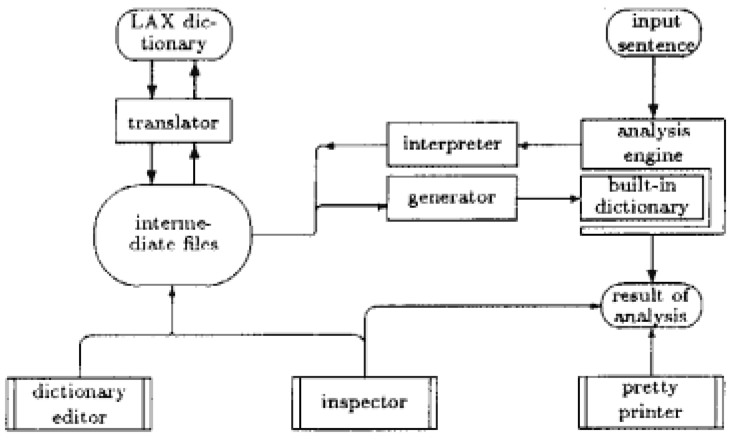


Figure 5: Developing Environment of LAX

using this interpreter with the built-in dictionary mentioned below. However, it is useful to debug the dictionary because we can get the result of the analysis in a few seconds.

After developing the dictionary, we can get a built-in dictionary and an action programs from the intermediate files in terms of a generator. The built-in dictionary is accessed by the analysis engine and an action program is called by the semantic construction engine.

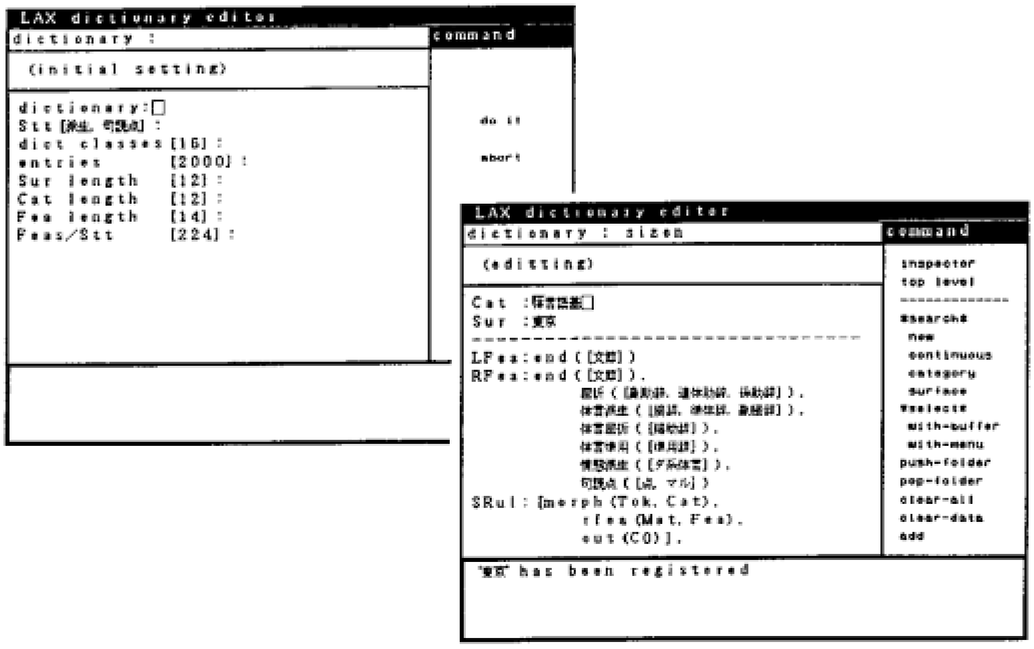
3.5 Development Tools

As development tools, we have a top-level window, a dictionary editor, an inspector and a pretty printer. Each of them is explained below.

3.5.1 Top-Level Window

When the LAX system is activated, the top-level window emerges first. Using this window, we can activate such tools as dictionary editor and inspector, transform the LAX-dictionary into the intermediate files, generate a built-in dictionary from the intermediate files, and so on.

Because the LAX system can work with multiple morphological dictionaries, we select the target dictionary in this window. We also designate what sort of dictionary (built-in or intermediate files) we use in order to access the target dictionary. These two analysis modes are called *compiled mode* and *interpretive mode*.



```
LAX dictionary editor
dictionary :                                command

 (initial setting)

dictionary:□                                  do it
Stt [終止, 句読点] :
dict classes [16] :                           abort
entries       [2000] :
Sur length    [12] :
Cat length    [12] :
Fea length    [14] :
Feas/Stt      [224] :
```

```
LAX dictionary editor
dictionary : sizen                          command

 (editing)                                   inspector
                                             top level
Cat  :普通名詞□                              -------------
Sur  :東京                                   #search#
-----------------------------------           new
LFea:end([文節])                              continuous
RFea:end([文節]),                             category
          [illegible] ( [格助詞, 連体助詞, 係助詞] ),     surface
          体言派生 ( [接辞, [illegible], [illegible]] ),  #select#
          体言[illegible] ( [[illegible]] ),            with-buffer
          体言修飾 ( [修用詞] ),                   with-menu
          情報派生 ( [ダ系体言] ),                push-folder
          句読点 ( [読, マル] )                    pop-folder
SRul: [morph(Tok, Cat),                      clear-all
          rfea(Mat, Fea),                    clear-data
          out(C0)],                          add

'東京' has been registered
```

Figure 6: Dictionary Editor

3.5.2 Dictionary Editor

The dictionary editor works with the intermediate files and provides the function of adding, searching, revising and deleting for each entry. When we invoke the dictionary editor as *new use*, the initial setting up window emerges, as shown on the left of Figure 6, in which we set a value for each entry: the dictionary name, state transition table names, maximum number of characters in the lexical entries, division number of the built-in dictionary, and so on.

In this mode, the intermediate files are created immediately by the dictionary editor instead of being transformed from the LAX-dictionary. The entry not assigned any value is assigned to a default value held by brackets. After setting up the entries, we move to the edit window shown on the right of Figure 6 as we do after invoking the dictionary editor as *continuous use*.

There are two ways to go to this edit window. One is, as mentioned above, from the top-level window; the other is from the inspector. The latter will be described in the next section.

Except in the case of going to the edit window from the inspector, there are no entries on the edit screen at first. To add a new morpheme to the dictionary, we simply fill each column and invoke an addition command.

In searching morphemes, we can use two sorts of keys: the surface characters of the morpheme and the morphological category. In both cases, the searched entries are assembled in a *folder* and the first entry in the folder is displayed. Other entries can be inspected in order, and for each of them we can apply such commands as deletion and revision. Replacing the surface characters of a searched morpheme enables us to make easy registration of another morpheme.

```
                          Tokyo  e   itte  morau .
input sentence      :  東京   へ  行って もらう 。
result of analysis :  東京・へ
                          to Tokyo
                          行・って・もら・う・。 / 行・って
                          have (someone) go           go and
                          もら・う・。
                          receive (something)
interpretations     :  (1)  東京・へ/ 行・って・もら・う・。
                               (Someone)  has (another) go to Tokyo.
                          (2)  東京・へ/行・って/もら・う・。
                               (Someone) goes to Tokyo and receives (something).
```

Figure 7: The Morphological Analysis

3.5.3 Inspector

The inspector is called from both the top-level window and the dictionary editor, and has two internal modes called *analysis mode* and *inspect mode*. The windows that correspond to each mode are shown in Figure 8. Note that these two windows never emerge at the same time.

Initially, the analysis mode is selected, in which we can get the result of the morphological analysis and the analysis state (success/failure), number of possible interpretations, number of character of the input sentence and analysis time. If analysis fails, we can also get information about the position and surface character of some unknown or unconnectable morphemes.

The under window in Figure 8 shows the result of the morphological analysis for the input sentence which is typed in the upper input window using the input sentence command. As for the result, each line corresponds to one word which consists of several morphemes separated by dots. If there are some ambiguities, alternatives are displayed on the same line, with each word separated by slashes as shown in Figure 7.

In this case, there are two interpretations for the input sentence. The first one consists of two words and the second one has three words. Note that the input sentence in Figure 7 is separated by blanks because of the explanation; the actual input sentence has no blanks.

Changing the display mode to *detail* or *more detail*, we can get more detailed information for each morpheme such as morphological category or connective state transition table and connective feature.

If analysis fails, that is, if there are morphemes that can not be concatenated with any adjoining morphemes, the morpheme is marked with an asterisk so that we can see that it should be modified.

In order to input a sentence to LAX, we can use the sentence select command that enable us to select the input sentence in a menu form. The sentences should be written in a certain file in advance.

In the under window shown in Figure 8, that is in the analysis mode, mouse clicking on a certain character written in the upper input window changes the mode into the inspect mode.

The over window in Figure 8 shows the inspector in the inspect mode. In this example, on the seventh character 「も」 in 「東京へ行ってもらう。」 the mouse was clicked. Then, possible morphological entries looked up from that character are displayed in the middle right window. In the middle left window, morphological entries that precede the selected character are indicated. These morphemes indicated in the middle left window can be concatenated with the morphemes indicated in the middle right window.

In this state, clicking the left button of the mouse on a morpheme shows the description of selected morpheme, which is indicated in just the lower window. Double left mouse click on a morpheme shows the candidates that can be connected with the selected morpheme with their head numbering in some order. Among the candidates, the definition of the morpheme with an asterisk mark is displayed in the lower window. Using these functions, we can find the errors of the morphological definition easily and quickly.

Moreover, a middle mouse click on a morpheme sends that morpheme's information to the *folder* of the dictionary editor. Double middle mouse click on a morpheme also sends its information to the folder and activate the dictionary editor continuously. Activated in this way, the dictionary editor displays the first entry of the folder and awaits the command. We can therefore revise the definition of necessary morphemes and return to the inspector immediately to confirm the result of the analysis.

3.5.4 Pretty Printer

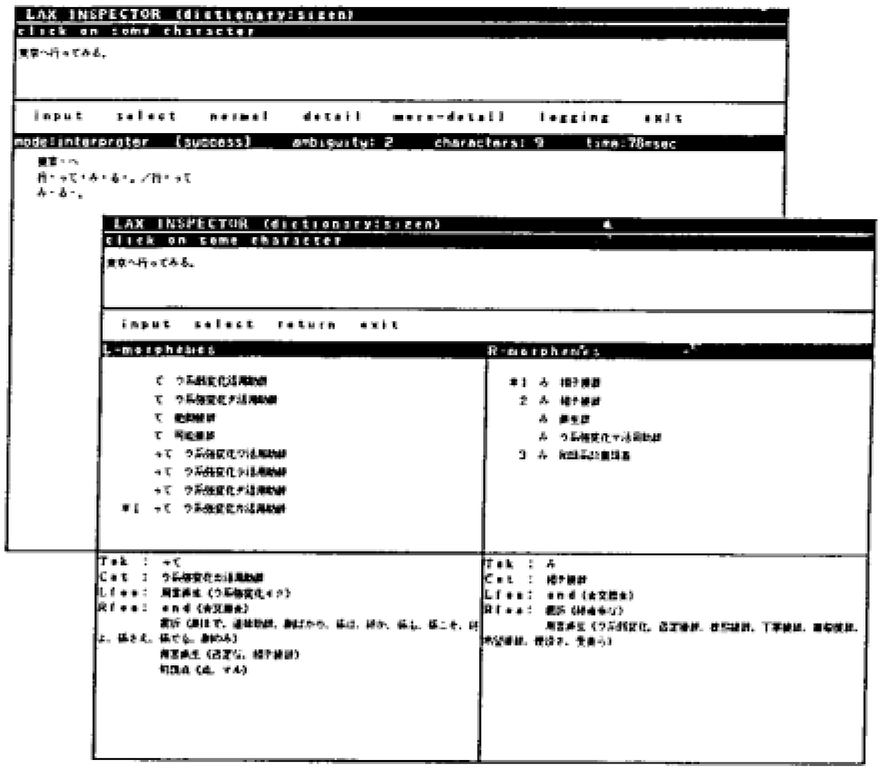
The pretty printer analyzes an input sentence and shows the ambiguities of the result of analysis clearly. The sentence is input to the pretty printer in the same way as to the inspector. When the result is forced out of the window, we can move it and look at any part of the result freely.

3.6 Further Research

Currently, the debugging environment for the semantic construction module is poor compared with that for the morphological analysis module. So an effective debugging tool for the semantic construction module should be developed.

Implementation of a lexical acquisition tool such as VEX system [10] is also under consideration.

In addition to developing these tools, we have been developing a morphological dictionary consisting of about 2,000 entries as part of the language data base. We are planning to refine the description of the dictionary and increase the entries up to 60,000 morphemes.



```
LAX INSPECTOR (dictionary:sizen)
click on some character
東京へ行ってみる。

  input   select   normal   detail   more-detail   logging   exit
mode:interpreter   (success)   ambiguity: 2   characters: 9   time:78msec
   東京・へ
   行・って・み・る・。/行・って
   み・る・。
```

```
LAX INSPECTOR (dictionary:sizen)
click on some character
東京へ行ってみる。

  input   select   return   exit
L-morphemes                              R-morphemes
      て  [illegible]                         *1  み  [illegible]
      て  [illegible]                          2  み  [illegible]
      て  [illegible]                             み  [illegible]
      て  [illegible]                             み  [illegible]
      って  [illegible]                        3  み  [illegible]
      って  [illegible]
      って  [illegible]
 *1   って  [illegible]
Tok  :  って                              Tok  :  み
Cat  :  [illegible]                       Cat  :  [illegible]
Lfea:  [illegible]                        Lfea:  end [illegible]
Rfea:  end [illegible]                    Rfea:  [illegible]
       [illegible]                               [illegible]
       [illegible]
       [illegible]
```

Figure 8: Inspector

4 Syntax Analyzer/*SAX*

In this section, we introduce the SAX (Sequential Analyzer for syntaX and semantics) system which is a module of the Language Tool Box (LTB) for syntactic and semantic analysis. First we describe an outline of the SAX including the parsing method and configuration of the system. Then we explain its grammar rules, and at the end of this section, we describe the debugging tools which provide a development environment for the user.

4.1 The SAX System

SAX is a syntactic and semantic analyzer based on logic programming. Unlike the bottom-up, depth-first processing regime of the BUP parsing algorithm, SAX employs a bottom-up and breadth-first parsing algorithm [11]. SAX parser was first implemented on DEC10 Prolog at ICOT in 1986 and reimplemented on PSI (Personal Sequential Inference Machine) the next year. Subsequently , some debugging tools were developed on PSI. PAX (Parallel Analyzer for syntaX and semantics), which is the parallel version of SAX, was also implemented on M-PSI (Multi Personal Sequential Inference Machine). SAX system provides with the user a facility to develop a grammar, and is composed of submodules, translator, and debugging environment.

Figure 9 shows the configuration of the SAX system. Grammar rules are translated into a parsing program written in ESP. The parsing program receives the result of the LAX (lexical analyzer for syntactic and semantic) system [12] or a sequence of words input from keyboard, and parses it at high speed. As the result of parsing, we obtain the semantic representation as well as a parse tree. If the result has some errors, we can use the debugging tools provided by the SAX system, such as the grammar debugger, and graphic display utility, and correct the grammar rules.

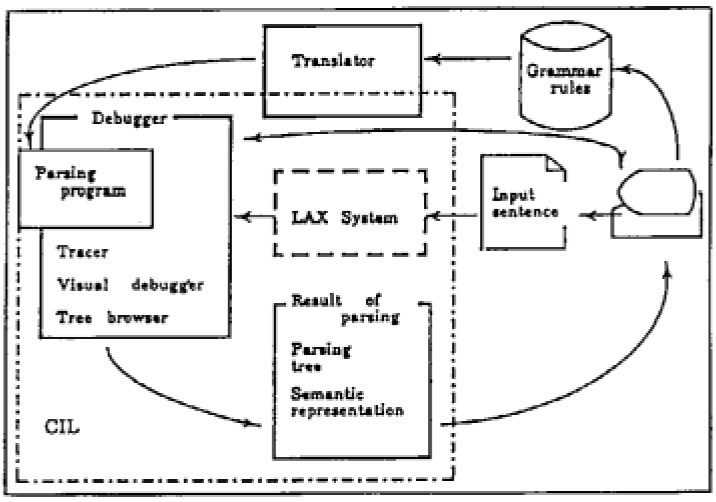


Figure 9: Configuration of the SAX system

4.2 Parsing Method

The SAX parsing algorithm is bottom-up parsing with top-down prediction. In other words, parsing proceeds by composing more parse trees based on the already recognized trees by applying grammar rules. If there are some applicable grammar rules, all of them are executed at the same time. That is to say, its parsing algorithm was devised for parallel parsing. However, it also works efficiently in sequential implementation. The major advantage of our system is that

```
head  → body_1,{extra_1}          (1)
         :: {pref_rule_1},        (2)
         ...                      (3)
        body_n,{extra_n}          (4)
         :: {pref_rule_n},        (5)
        &{delayed_extra},         (6)
        &&{pref_rule}.            (7)
```

Figure 10: SAX grammar rules label'figure2'

the parsing process proceeds from the bottom up: therefore, the left-recursive rules have no problems, and the parsing process does not involve backtracking, which means that there is no redundant construction of syntactic structures.

4.3 SAX Grammar Rules

The SAX grammar rule [13] is basically an extension of DCG (Definite Clause Grammar) [2] [14] and is executed parsing bottom-up and breadth-first with some restriction. By using these extensions, we can analyze more effectively.

In the above grammar, **head** in line (1) and **body_i**'s in line (1) and (4) represent grammatical categories and are represented in the form of Prolog terms. **extra_i**'s in line (1) and (4) are extra conditions on the rule, expressed as a Prolog goals. We can write the extra condition enclosed in ' {' and '} ' after **body_i**.

1. Delayed Extra Condition

   In our system, in addition to the extra conditions, a delayed extra condition (line (6)) can be specified. A delayed extra condition is enclosed in braces ' {} ' and prefixed by ' & ' as in the line (6). Basically, extra conditions are used to restrict unsuitable parsing trees, but delayed extra conditions are used to compose semantic structure. The Prolog goals in the delayed extra conditions are evaluated after parsing.

2. Preference Calculation

   Disambiguation of sentence interpretations is one of the hardest problems in natural language processing. Therefore, many approaches have been investigated, such as the solution of disambiguation using discourse structure [15]. The Preference rules in lines (2),(5),and (7) enable us to calculate one of the plausible results using local information of the parsing process. Preference rules are enclosed in braces ' {} ' and prefixed by '

[2]As in DCG, cyclic rules cannot be written in the SAX grammar rule.

&& ' or ' : : '. Moreover, our system supplies information related to the calculation of lexical preference as follows.

- In '::'-prefixed pref_rule_i

  Pref_cat ··· In **pref_rule_i**, preference for **body_i** in the rule.

  Pref ··· In **pref_rule_i**,preference for **body_i** in the rule which is calculated in the rule whose **head** is **body**.

- In '&&'-prefixed pref_rule

  pref_CAT ··· Preference for the head. It can give after successing this rule.

  prefs(i) ··· Preference in **pref_rule_i**.

4.4 Debugging Environment

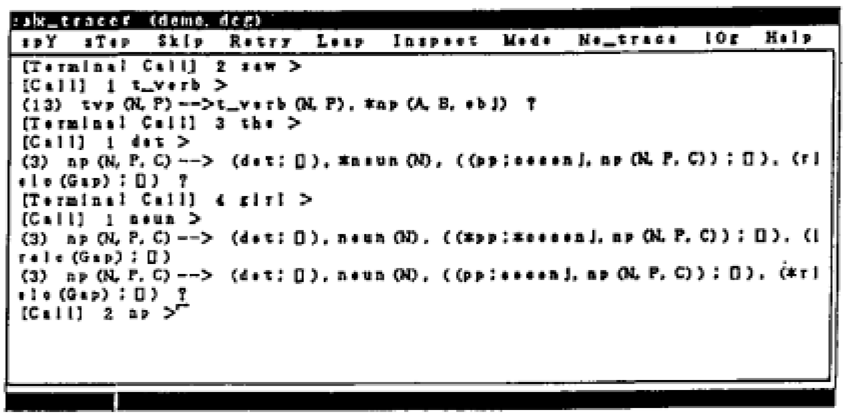
The SAX system provides grammar debugging tools in order to develop grammar effectively. The SAX system provides two types of debugging environments, the static debugging environment and the dynamic debugging environment [16]. The dynamic debugging environment such as tracer is used to locate the fail point of the parser during its execution. On the other hand, in the static debugging environment, the visual debugger or tree browser is used to find the fail points of extra conditions or the delayed extra conditions after the parse has been done.

4.4.1 Tracer

The tracer, dynamic debugging environment, gives the information on the parsing process. Figure 11 shows a display of the execution of tracer, and ' ✻ ' shows the parse points currently being applied. As with the Prolog tracer, the user can specify the behavior of the parsing process by selecting commands such as 'step', 'skip', and 'leap' from the menu. When we evaluate extra conditions which were written in CIL program, the CIL debugger starts, and tracing is performed.

4.4.2 Visual Debugger

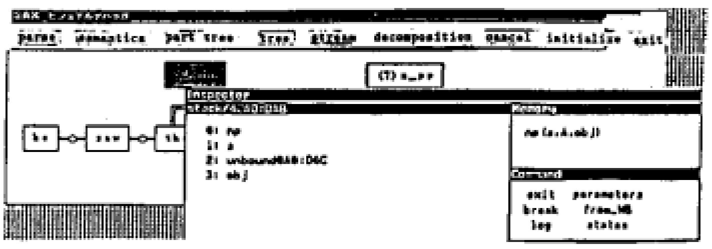
The visual debugger, static debugging environment, shows the information on a partial parsing using the results of a parse. Figure 12 is a display of the visual debugger. After parsing a sentence, the user can inspect the resulting parsing trees by using the visual debugger. The visual debugger gives the user information on the partial parsing trees of the sub sequence of the input sentence, which is specified by the user. In Figure 12, the start



```
[Terminal Call] 2 saw >
[Call] 1 t_verb >
(13) tvp (N, P) -->t_verb (N, P), *np (A, B, obj) ?
[Terminal Call] 3 the >
[Call] 1 det >
(3) np (N, P, C) --> (det: []), *noun (N), ((pp;eeeen), np (N, P, C)) ; []), (rl
ele (Gap) ; []) ?
[Terminal Call] 4 girl >
[Call] 1 noun >
(3) np (N, P, C) --> (det: []), noun (N), ((*pp:*eeeen), np (N, P, C)) ; []), (l
rele (Gap) ; [])
(3) np (N, P, C) --> (det: []), noun (N), ((pp:eeeen), np (N, P, C)) ; []), (*rl
ele (Gap) ; []) ?
[Call] 2 np >
```

Figure 11: Tracer

point is 'saw' and the end point is 'man', the system looks up for the result of parsing sequence from 'saw' to 'man'. If the result of parsing succeeds, the system displays the obtained root category of the resultant trees. In Figure 12, the parsing succeeds, and root category 'tvp' is obtained. In this way, the user constructs the parsing trees one after another. However if the result of parsing fails, the user can narrow the candidates for being an erroneous rule. The visual debugger can also display the semantic information by clicking on a tree node, or display the part tree by clicking a mouse on the 'sub_tree' button in the window label. Figure 13 respectively.


parse semantics part tree tree string decomposition cancel initialize exit
(7) s_pp
Inspector
Memory
Command
he
saw
exit
parameters
break
from_MB
log
states

```
0: np
1: a
2: unboundVAR:DAC
3: obj
```

```
np (a,A,obj)
```

Figure 12: Visual debugger

4.4.3 Tree Browser

The tree browser shows the results of syntactic analysis, the number of trees constructed, time required for the parse, and the description of pref-

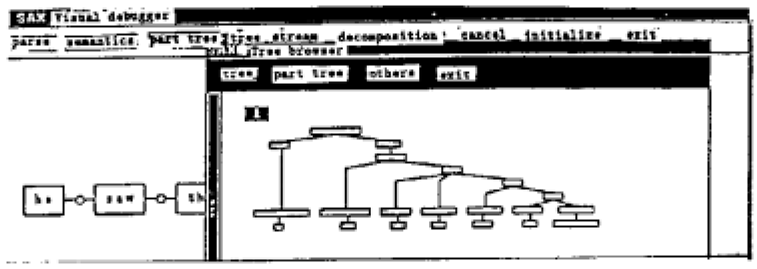
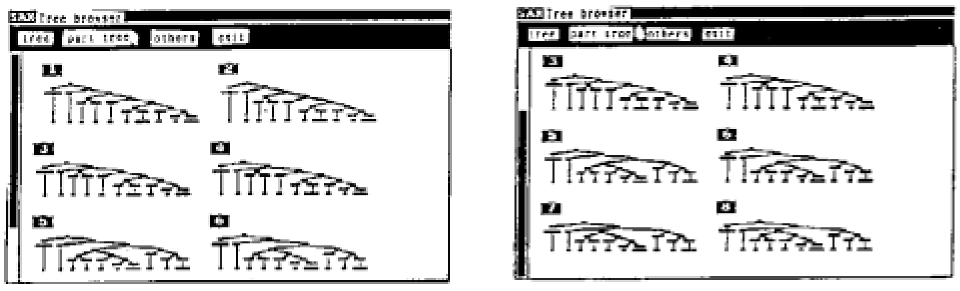


Figure 13: Part tree in Visual debugger



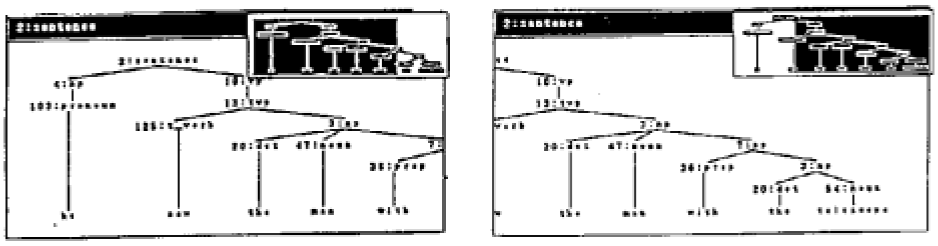
before scrolling after scrolling

Figure 14: Tree browser (menu window)

erence score. Figure 14 shows the menu indication of the reduction of the result of the parsing tree. The user select a parse tree from the menu and select 'tree' or 'part tree' from the menu. In Figure 14, if '2' and 'tree' are selected, the tree of 2 is displayed; as shown in Figure 15. If the tree is too big to display at one time on the window, the user can shift his viewpoint by selecting the scroll bar and can inspect or see the details of the parse tree.

4.5 Future Research

We have described the SAX system, its parsing method, the features of the grammar rules and debugging environment. In the future, we plan to improve the debugging environment, especially the tracer. We are to add a function to display the parse tree as it is being constructed by the parser. For the translator, we must consider the problem of variable restrictions. Another important aspect of future research is to design the High-level grammar description. Basically, the SAX grammar rule is an extension



before scrolling after scrolling

Figure 15: Tree browser (indication of graphic scope)

of DCG and allows only one head category to the left hand side of the rules. But it is proved that the SAX allows some head category to the left hand side of the rules [3] [17]. Using this capability, we could represent dependency grammar in addition to the phrase structure grammar, but there are as yet no debugging tools for this. We are to expand the SAX system accordingly.

5 Sentence Generator

This section describe the *sentence generator* of the LTB and its development tools.

5.1 Outline of the Sentence Generator

The sentence generator provides a function for generating a sentence from a given intermediate representation. The intermediate representation is a kind of frame structure written in PST, which was described in section 2.

One of the most important features of the sentence generator is that the major process of the sentence generation was implemented as a **macro expansion**. Speaking more concretely, the sentence generator, according to given **expansion rules**, expands each **macro expression** included in a given intermediate representation. Through this macro expansion, the input intermediate representation is rewritten into more specific structure. The reason why we adopted this method is that the sentence generator should be flexible enough to accept a variety of structure as input. Considering that the sentence generator is used in various applications, this flexibility is important. Users can design the intermediate representation

[3] Unlike XG (Extraposition Grammars),current extension does not allow any rule of the form 'A,Skip, B → C,...,D.', for any sequence of words Skip can be displaced.

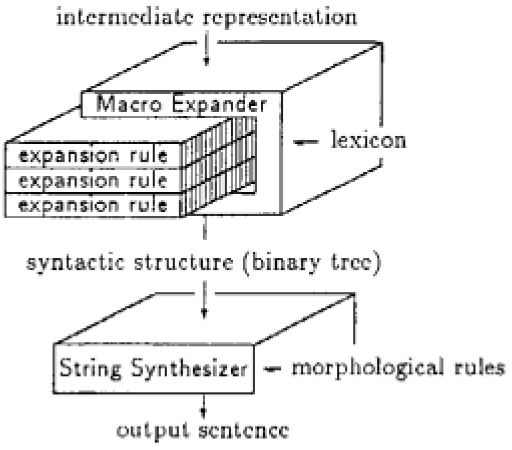


Figure 16: Configuration of **generation engine**

they want, by changing the expansion rules. The expansion rules can be seen as encoded linguistic knowledge.

As for expansion rules, a prototype of expansion rule set for Japanese, called **standard expansion rules** is currently available as one of the components of the sentence generation module, which is intended to serve as a basis for more improvements by each user.

Like other modules, some development tools are prepared, in the sentence generation module. These tools are introduced in 5.3, with their windows.

5.2 Flow of the Sentence Generation/Generation Engine

Sentence generation is performed by **generation engine**, which consists of two sub modules: **macro expander** and **string synthesizer**. The configuration of the generation engine is shown in Figure 16. Sentence generation is done in the following sequence:

(a) **macro expansion**

(b) **syntactic structure generation**

(c) **morphological processing**

2a. and 2b. are done by macro expander, and 2c. by string synthesizer.

```
{'REL'/ { 'LEX'/ 殴る (hit)},
 'ROLE'/ {'AGENT'/{'COMP'/{'LEX'/ 先生 (teacher) }},
          'OBJECT'/{'COMP'/{'LEX'/ 生徒 (student) }}}}
```

A teacher hits a student.

Figure 17: Example of Intermediate Expression with Macro

```
{head {lex/ 殴る (hit)},
 comp/{head/{head/{lex/ を (case marker for object)},
             comp/{lex/ 生徒 (student)},
       comp/{ head/{lex/ が (case marker for agent)},
              comp/{head/{lex/ 先生 (teacher) }}}}
```

A teacher hits a student.

Figure 18: Example of Primitive Expression

5.2.1 Macro Expansion

Macro expander expands each macro expression in a given intermediate representation, according to the corresponding expansion rule.

Each expansion rule is written like a *filter*, a filter which receives an intermediate representation, expands the macro expression which the rule takes charge of, and sends the result of the expansions. Passing through the filters, one after another, the given intermediate representation is transformed, step by step into more specific expression.

Let us give a example of macro expansion based on the **standard expansion rules**. The example of an intermediate representation with macro expression, and its fully expanded representation (called **primitive expression** – the result of macro expansion– are shown in Figures 17 and 18.

In the example of Figure 17, the expression

'AGENT'/{'COMP'/{'LEX'/ 先生 }}
is a macro expression for

```
comp/{head/{lex/ が },
        comp/{head/{lex/ 先生 }}}
```

in the Figure 18.

The primitive expression, as is known from the Figure 18, is in a form of binary tree of **head** and **complement**.

According to **JPSG (Japanese Phrase Structure Grammar)**[18] framework, every Japanese sentence is essentially generated by only one phrase structure rule, **head-complement** configuration. In other words, any syntactic structure of Japanese sentence can be represented as a binary tree of **head** and **complement**. We adopted this idea to represent a primitive expression.

In this expression, the values of lex, which are the leaves of the binary tree, are the lexical entries of each word in the dictionary. The postpositions and auxiliary verbs are designated explicitly with lex. The word order is also specified explicitly in **primitive expression**, because a complement immediately precedes its head, in Japanese.

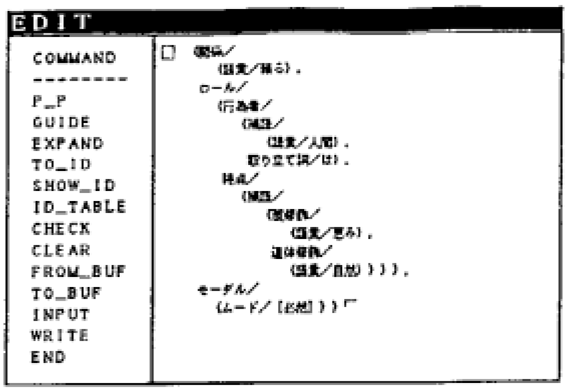
5.2.2 Syntactic Structure Generation

After completion of macro expansion, macro expander transforms the intermediate representation, (primitive expression) into a structure of nested list, called **syntactic structure**, which is similar to the intermediate representation. Each element of this list is a pair of a part of speech and lexical information of the word, which corresponds to the leaf of the tree in the binary tree of the primitive expression. The lexical information is taken from the lexicon (See Figure 16), and is referred in the next stage, **morphological processing**.

5.2.3 Morphological Processing

String Synthesizer perform morphological processing to a **syntactic structure**, by referring to lexical information of words in the **syntactic structure** and morphological rules. Morphological rules are described in the form of two tables: **connection table** and **inflection table**.

The rules on **connection table** describe the inflection form of an inflectional word which has some inflection type, when the directly modified word by that word has some lexical data. The directly modified word can be found from **head-complement relations** in that **syntactic structure**. The rules on **inflection table** describe the surface expressions of inflection forms about every inflection type.



```
□ (関係/
    (語彙/編む) ,
  ロール/
    (行為者/
      (概念/
        (語彙/人間) ,
      取り立て詞/は) ,
    対象/
      (概念/
        (限定格/
          (語彙/[illegible]) ,
        連体修飾/
          (語彙/自然) ) ) ,
  モーダル/
    (ムード/ [断定] ) ) 
```

Figure 19: The edit-window for intermediate expression

5.3 Development Tools

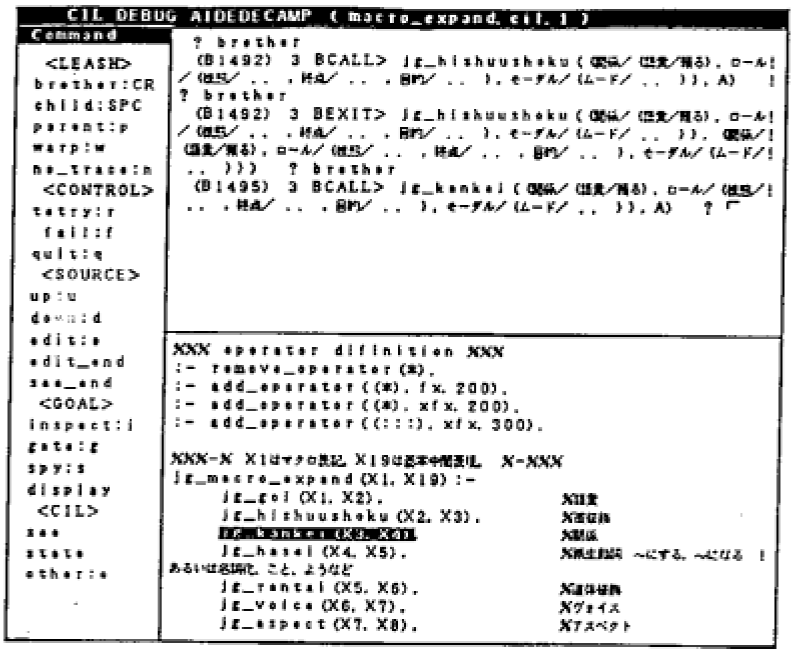
Sentence generator provides the following development tools,

Editors

Editor for intermediate representation The editor for intermediate representation has some useful additional functions: labeling to PST-structure, guide and syntax check for macro expression, and so on. The PST-structure labeled by labeling function, can be referred with this label in any intermediate representation. That is, labeled PST-structures can be used as common components of intermediate representation.

Editor for syntactic structure The editor for syntactic structure has graphic display function as an additional function. The edit-window for Intermediate representation is shown in figure 19.

Debugger Debugger of sentence generator is build on the CIL debugger. Using the function provided by CIL debugger, the user can consult or compile the generation rules, trace and control the execution of a sentence generation. The debug window is shown in Figure 20.



```
CIL DEBUG AIDEDECAMP ( macro_expand, cil, 1 )
Command
  <LEASH>
brother:CR
child:SPC
parent:p
warp:w
no_trace:n
  <CONTROL>
retry:r
  fail:f
quit:q
  <SOURCE>
up:u
down:d
edit:e
edit_end
see_end
  <GOAL>
inspect:i
gate:g
spy:s
display
  <CIL>
see
state
other:o
```

```
 ? brother
  (B1492)  3 BCALL> jg_hishuushoku ( 関係/ (話題/有る), ロール!
/ (格助/  ..  , 持点/  ..  , 目的/  ..  ), モーダル/ (ムード/  ..  )), A)   !
? brother
  (B1492)  3 BEXIT> jg_hishuushoku ( 関係/ (話題/有る), ロール!
/ (格助/  ..  , 持点/  ..  , 目的/  ..  ), モーダル/ (ムード/  ..  )),  関係/!
(話題/有る), ロール/ (格助/  ..  , 持点/  ..  , 目的/  ..  ), モーダル/ (ムード/!
 ..  )))   ? brother
  (B1495)  3 BCALL> jg_kankei ( 関係/ (話題/有る), ロール/ (格助/!
 ..  , 持点/  ..  , 目的/  ..  ), モーダル/ (ムード/  ..  )), A)   ? ┌
```

```
%%% operater difinition %%%
:- remove_operater(*).
:- add_operater((*), fx, 200).
:- add_operater((*), xfx, 200).
:- add_operater((:::), xfx, 300).

%%%-% X1はマクロ表記, X19は基本中間表現  %-%%%
jg_macro_expand(X1, X19) :-
      jg_goi(X1, X2).                                 %語彙
      jg_hishuushoku(X2, X3).                         %被修飾
      jg_kankei(X3, X4).                              %関係
      jg_hasei(X4, X5).                               %派生動詞 ~にする, ~になる !
あるいは名詞化, こと, ようなど
      jg_rentai(X5, X6).                              %連体修飾
      jg_voice(X6, X7).                               %ヴォイス
      jg_aspect(X7, X8).                              %アスペクト
```

Figure 20: The debug window

5.4 Further Research

As a general purpose tool, it is desirable for LTB sentence generator to allow users to modify these rules easily, to allow users to define the syntax of intermediate representation as they want. For this purpose, we have to define the declarative semantics on macro expansion rules.

6 LTB-Shell

LTB–shell is a new integrated user interface of the LTB. It is now under development, so we will just give an outline of the LTB–shell in this section.

6.1 Purpose of LTB–shell

Each LTB tool, that is LAX, SAX, Generator, CIL, and DataBase has it own user interface, but it lacks an integrated user interface, which gives the user an integrated method to invoke each tool and a method to combine a number of tools into one (Figure 21).

The user using only one LTB tool will not have much problem with the current LTB user interface. But the user using several LTB tools at once will have trouble using those tools together.

LTB–shell is a user interface aimed to help those users by an integrated method of tool invocations and tool combinations

(Figure 22).

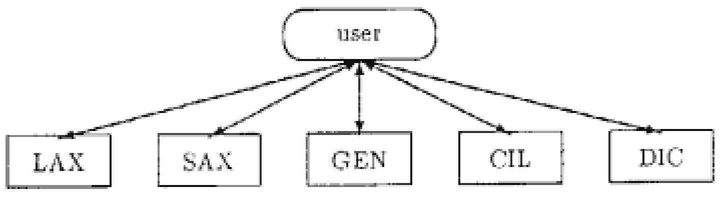


Figure 21: current user interface of LTB

6.2 User Interface of LTB–shell

LTB–shell provides two kinds of user interface. One is a command language. With command language interface, the user gives commands to the shell by typing in the command language. The shell, in turn, calls upon the LTB tools. The other is a graphical user interface. With this user interface, the user gives commands by selecting icons or selecting menus in shell window by a mouse.

In this paper we will describe some features of LTB–shells based on command language user interface.

6.3 Features of LTB–shell

LTB–shell has the following features.

- **Command invocation:** Each LTB tool is invoked as a single command followed by number of options. Options are used to specify detail tasks of each command. Each LTB tool is executed as a process different from shell process. Commands invoked by the user which correspond to the LTB tool do not always create a new process, as the creation of the process causes an overhead. Instead, commands reuse the corresponding process if there is one, or create a corresponding process if there is none.

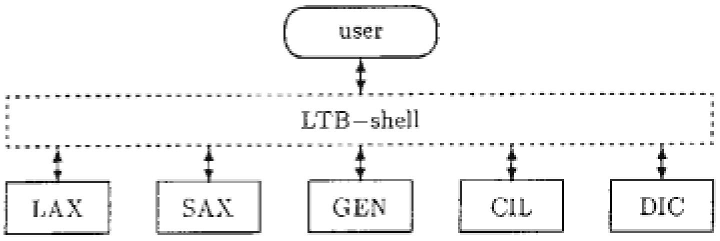


Figure 22: user interface using LTB−shell

```
LTB> lax −g grammar1 -input "....."
```

The above example shows a sample session between user and LTB−shell. "LTB> " is a prompt of LTB−shell. *lax* is a command name. *−g grammar*1 is an option and its argument. This input invokes lexical analysis over input sentence "....." using a grammar object *grammar*1. As described earlier, this command execution creates a new LAX process only if the process does not exist. A process management, that is a creation, termination, and registration of the process, is done mostly by LTB−shell, though some methods for process management are open to the user.
There may be some tasks that cannot be expressed directly by the command−options format. In such cases the user should invoke the ordinary LTB tools from the LTB−shell, and indicate the tasks with the tool user interface.

- **Redirection:** The user can specify the input sources and output destinations of a command. The idea is called **redirection** and is used in many shells of operating systems [19].
  In LTB−shell, each command should take data from standard-input and send data to standard-output. *command* } *file*1 implies that any output of the command to the standard-output is sent to the file file1. The user can specify an window or buffer instead of the file. *command* { *file*2 implies that the command gets a data from file file2. The user can also specify a window or buffer instead of the file.

  ```
  LTB> lax { file1 } widow1
  ```

  In this example an input to lax command comes from file file1, and output from the command goes into window.
- **Pipes:** The LTB−shell will allow the user to send an output of one command to an input of another command. This idea is called **pipes**. Pipes are also used in many shells of operating systems.

command1|*command2* implies that an output from the command1 is sent to the command2. Pipes and redirections can be used in combination.

```
LTB> lax  {  file1  |  sax
```

In this example an input to lax command comes from file1, and output from the command goes into the next sax command.

- **Job control:** A sequence of commands which makes up an input to the LTB– shell is called, **jobs**. Each job is executed either as foreground job or background job. Foreground job is a job running on shell window which occupies the window. So the output to the standard-output which is not redirected will be printed on the shell window. On the other hand, background job cannot write to the shell window. When any of the commands in the job tries to write to the shell window it will be suspended.

  To run a command in background, one put "&" at the tail of the job. A number of background jobs can be run in parallel, while only one foreground job can exist. So users should run jobs in background if they want to run them in parallel.

  ```
  LTB> lax –g  grammar1  |  sax  }  file1 &
  LTB> lax –g  grammar1  |  sax   &
  LTB> lax –g  grammar1  |  sax  }  file3
  ```

  In the above example, the first two jobs are run in background jobs, and the last job is run in foreground. The second job will suspend when the second command sax tries to write it output to the shell window. Shell commands for managing jobs, that is making jobs fore(back)ground, killing jobs, and suspending jobs, are open to the user.

- **other features:** LTB–shell will also support **history** feature and **alias** feature.

  The shell command is not restricted to LTB tool commands. Commands usually used on SIMPOS shell will also be available on LTB–shell.

7 Conclusions

In this paper, we introduce LTB, mainly presenting currently available tools provided by it.

The first version of LTB was only recently released with manuals. From now on, we should improve it reflecting many user's comments.

Finally, we hope many NLP applications will be built on LTB.

Acknowledgements

Many programmers cooperated with us in the design and coding of LTB. First of all, we would like to thank them for their efforts. We would also like to thank Dr. Uchida, the chief of the second laboratory, and Dr. Yoshioka, the assistant chief of the second laboratory, for their encouragement and support.

References

[1] H. Nishikawa et al. The personal inference machine (PSI): Its design philosophy and machine architecture. Technical Report 13, ICOT, 1983.

[2] R. Sugimura et al. *Natural Language Processing in the Experimental Discourse Understanding System*. ELLIS HORWOOD, 1989.

[3] T. Chikayama. ESP reference manual. Technical Report 044, ICOT, 1984.

[4] K. Mukai and H. Yasukawa. Complex indeterminates in Prolog and its application to discourse models. *New Generation Computing*, 3, 1985.

[5] K. Mukai. A system of logic programming for linguistic analysis based on situation semantics. In *workshop on semantic issues in human and computer languages*. CSLI, 1988.

[6] A. Colmerauer. Prolog-II: Reference manual and theoretical model. *International report, Gourp Intelligence Artificialle, Universite d'Aix-Marseille II*, 1984.

[7] R.Sugimura, Y.Kubo, and Y.Matsumoto. *Logic Based Lexical Analyzer LAX*. 1989.

[8] A.V.Aho, J.E.Hopcroft, and J.D.Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, 1983.

[9] K. Morioka. *Goi no Keisei (Formation of a vocabulary* in Japanese*)*, volume 1 of *Gendai-go Kenkyuu*. Meiji Shoin, 1987.

[10] H.Alshawi, D.M.Carter, J.van Eijick, R.C.Moor, D.B.Morgan, and S.G.Pulman. Overview of the Core Language Engine. In *Proceedings of The International Conference of Fifth Generation Computer Systems*, pages 1108–1115. ICOT, 1988.

[11] Y. Matsumoto and R. Sugimura. A parsing system based on logic programming. In *Proceedings of IJCAI 87*, 1987.

[12] R. Sugimura, K. Akasaka, Y. Kubo, H. Sano, and Y. Matsumoto. Ronri-gata keitaiso kaiseki LAX (logic based lexical analyzer lax *in japanese*). In *Proceedings of the Logic Programming Conference '88*, pages 213–222. ICOT, 1988. English version will be appeared in The Lecture Notes on Computer Science.

[13] Y. Matsumoto and R. Sugimura. Koubun kaiseki system SAX no tameno bunpô kijutu gengo (grammar description language for the sax parsing system *in japanese*). In *5th Conference Proceedings of Japan Society for Software Science and Technology*, pages 77–80, Tokyo, 1988.

[14] F.C.N. Pereira and D.H.D Warren. Definite clause grammars for language analysis- a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13, 1980.

[15] R. Sugimura. Ronri-gata bunpô ni okeru seiyaku kaiseki (constraint analysis on logic grammars) (*in japanese*). In *Proceedings of the 2nd Annual Conference of Japanese society for Artificial Intelligence*, pages 427–430. Japanese society for Artificial Intelligence, 1988. a prized paper.

[16] R. Sugimura, K. Hasida, K. Akasaka, Y. Kubo, K. Hatano, T. Okunishi, and T. Takizuka. A software environment for research into discourse understanding systems. In *FGCS'88*, pages 285–295. ICOT, 1988.

[17] R. Sugimura. *layered stream wo motiita nihongo kaisekisyori (Japanese analysis based on layered stream) (*in Japanese*)*. ICOT TR. ICOT, 1989.

[18] T. Gunji. *Japanese Phrase Structure Grammar*. Dordrecht D. Reidel, 1987.

[19] SIMPOS user's manal. Technical report, ICOT, 1989.