〔非公開〕

TR-C-0010

構文解析ツール　ＰＡＲＳＥＲ

高橋　友一　　島　健一
TOMOICHI TAKAHASHI　KEN'ICHI SHIMA

木下　茂行　　横田　政憲　　中島　俊介
SHIGEYUKI KINOSHITA　MASANORI YOKOTA　SYUNSUKE NAKAJIMA

１９８８．４．１４．

ＡＴＲ通信システム研究所

構文解析ツール：

PARSER

木下茂行　横田政憲　中島俊介　島　健一　高橋友一

1988.4
ＡＴＲ通信システム研究所

序章：

自然言語処理技術は、人工知能に関係するドメインを研究対象とする者にとって、それ自体、大きな研究対象であるほか、インタフェースを研究対象とする者にとっては、基盤技術の一つである。『人間主体の通信システム：Human Oriented Communication System （ＨＯＣＳ）を目指して』を統一テーマとしているＡＴＲ通信システム研究所における

・通信ソフトウェアの自動作成にける、文書からの知識の抽出。

・ユーザフレンドリー通信における言語情報の処理。

等、

の研究テーマにおいて、入力テキストを解釈する構文解析機能は、後者の観点から必須のツールの一つである。

以上の観点から、

・走行環境

Ｃｏｍｍｏｎ　Ｌｉｓｐ上。

（ＶＡＸ，ＳＵＮ，Ｓｙｍｂｏｌｉｃｓ上では実績有り。）

・拡張文脈自由文法を用い、格フレームが作成できるようにＬＩＳＰプログラムが文法中に記述出来る。

・オンラインでもオフラインでも利用出来る。

等の特徴がある構文解析ツール（ＰＡＲＳＥＲ）を作成した。

今後、作成した構文解析ツール（ＰＡＲＳＥＲ）を、文書からの知識の抽出、自然言語を用いたインタフェースの研究に活用していく。

最後に、本ツールを試作する機会を与えて頂いたＡＴＲ通信システム研究所　山下社長、知能処理研究室　小林室長、通信ソフトウェア研究室　門田室長に感謝します。

Ⅰ章. 機能仕様

1．文法

1．1　文法規則の書式

拡張されたＣＦＧ（Context Free Grammer）であり、右辺が空である規則（ε規則）、右辺に繰り返しのある規則を記述することができる。記述方式は以下の通りである。

```
<grammer>  ::= (<lhs> . <rhs>)
<lhs>      ::= (<cat> <struct> <check>)
<rhs>      ::= nil |                          ;;;  ε規則
               ( <term> {<term>}* )
<term>     ::= <term1> |
               ( <term1> . <continue> )       ;;;  繰り返し規則
<term1>    ::= ( <cat> <var> <check> )
<continue> ::= ( nil ) |
               ( <count> <var> <check> )
<cat>      ::= string                         ;;;  文法カテゴリー
<struct>   ::= lisp object                    ;;;  構造作成
<check>    ::= lisp object                    ;;;  チェック
<var>      ::= symbol                         ;;;  下からの構造を受ける変数
<count>    ::= (number) |                     ;;;  number 以上の繰り返し
               (number1 number2)              ;;;  number1 から number2
                                              ;;;  までの繰返し
```

文法は構文解析（3．3参照）で参照される。その際の動作の概要を以下に示す。
<term> の <var> は下からの構造を受け <check> の <var> が置き換えられ評価される。最後に右辺の <check> を左辺の <var> で置き換えて評価される。それらの結果のうち1つでも nil となればその文法は適用されず、以降の <check> の評価は行なわれない。全ての結果が non nil の場合に限りこの規則の適用を行ない、右辺の <struct> の評価した結果を構造データとして上に送る。

１．２　文法の登録

defgram <lhs> . <rhs>　　［マクロ］

マクロ defgram により文法を登録する。
また、トップダウン予測情報は、defgram 実行時に大域変数 *link-category-list* (リスト) に("下の文法カテゴリー" "上の文法カテゴリー")の形態で登録される。

－－例－－

```
(defgram ("名詞句" (cons 'agent n) t)      ;;;  通常の規則
         ("名詞" n t)
         ("助詞" p (equal 'ga (car p))))

(defgram ("修飾語句" (list np2) t)         ;;;  繰返し規則
         (("名詞句" np1 t) (2 4) np2 t))   ;;;  ２から４回の繰返し

(defgram ("修飾語句" (list np2) t)
         (("名詞句" np1 t) (2) np2 t))     ;;;  ２回以上の繰返し

(defgram ("名詞句"  n t)
         ("名詞" n t)
         (("助詞" p t) nil))               ;;;  ０または１回の繰返し

(defgram ("名詞句" (eps) t))               ;;;  ε規則
```

１．３　文法登録時のシンタックスチェック

構文解析実行時のエラーを防止するために、文法登録時にシンタックスチェックを行う。

以下の場合は、適切な warning を出力し登録しない。

（１）通常の文法の場合

①右辺

・<term1> のレングスが３でない。
・<cat> がストリングでない。
・<var> がシンボルでない。
・一連の <term1> 中に <var> が重複して現れる。
・<check> に <var> でない未束縛のシンボルが使用されている。ただしマクロとスペシャルフォームの引数についてはこの限りでない。

②左辺

・<term1> のレングスが３でない。
・<cat> がストリングでない。
・<struct> <check> に右辺の <var> 以外の未束縛のシンボルが使用されている。ただしマクロとスペシャルフォームの引数についてはこの限りでない。

（２）繰り返し規則の場合

①　<continue> が (nil) の繰返しの <term>

a．右辺
・<term1> について（１）の右辺と同じチェック
・<continue> が (nil) でない。

b．左辺
・（１）と同じ

②　<continue> が (nil) でない繰返しの <term>

a．右辺
・<term1> について（１）の右辺と同じチェック
・<continue> のレングスが３でない。
・<continue> の <var> がシンボルでない。
・<continue> の <check> に <continue> の <var> 以外の未束縛のシンボルが使用されている。ただしマクロとスペシャルフォームの引数についてはこの限りでない。
(i) <count> が (number) の場合

・number が正の整数でない。
(ii)<count> が (number1 number2)の場合
(i) に加えて number1 number2 が昇順でない。

b．左辺
・(1)と同じ

(3) ε規則の場合
(1)に同じ

(4) 追加
<struct> <check> において、未定義の関数が含まれている場合は、warning を出力するが、文法ハッシュテーブルには登録する。

１．４　文法の持ち方

文法は、大域変数 *grammer*にハッシュテーブル（以下文法ハッシュテーブル）として登録される。

（１）　文法ハッシュテーブルの初期化

finit initialize-table　　［関数］

文法ハッシュテーブルは、関数 finit で初期化される。この関数は、文法ハッシュテーブルのだけではなく、辞書ハッシュテーブル（辞書データもハッシュテーブルで持つ、３参照）の初期化も行うことができる。

initialize-table は、ｇ ｄ ｂ のいずれかであり、ｇ であれば文法ハッシュテーブルを、ｄ であれば辞書ハッシュテーブルを、ｂ であればそれらの両方を初期化する。なお、ｇ または ｂ が指定された際は、トップダウン予測情報も初期化される。　　（付録２　参照）

－－例－－

```
(finit 'g)          ;;;  文法ハッシュテーブル初期化
(finit 'd)          ;;;  辞書ハッシュテーブル初期化
(finit 'b)          ;;;  文法・辞書ハッシュテーブル初期化
```

（２）　文法登録の際のハッシュ関数について

文法登録の際のハッシュ関数は右辺左端の文法カテゴリーを引数とし、ユーザが自由に定義できる。

gram-key cat　　［関数］

ハッシュ関数のデフォルトは、通常、右辺左端の文法カテゴリー（ストリング）であり、ε規則の場合 ""（空ストリング）である。

```
(defun gram-key (cat) (if cat cat ""))
```

－－参考－－

ハッシュ関数計算の結果同一キーになった場合、一つのキーに対して複数の文法が登録される。　　（付録１　参照）

１．５　文法の検索

find-gram cat　［関数］

文法ハッシュテーブルに対して文法の検索を行うには、関数 find-gram を用いる。

cat はストリングであり、右辺左端の文法カテゴリーに一致する文法全てのリストを返す。

なおε規則の場合は、２．３のハッシュ関数 gram-key の定義に依存する。デフォルトは "" （空ストリング）である。

１．６　文法ファイルのローディング

fload &rest data-files　［関数］

ファイルにまとめて記述された文法（以下このファイルを文法ファイルという）は、fload を用いてまとめてローディングできる。この関数は、文法ファイルだけではなく辞書ファイルのローディングもできる。その際、文法・辞書ハッシュテーブルを初期化する。

```
――例――
(fload "a.gram" "b.gram")
                ;;;  文法ファイル a.gram b.gram をローディング
(fload "a.gram" "b.gram" "a.dict")
                ;;;  文法・辞書ハッシュテーブルを初期化
```

2．辞書

2．1　辞書の書式

辞書には、語の活用の有無を記述できる。記述方式は以下の通りである。

```
<dictionary> ::= ( <lexs> <cats> <struct> <check> )
<lexs>       ::= <lex> |          ;;;  活用無し
                 ( <lex>          ;;;  活用有り
                  [<lvar>] )      ;;;  活用語尾を受ける変数
<cats>       ::= <cat> |          ;;;  品詞
                 ( <cat>          ;;;  活用有り
                   <app>          ;;;  活用の種類（カ行五段 etc）
                  [<avar>] )      ;;;  活用形を受ける変数
<struct>     ::= lisp object      ;;;  構造
<check>      ::= lisp object      ;;;  チェック
<lex>        ::= string           ;;;  見出し語
<cat>        ::= string           ;;;  品詞
<app>        ::= string
<lvar>       ::= symbol
<avar>       ::= symbol
```

辞書は形態素解析（3．2参照）で参照される。その際の動作を以下に示す。

（1）　活用がない語の場合

<check> が評価され、その結果が non nil であれば、<struct> を上に送る。

（2）　活用がある語の場合

活用語尾と、その活用形をそれぞれ <lvar> <avar> に束縛し、<check> を評価する。その結果が non nil であれば <struct> の <lvar> <avar> を置き換え構造データとして上に送る。

2．2　辞書の登録

defdict <lexs> <cats> <struct> <check>　　［マクロ］

関数 defdict により辞書を登録することができる。

――例――
(defdict "人" "名詞" (hito hum) t)　　;;;　活用がない場合
(defdict ("泣" n) ("動詞" "カ行五段" k) ((na n k) hum) t)
　　;;;　活用がある場合

2．3　辞書登録時のシンタックスチェック

形態素解析実行時のエラーを防止するために、辞書登録時にシンタックスチェックを行なう。
以下の場合は、適切な warning を出力し登録しない。

（1）　活用がない場合
・引数が４つでない。
・<lex> がストリングでない。
・<cat> がストリングでない。
（2）　活用がある場合
（1）に加えて
・引数が４つでない。
・<lvar>（活用語尾を受ける変数）が記述されている場合シンボルでない。
・<avar>（活用形を受ける変数）が記述されている場合シンボルでない。
・<app>（活用の種類）が活用テーブル（３．９参照）にない。
・<check> に <lvar> <avar> でない未束縛のシンボルが使用されている。ただしマクロとスペシャルフォームの引数についてはこの限りでない。

２．４　辞書のデータの持ち方

辞書は、大域変数 *dictionary* にハッシュテーブル（以下辞書ハッシュテーブル）として登録される。

（１）　辞書ハッシュテーブルの初期化
２．３　（１）の関数 finit で行う。
－－例－－
(finit 'd)　　;;;　辞書ハッシュテーブルのみ初期化

（２）　辞書登録の際のハッシュ関数について
辞書の登録する際のハッシュ関数は、見出し語の先頭の文字の内部コードを引数とし、ユーザが自由に設定できる。

dict-key inter-code　　［関数］

ハッシュ関数のデフォルトの定義は以下の通りである。

```
(defun dict-key (inter-code)
   (let ((jis-code (inter-to-jis inter-code)))
      (if (not jis-code)
          (error "Illegal code ~A" inter-code)
          (+ (/ (logand jis-code #x7F00) #x2)
                (logand jis-code #x7F))))
          ;;; hash(JIS) = ((JIS and x7F00) / x2) + (JIS and x7F)
```

－－参考－－
ハッシュ関数計算の結果同一キーになった場合、一つのキーに対して複数の文法が登録される。　　（付録１　参照）

２．５　辞書の検索

find-dict lex &key :hinshi :kouzou :check　　［関数］

辞書ハッシュテーブルに対して検索を行なう。
lex（ストリング）で指定された見出し語をもつ辞書情報すべてをリストとして返す。
キーワード :hinshi :kouzou :check で指定される引数は、それぞれ defdict の　cat struct check に対応し、指定された場合はそれらを含めてマッチする辞書情報すべてをリストとして返す。

２．６　辞書の削除

del-dict lex &key :hinshi :kouzou :check　　［関数］

辞書ハッシュテーブルに対して、削除を行なう。
lex で指定された見出し語をもつ辞書情報をハッシュテーブルから全て削除する。
キーワード :hinshi :kouzou :check で指定される引数は、それぞれ defdict の cat struct check に対応し、指定された場合はそれらを含めてマッチする情報を削除する。

２．７　辞書ファイルのローディング

２．６の fload で行なう。

```
――例――
(fload "a.dict" "b.dict")
          ;;;  辞書ファイル a.dict b.dict をローディング
(fload "a.dict" "b.dict" :initial 'd)
          ;;;  辞書ハッシュテーブルを初期化
```

２．８　辞書ハッシュテーブル修正の記録

fload 以外の手段で辞書ハッシュテーブルに対して行なわれた修正（defdict,del-dict）は、大域変数 *dict-log-file* に束縛されているファイル（辞書修正ログファイル）にその操作イメージが記録される。*dict-log-file* の初期値はカレントディレクトリの "dict.log" である。

```
――例――
(defdict "人" "名詞" (hito hum) t)
(defdict ("泣" n) ("動詞" "カ行五段" k) ((na n k) hum) t)
```

辞書修正ログファイル

```
(defdict "人" "名詞" (hito hum) t)
(defdict ("泣" n) ("動詞" "カ行五段" k) ((na n k) hum) t)
```

2．9　活用のある語を登録する際の制約

(1)　語幹のある語

活用のある語で語幹のあるものを登録する際は、語幹とその活用の種類を記述するだけで形態素解析（3．2参照）を行なうことができるが、その際、活用テーブルと整合性のある記述を行なわないと形態素解析に正しい動作を期待できない。

活用テーブルは大域変数 *katsuyou-table* に束縛し以下のような設定にする。

```
(defconstant *katsuyou-table*
  '(("動詞"
     ("カ行五段"
       ("未然" "か" "こ") ("連用" "き" "い").....
     ("ガ行五段"
       ("未然" "が" "ご") ("連用" "ぎ" "い").....
     ....................................
     ("ア行上一段"
       ("未然" "") ("連用" "") ("終止" "る").....
     ("カ行上一段"
       ("未然" "") ("連用" "") ("終止" "る").....
     ....................................
     ("ア行下一段"
       ("未然" "") ("連用" "") ("終止" "る").....
     ("カ行下一段"
       ("未然" "") ("連用" "") ("終止" "る").....
     ....................................
     ("サ行変格"
       ("未然" "さ" "せ" "し") ("連用" "し").....
     ....................................
    ("形容詞"
     ("シク"
       ("未然" "かろ") ("連用" "かっ" "く")
     ....................................
    ("形容動詞"
     ("ダ"
       ("未然" "だろ") ("連用" "だっ" "て")
     ....................................))))
```

扱える活用の種類は以下の通りである。

動詞
　カ行五段　ガ行五段　サ行五段　タ行五段　ナ行五段　バ行五段
　マ行五段　ラ行五段　ワ行五段
　ア行上一段　カ行上一段　ガ行上一段　サ行上一段　ザ行上一段　タ行上一段
　ナ行上一段　ハ行上一段　バ行上一段　マ行上一段　ラ行上一段
　ア行下一段　カ行下一段　ガ行下一段　サ行下一段　ザ行下一段　タ行下一段
　ナ行下一段　ハ行下一段　バ行下一段　マ行下一段　ラ行下一段
　サ行変格（ただし語幹のあるもののみ、例えば"勉強する"など）
形容詞
　シク
形容動詞
　ダ

活用テーブルの特徴は以下の通りである。
・活用の種類には、活用の行（ア行、カ行など）を含めて記述してある。従ってdefdictを記述する際の活用の種類（３．１の<app>）には、*katsuyou-table*に準じた記述をしなければならない。（関連　３．３　辞書のシンタックスチェック）
・活用において変化しないものを語幹として扱っている。従って、動詞の上一段活用、下一段活用においては標準的な活用と異なっている。
・音便についても活用テーブルで扱う。扱う音便は以下の通りである。
　　イ音便　カ行五段　ガ行五段
　　撥音便　バ行五段　マ行五段　ナ行五段
　　促音便　タ行五段　ア行五段　ワ行五段　ラ行五段

＊注　参考文献　中学国文法　小畑哲雄

（２） 語幹のない語

語幹のない動詞および助動詞についてはすべての活用を登録するものとする。

－－例－－

動詞 "来る"

```
(defdict "こ"   "動詞" ((kuru mizen    ko) hum) t)
(defdict "き"   "動詞" ((kuru renyou   ki) hum) t)
(defdict "くる" "動詞" ((kuru shuusi kuru) hum) t)
(defdict "くる" "動詞" ((kuru rentai kuru) hum) t)
(defdict "くれ" "動詞" ((kuru katei  kure) hum) t)
(defdict "こい" "動詞" ((kuru meirei  koi) hum) t)
```

助動詞 "だ"

```
(defdict "だろ" "助動詞" ((daro mizen  daro) hum) t)
(defdict "だっ" "助動詞" ((daro renyou datt) hum) t)
(defdict "で"   "助動詞" ((daro renyou   de) hum) t)
(defdict "だ"   "助動詞" ((daro shuusi   da) hum) t)
(defdict "な"   "助動詞" ((daro rentai   na) hum) t)
(defdict "なら" "助動詞" ((daro katei  nara) hum) t)
```

3．構文解析

3．1　構文解析の方法

構文解析は、bottom-up 法にtop-down 予測情報を取入れたものとする。
処理の大まかな流れは、辞書情報をもとに形態素解析を行ない、その形態素解1つに対して、文法情報を用いて構文解析を行なう。以後解が見つかる間その処理を繰返す。

3．2　形態素解析

形態素解析は、最長一致法に従い前から順番に行なう。語の同定に失敗した時は未知語として処理することができる。また文章中に区切り箇所の設定が行なえる。

(1)　未知語処理

ａ．未知語処理のスイッチ

形態素解析時に辞書にない単語を未知語として処理できる。未知語処理は、大域変数 *unknown* が non nil の場合行なわれる。*unknown* の初期値は t である。

ｂ．未知語情報の登録

defunknown cat var struct check　　[マクロ]

unknown が non nil の場合、形態素解析時に未知語の切出しが行なわれるが、その際参照する未知語情報は、マクロ defunknown で登録される。それぞれの引数は defdict に準じる。var は 未知語の表記を受ける変数で、struct 中の var は未知語採用時に置き換えられる。これらの情報は大域変数 *unknown-information* に push される。なお、未知語情報が登録されていない時は、未知語処理を行なわない。

(2)　語・文章の区切り箇所の指定

語、文章の切れる箇所を文章に記号を挿入することにより予め設定しておくことができる。（以下、後の切れる箇所を指定する記号を区切り記号、文の切れる箇所を指定する記号を終了記号という。）
区切り記号は、大域変数 *kugiri* にセットされているストリングのリスト、終

了記号は大域変数 *end* にセットされているストリングのリストとする。（ユーザは自由に設定することができる。）

－－例－－
```
(setq *kugiri* '("," "、" "(" ")" "[" "]" .....etc))
(setq *end* '("。" "？" "！" "．" "＊".....etc))
```

a．区切り記号の取扱について

形態素解析において、文章中の区切り記号を越えて語の切出しは行なわない。

b．終了記号の取扱について

文章は、基本的に最後の要素が終了記号であるものとする。もし、文章の最後の要素以外に終了記号が出現した場合はそこまでの文章について処理を行い、残りは無視する。

なお、区切り記号、終了記号を指定されていない（*kugiri* *end* の値が nil）場合上記の処理は行なわれない。*kugiri* *end* の初期値は nil である。

－－例－－
```
(parse "文章" "犬が吠える。")
(parse "文章" "犬が吠える。人が泣く。")
            ;;; *end* に "。" があれば、"犬が吠える。"のみ処理、なければ
            ;;; 全てを処理
```

－－参考－－
形態素解析のアルゴリズム
① 文章にマッチする辞書で最長のものを候補として得る。
② 候補がなければ③へ。
③ 未知語処理を行なう。

未知語処理のアルゴリズム
① 最長一致で辞書に候補が見つからない。
② 候補が見つかるまで一つずつ右にずらす。
③ 候補が見つかった地点より左を未知語とする。

３．３　構文解析

形態素解析の結果に対して、指定された文法カテゴリーを根とする構文解析木を求める。解析は bottom-up 的に行ない、top-down 予測情報により処理を効率化する。

３．４　構文解析プログラムの関数仕様

```
parse root bunsho &key :input :output :print :lexical
```

bunsho（ストリング）の先頭から終了記号（４．２参照）まで（終了記号が指定されていない場合は最後まで）のサブストリングに対してキャリッジリターン（#¥newline）スペース（#¥space）を削除し、root（ストリング）が根となるようなすべての構文解析木を求め、その個数と、それぞれの構文解析木がもつ構造データのリスト、処理結果を multiple-values で返す。ここで処理結果とは、root を根とする構文解析木を作成の結果余った形態素解のリストであり、余りがない場合は "" （空ストリング）となる。

bunsho に nil を指定し、キーワード :input にファイルまたはストリームを指定するとそのファイルから１ストリングを読み込み、それを bunsho として処理する。:input で指定されたファイルが存在しない場合、ストリームが正しくない場合、ファイルにストリングが記述されていない場合はエラーとなる。また、input がストリングの場合でファイルの最後に到達している場合は、end_of_stream （シンボル）を返す。

:output にファイルまたはストリームが指定されていると、最終的な構造データをそのファイルに出力し、解の個数と処理結果を multiple-values で返す。:output で指定されたファイルがすでに存在している場合は付加えられる。

:print に t が指定されると、形態素解析、構文解析の解が見つかる度にその解を出力し、入力待ちとなる。そこでユーザの入力に従い次の形態素解析や構文解析に移行したり、終了したりする。
nil を指定すると結果出力なしにすべての解を求める。:print のデフォルトは t である。

:lexical に t が指定されると、構文解析木のリストイメージを構造データとして返す。nil であれば、root の構造を構造データとして返す。:lexical のデフォルトは nil である。

```
－－例－－
(parse "文章" "自然言語についての文献がほしい。")
(parse "文章" :input in :output out)  ;;;  in out はストリーム
(parse "文章" "自然言語についての文献がほしい。" :print nil)
                                        ;;;  中間出力なし
```

付録．1

形態素解析について

形態素解析の手法としては、

(1) アルゴリズムの簡単さ、

(2) 処理時間の速さ、

(3) 構文解析での失敗でバックトラックした際の次候補の管理がしやすい、

等の点から、最長一致法を採用した。形態素解析のための文法としては、学校文法を参考にしたが、一部処理の簡略化のために変更を行った。形態素解析の問題点として以下のことを考慮した。

1．活用のある語のマッチング処理の方法

活用のある語については、活用語尾を含んだ部分とのマッチングを行わなければならない。そのための戦略として次のことを行った。

(1) 活用のある語は辞書には語幹のみを登録し、語尾を活用テーブルに登録することで、全体としてマッチングを行う構造にした。

(2) 助動詞、不規則動詞（する、来る）は語幹がなく、(1)の処理ではうまくいかない。これらの語は数が少なく、共通する規則性がないので、活用形を全て辞書に登録し、活用テーブルは参照しないことで解決した。但し、語幹のあるサ変動詞（制御する等）は活用テーブルを用いることとした。

(3) 上一段、下一段の動詞のなかには語幹のない動詞があり（見る、診る、観る）、(1)の処理ではうまくいかない。これらの語は数が多く、規則性があるため、全ての活用を辞書に登録することは不経済である。ここで上一段、下一段動詞の活用を調べると、

「＊、＊、＊る、＊る、＊れ、＊ろ、＊よ」（＊は同じ文字）

となる。よって、上一段、下一段の動詞は活用語尾の一文字目を語幹に含むこととし、活用テーブルは未然形、連用形については活用語尾なしとすることによって解決した。

＜まとめ＞

・活用テーブルを作成する語について

形容詞、形容動詞、五段活用動詞、下一段・上一段動詞、「する」を除くサ変動詞。

辞書の見出し語は語幹のみを登録。

下一・上一動詞の語幹は原形から「る」を除いた部分とする。

辞書には情報として活用の種類を記述しておく。

それぞれの活用の種類に対して活用テーブルを作る。

・活用テーブルを作成しない語について

助動詞、「来る」、「する」。

辞書の見出し語は活用形の全てを登録。

辞書には情報として活用形を記述しておく。

2．処理の高速化

2．1　ハッシュテーブルによる処理の高速化

形態解析においては、辞書の検索範囲をせばめることにより処理を高速化できる。そこで、ＬＩＳＰのハッシュ機能を用い、検索の範囲をせばめることを考え、ハッシュテーブルのキーを計算する適当な関数を用意し、同じキーになる見出し語は、リストとしてテーブルに割り当てることとした。ハッシュテーブルのサイズについては、あまり大きくしすぎることはメモリの負担をかけすぎるのではないかとの危惧があり（辞書の見出し語の数にもよるが限界はよくわかっていない）適当な大きさ（2^{14}程度）のテーブルサイズにすることにした。

2．2　形態素解析のアルゴリズム

最長一致法のアルゴリズムとして次の二つを考え、比較検討した。

検討項目	アルゴリズム１	アルゴリズム２
概要	入力全体が見出し語と一致するかどうかを調べ、一致すれば単語として切り出す。一致しなければ、最後尾の一文字を削除し、この処理を繰り返す（図１）。ハッシュテーブルのキーを計算する関数は、見出し語の語長と、一文字目の文字コードの一部により計算される。	入力全体から見出し語の語長分だけ切出し、一致するかどうかを調べる。一致した単語の中で最長の単語を切り出す（図１）。ハッシュテーブルのキーを計算する関数は、一文字目の文字コードの一部によって計算される。
検索空間	単語が見つかった時点で検索を打ち切るため、場合によって検索空間は異なる。検索空間が最も広くなる場合の大きさは、一文字目に対応した複数のハッシュテーブルに登録されている見出し語の総数である。	検索空間は入力の一文字目に依存し、未知語である場合も含めて常に一定であり、その大きさは、一文字目に対応したハッシュテーブルに登録されている見出し語の数である。
検索時間	検索は単純な比較でよいため、一つの見出し語に対する検索時間は短い。	入力から見出し語の語長分切り出すという操作が入るため、検索時間が余分にかかる。
活用テーブル検索	単語を切り出す際、毎回、活用テーブルの全てを検索する。	マッチした際に、その単語の活用形のみ検索する。

検索空間をハッシュテーブルのサイズが等しいとして比較すると、アルゴリズム１の最悪の場合の検索空間は、見出し語の最大語長倍、アルゴリズム２より大きくなる。アルゴリズム１は一致した時点で検索を打ち切るということを考慮しても、長い単語から順に検索することと、長い単語ほど出現頻度が低いという直観（日本語でよく使われる単語は１文字か２文字の比較的短い単語が多い）により、検索空間の面からはアルゴリズム２の方が有利である。また、検索時間、活用テーブルの検索時間については、それぞれアルゴリズム１、アルゴリズム２が有利である。これらのことを総合的に考慮してアルゴリズム２の方が有利であると判断し、今回のパーザではアルゴリズム２を採用することにした。

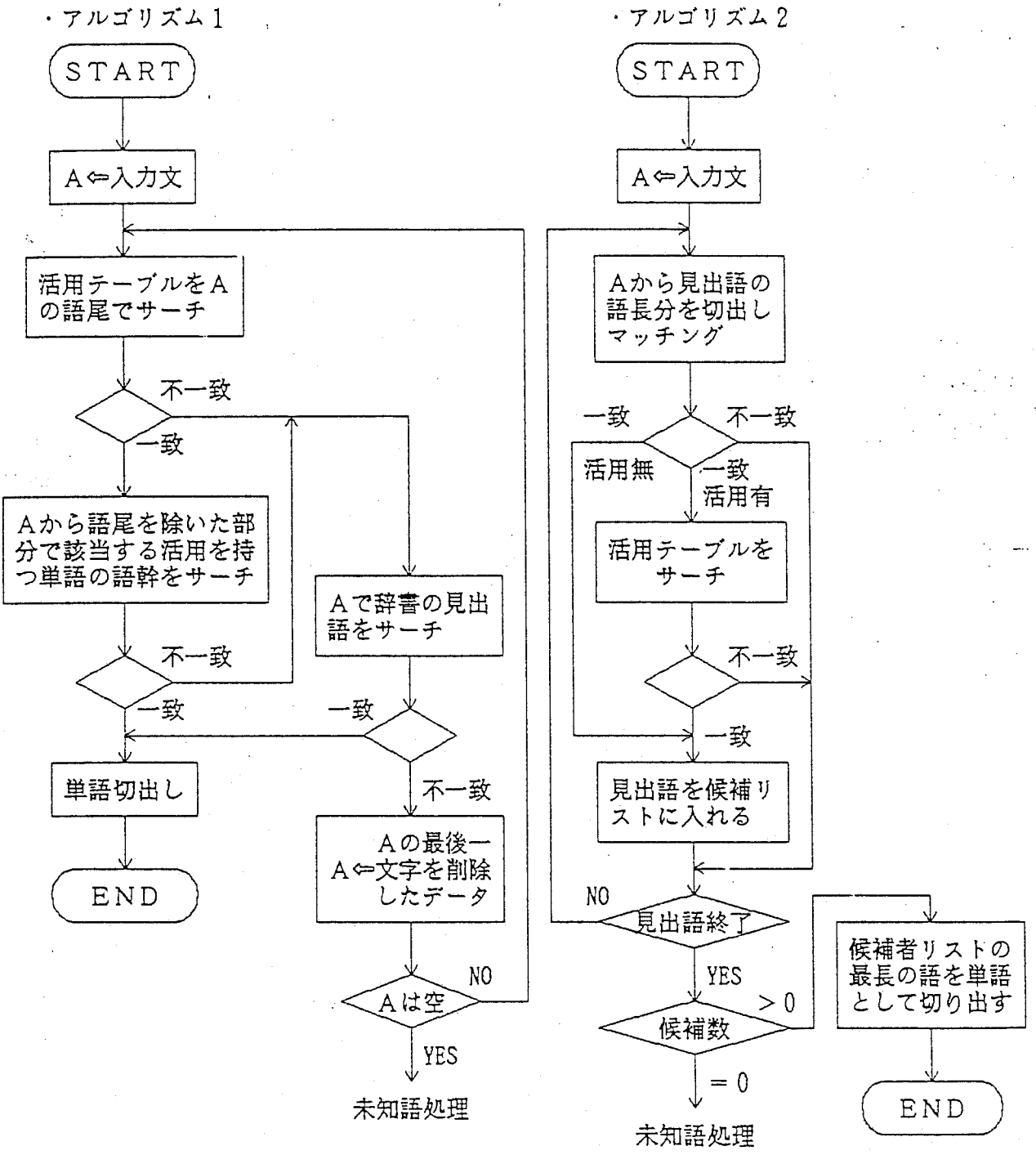


図１．形態素解析のアルゴリズム

＜まとめ＞

・アルゴリズムとしてアルゴリズム２を使う。

・辞書の見出し語は、ハッシュテーブルにリスト形式で格納されている。

・ハッシュテーブルのキーは最初の文字のＪＩＳコードをもとに計算する。

・キーを計算する関数は効率のよいものに変えるためにユーザが定義できる。

＜参考＞　ハッシュテーブルのキーを計算する関数

見出し語の一文字目のＪＩＳコードをもとに計算する。キーを計算する関数は効率のよいものに変えるためにユーザが定義できる。デフォルトのキー関数としては次の計算式を用いた。ただし、Ｘは一文字目のＪＩＳコードである。

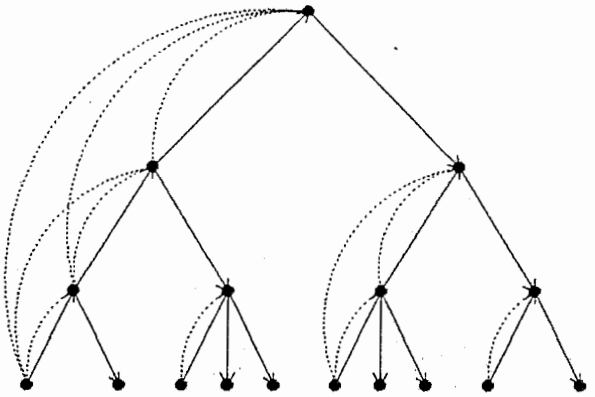
キー関数＝（+（AND X 7FH）（/（AND X 7F00H） 2））

即ち、この関数はＪＩＳコードの有効ビット（上位バイトの下位７ビットと下位バイトの下位７ビットを使った１４ビット）によってキーを計算する計算式である。

付録 2. bottom up 構文解析の効率化

1. top down 予測情報

下図において、各部分構文木の根と、その根から最左の枝を辿って得られる節との関係 (破線) は、構文解析の到達可能性を表す。この top down 予測情報を用いて規則の適用を制限することにより、 bottom up 構文解析の効率化を図ることができる.



2. 表現方法

1. top down 予測情報は、 $(right_1.root)$ を要素とするリストである。
2. $right_1$ は、 規則の右辺左端の範とする。
3. $root$ は、 部分構文木の根の範とする。
4. $(right_1.root)$ は、 $right_1$ が $root$ に接続可能であることを表す。
5. top down 予測情報は、 大域変数 *link-category-list* に束縛される。

3. 抽出方法

1. *link-category-list* を空にする。
2. 規則に現れる全ての範に対して以下を行う。
   (a) ある範を cat とする。
   (b) $(cat.cat)$ を *link-category-list* に追加する。
3. すべての規則に対して以下を行う。
   (a) ある規則の右辺左端の範を $right_1$ とする。
   (b) その規則の左辺の範を $left$ とする。
   (c) *link-category-list* に $(right_1.left)$ が含まれていれば何もしない。
   (d) *link-category-list* に $(right_1.left)$ が含まれていなければ以下を行う。
      i. x を任意の範とする。
      ii. y を任意の範とする。
      iii. *link-category-list* の中で、 $(x.right_1)$ に一致する x と $(left.y)$ に一致する y の すべての組み合わせについて、 $(x.y)$ を *link-category-list* に追加する。

4. 抽出例

次の規則に対応して、 *link-category-list* が追加されていく様子を例示する。

	(a a) (b b) (c c) (d d) (e e) (f f) (g g)
a→b,c.	(b a)
d→b,e,f.	(b d)
b→g,h.	(g b) (g a) (g d)
g→e,f.	(e g) (e b) (e a) (e d)

5. 解析方法

解析に先立ち木の根の範が与えられ、その範を部分木として解析を行う。

ある部分木の解析方法を以下に述べる。

1. その部分木の根の範を$root$とする。
2. 3. または 4. のどちらかが成功すれば成功、ともに失敗すれば失敗。
3. (a) 先頭の単語の範を$word$とする。
   (b) *link-category-list* に $(word.root)$ が含まれていなければ失敗。
   (c) $word$について規則の適用を行い、成功すれば成功、失敗すれば失敗。
4. (a) 右辺が空である規則の左辺の範を$left$とする。
   (b) *link-category-list* に $(left.root)$ が含まれていなければ失敗。
   (c) $left$について規則の適用を行い、成功すれば成功、失敗すれば失敗。

ある範について規則の適用方法を以下に述べる。

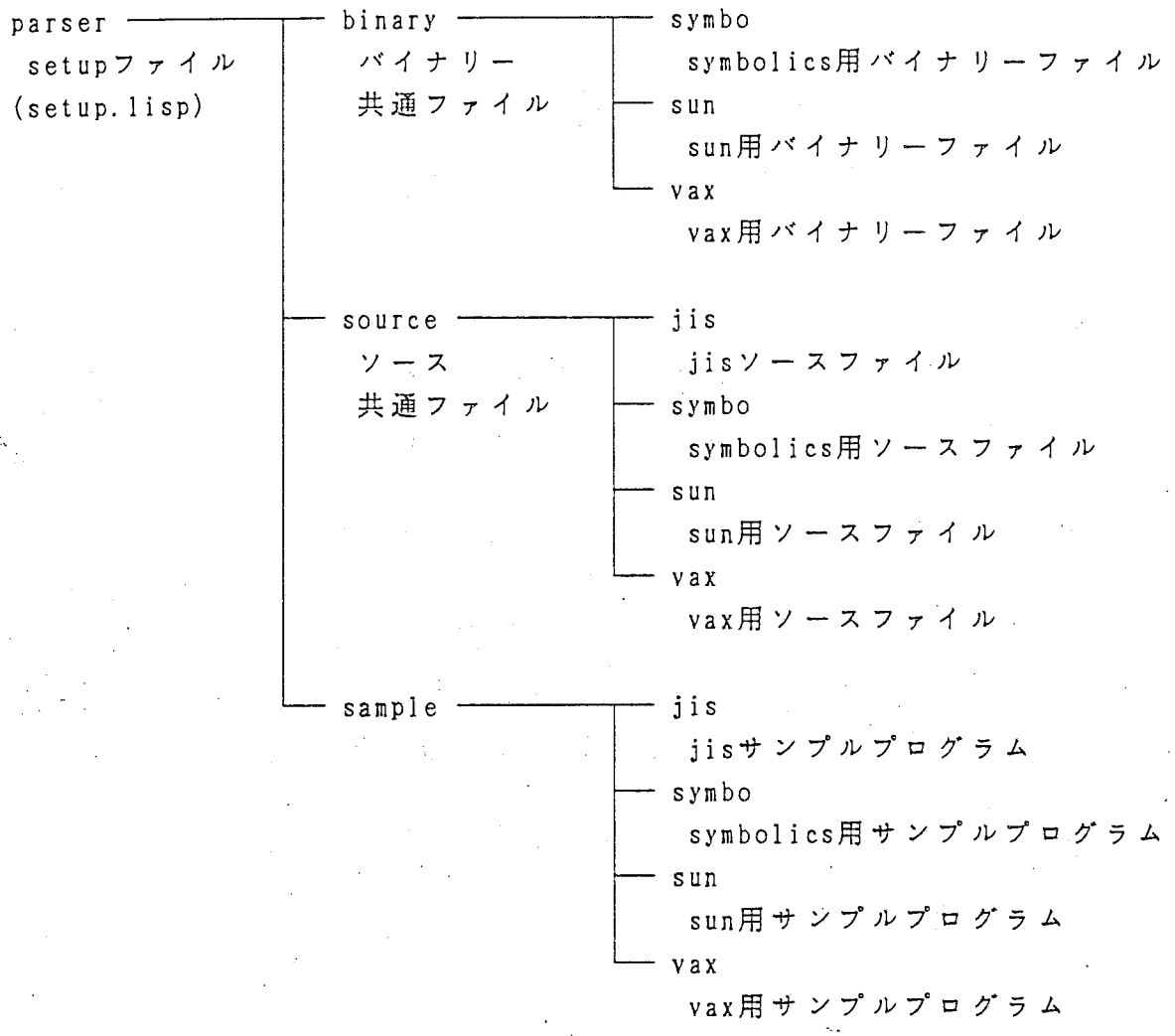
1. 適用する範をcatとする。
2. ある規則の右辺の範を左から$right_i(i = 1, 2, ..., n)$とする。
3. その規則の左辺の範を$left$とする。
4. catと$right_1$の等しい規則がなければ失敗。
5. $right_i(i = 2, 3, ..., n)$を順に部分木の根の範とし、残りの単語列に対して再帰的に解析を進める。いずれかの部分木の解析が失敗すればその時点で失敗となる。
6. $left$について再帰的に規則の適用を行い、成功すれば成功、失敗すれば失敗。

Ⅱ章. 操作仕様

1．インストール

(1) ＭＴの構成

以下の様なディレクトリィ構成になっている。



(2) 環境設定

setupファイル(setup.lisp)中の *install-directory* の値を parser の上のディレクトリーに設定する。

(3) コンパイル

関数 parser-compile-all で必要な全てのソースファイルがコンパイルされ、バイナリーファイルが生成される。

2．準備

(1) セットアップ

setupファイルをローディングする。

Command: (load "適切なパス名setup.lisp")

.....

必要なファイルがローディングされる。

.....

Parser: ;;; プロンプト変更
;;; (VAX 版については変更されない)

(2) 環境設定、ユーザ定義関数など

ファイル：prs_usrdefine.lisp中に記述してある以下の関数は必要に応じてユーザが書き換えることを推奨する。

gram-key [関数]
文法ルールを登録する際のハッシュ関数

dict-key [関数]
辞書を登録する際のハッシュ関数

以下の大域変数は、プログラム実行前に適切なパス名に変更しておく必要がある。

dict-log-file [大域変数]
辞書登録・削除のログファイル

current-directory [大域変数]
指定されたパス名が存在しない場合、この大域変数で指定されるパス名のファイルを対象にする。

3．辞書

（1） 辞書登録

defdict dict　　［マクロ］

指定された辞書情報(gram)を辞書ハッシュテーブルに登録する。

【辞書の書式】
辞書の書式は以下の通りである。

```
<dict>      ::= ( <lexs> <cats> <struct> <check> )
<lexs>      ::= <lex> |          ;;;  活用無し
                ( <lex>          ;;;  活用有り
                  [<lvar>] )     ;;;  活用語尾を受ける変数
<cats>      ::= <cat> |          ;;;  品詞
                ( <cat>          ;;;  活用有り
                  <app>          ;;;  活用の種類（カ行五段 etc）
                  [<avar>] )     ;;;  活用形を受ける変数
<struct>    ::= lisp object      ;;;  構造
<check>     ::= lisp object      ;;;  チェック
<lex>       ::= string           ;;;  見出し語
<cat>       ::= string           ;;;  品詞
<app>       ::= string
<lvar>      ::= symbol
<avar>      ::= symbol
```

【活用のある語の登録】
活用のある語を登録する際は、大域変数 *katsuyou-table* と整合性のある記述を行なう必要がある。
デフォルトで扱える活用の種類は以下の通りである。

動詞
カ行五段　ガ行五段　サ行五段　タ行五段　ナ行五段　バ行五段
マ行五段　ラ行五段　ワ行五段
ア行上一段　カ行上一段　ガ行上一段　サ行上一段　ザ行上一段　タ行上一段
ナ行上一段　ハ行上一段　バ行上一段　マ行上一段　ラ行上一段
ア行下一段　カ行下一段　ガ行下一段　サ行下一段　ザ行下一段　タ行下一段
ダ行下一段　ナ行下一段　ハ行下一段　バ行下一段　マ行下一段　ラ行下一段
サ行変格（ただし語幹のあるもののみ、例えば "勉強する" など）

形容詞
　シク
形容動詞
　ダ

＊注　語幹のない動詞および助動詞についてはすべての活用を登録するものとする必要がある。

```
――例――
動詞　"来る"
(defdict "こ"　"動詞" ((kuru mizen　ko) hum) t)
(defdict "き"　"動詞" ((kuru renyou　ki) hum) t)
(defdict "くる" "動詞" ((kuru shuusi kuru) hum) t)
(defdict "くる" "動詞" ((kuru rentai kuru) hum) t)
(defdict "くれ" "動詞" ((kuru katei　kure) hum) t)
(defdict "こい" "動詞" ((kuru meirei　koi) hum) t)

助動詞　"だ"
(defdict "だろ" "助動詞" ((daro mizen　daro) hum) t)
(defdict "だっ" "助動詞" ((daro renyou datt) hum) t)
(defdict "で"　"助動詞" ((daro renyou　de) hum) t)
(defdict "だ"　"助動詞" ((daro shuusi　da) hum) t)
(defdict "な"　"助動詞" ((daro rentai　na) hum) t)
(defdict "なら" "助動詞" ((daro katei　nara) hum) t)
```

【シンタックスチェック】

辞書登録の際以下のシンタックスチェックを行なう。該当する場合は適切な warning を出力し、辞書ハッシュテーブルに登録しない。

a．活用がない場合
- 引数が４つでない。
- <lex> がストリングでない。
- <cat> がストリングでない。

b．活用がある場合

（１）に加えて
- 引数が４つでない。
- <lvar>（活用語尾を受ける変数）が記述されている場合シンボルでない。
- <avar>（活用形を受ける変数）が記述されている場合シンボルでない。
- <app>（活用の種類）が活用テーブル（３．９参照）にない。
- <check> に <lvar> <avar> でない未束縛のシンボルが使用されている。ただ

しマクロとスペシャルフォームの引数についてはこの限りでない。

【リターン値】

defdict は、登録に成功すればその定義イメージを、失敗すれば NIL を返す。（シンタックスエラー、または、重複登録）

【その他】

なお、fload nfload（５．参照）以外の方法で（リスプインタプリタ上で直接定義された場合や、load による場合など）defdict による辞書ハッシュテーブルの変更が行なわれた場合、その記述イメージが *dict-log-file* に束縛されているファイルに付加えられる。

－－例－－

```
Parse: (defdict "人" "名詞" (hito hum) t)          ;;; 活用がない場合
(DEFDICT "人" "名詞" (HITO HUM) T)                 ;;; リターン値
Parse: (defdict ("泣" n) ("動詞" "カ行五段" k) ((na e i) hum) t)
                                                   ;;; 活用がある場合
(DEFDICT ("泣" N) ("動詞" "カ行五段" K) ((NA E I) HUM) T)
                                                   ;;; リターン値
Parse: (defdict ("泣" n) ("動詞" "カ行五段" k) ((na e i) hum) t)
NIL                                                ;;; リターン値
```

ファイル　*dict-log-file*

```
.....................................
(DEFDICT "人" "名詞" (HITO HUM) T)
(DEFDICT ("泣" N) ("動詞" "カ行五段" K) ((NA E I) HUM) T)
```

(2) 辞書検索

find-dict lex &key :hinshi :kouzou :check [関数]

ハッシュテーブルに対して辞書情報の検索を行う。
lex で指定された見出し語を持つ辞書情報全てをリストにして返す。
キーワード hinshi kouzou check はそれぞれ defdict の hinshi kouzou check に対応しており、指定されたときはそれらを含めてマッチする辞書情報全てをリストにして返す。
該当する辞書情報がないときは nil を返す。

－例－

Parse: (find-dict "人")
(("人" "名詞" (HITO HUM) T) ("人" "名詞" (NIN) T))
Parse: (find-dict "人" :hinshi "名詞" :kouzou (hito hum) :check t)
(("人" "名詞" (HITO HUM) T))

(3) 辞書削除

del-dict lex &key :hinshi :kouzou :check [関数]

指定された辞書情報を辞書ハッシュテーブルから削除する。
lex で指定された見出し語を持つ辞書情報を辞書ハッシュテーブルから全て削除する。
キーワード hinshi kouzou check はそれぞれ defdict の cat struct check に対応しており、指定されたときはそれらを含めてマッチする辞書情報が全て削除される。
削除項目がある場合はその記述イメージが、ない場合は nil を返す。
del-dict による辞書ハッシュテーブルの変更が行なわれた場合、その記述イメージが *dict-log-file* に束縛されているファイルに付加えられる。

－－例－－

Parse: (del-dict "人")
(DEL-DICT "人")
Parse: (del-dict "人" :hinshi "名詞" :kouzou (hito hum) :check t)
NIL

ファイル　*dict-log-file*

```
(DEFDICT "人" "名詞" (HITO HUM) T)
(DEFDICT ("泣" N) ("動詞" "カ行五段" K) ((NA E I) HUM) T)
.....................................
(DEL-DICT "人")
```

4．文法関係

(1) 文法登録

defgram lhs . rhs　　［マクロ］

右辺が lhs 左辺 rhs の文法を文法ハッシュテーブルに登録する。

【文法の書式】
文法の書式は以下の通りである。

```
<lhs>      ::= (<cat> <struct> <check>)
<rhs>      ::= nil |                          ;;;  ε規則
               ( <term> {<term>}* )
<term>     ::= <term1> |
               ( <term1> . <continue> )       ;;;  繰り返し規則
<term1>    ::= ( <cat> <var> <check> )
<continue> ::= ( nil ) |
               ( <count> <var> <check> )
<cat>      ::= string                         ;;;  文法カテゴリー
<struct>   ::= lisp object                    ;;;  構造作成
<check>    ::= lisp object                    ;;;  チェック
<var>      ::= symbol                         ;;;  下からの構造を受ける変数
<count>    ::= (number) |                     ;;;  number 以上の繰り返し
               (number1 number2)              ;;;  number1 から number2
                                              ;;;  までの繰返し
```

【シンタックスチェック】
文法登録の際以下のシンタックスチェックを行なう。該当する場合は、適切な warning を出力し登録しない。

a．通常の文法の場合

①右辺
- <term1> のレングスが3でない。
- <cat> がストリングでない。
- <var> がシンボルでない。
- 一連の <term1> 中に <var> が重複して現れる。
- <check> に <var> でない未束縛のシンボルが使用されている。ただしマクロとスペシャルフォームの引数についてはこの限りでない。

②左辺

・<term1> のレングスが３でない。

・<cat> がストリングでない。

・<struct> <check> に右辺の <var> 以外の未束縛のシンボルが使用されている。ただしマクロとスペシャルフォームの引数についてはこの限りでない。

b．繰り返し規則の場合

① <continue> が (nil) の繰返しの <term>

(a) 右辺

・<term1> について(１)の右辺と同じチェック

・<continue> が (nil) でない。

(b) 左辺

・(１)と同じ

② <continue> が (nil) でない繰返しの <term>

(a) 右辺

・<term1> について(１)の右辺と同じチェック

・<continue> のレングスが３でない。

・<continue> の <var> がシンボルでない。

・<continue> の <check> に <continue> の <var> 以外の未束縛のシンボルが使用されている。ただしマクロとスペシャルフォームの引数についてはこの限りでない。

(i) <count> が (number) の場合

・number が正の整数でない。

(ii)<count> が (number1 number2)の場合

(i) に加えて number1 number2 が昇順でない。

(b) 左辺

・(１)と同じ

c．ε規則の場合

(１)に同じ

d．追加

<struct> <check> において、未定義の関数が含まれている場合は、warning を出力するが、文法ハッシュテーブルには登録する。

【リターン値】

defgram は、登録に成功すればその文法情報を、失敗すれば NIL を返す。（シンタックスエラー、または、重複登録）

－－例－－

```
(defgram ("名詞句" (cons 'agent n) t)     ;;; 通常の文法規則
         ("名詞" n t)                     ;;;
         ("助詞" p (equal 'ga (car p))))  ;;;

(defgram ("修飾語句" (list np2) t)        ;;; 繰返し規則
         (("名詞句" np1 t) (2 4) np2 t))  ;;;

(defgram ("名詞句" (eps) t))              ;;; ε規則
```

(2) 文法検索

find-gram cat　　[関数]

ハッシュテーブルに対して文法の検索を行う。
cat はストリングでなければならない。cat が右辺左端の文法カテゴリーに一致する文法全てを返す。
なお、ε規則を検索する場合には、関数 get-epsilon-rule で行なうこと。

```
Parser: (find-gram "名詞句")
((("名詞句" (cons 'agent n) t) ("名詞" n t) ("助詞" p (equal 'ga (car p))))
 (("修飾語句" (list np2) t) (("名詞句" np1 t) (2 4) np2 t)))
 ..................................................
```

5．文法・辞書のロード

（1） 文法・辞書ハッシュテーブルの初期化

finit initialize-table　　［関数］

文法ハッシュテーブルは、関数 finit で初期化される。この関数は、文法ハッシュテーブルのだけではなく、辞書ハッシュテーブル（辞書データもハッシュテーブルで持つ、3参照）の初期化も行うことができる。

initialize-table は、g d b のいずれかであり、g であれば文法ハッシュテーブルを、d であれば辞書ハッシュテーブルを、b であればそれらの両方を初期化する。なお、g または b が指定された際は、トップダウン予測情報も初期化される。

```
－－例－－
(finit 'g)          ;;;  文法ハッシュテーブル初期化
(finit 'd)          ;;;  辞書ハッシュテーブル初期化
(finit 'b)          ;;;  文法・辞書ハッシュテーブル初期化
```

（2） 文法・辞書ファイルのローディング

a．ハッシュテーブルを初期化する場合

fload &rest data-files　　［関数］

文法・辞書ファイルは、関数 fload を用いてまとめてローディングする。その際、文法・辞書ハッシュテーブルを初期化する。

```
－－例－－
(fload "a.gram" "b.gram")
(fload "a.gram" "b.gram" "a.dict")
```

b．ハッシュテーブルを初期化しない場合

nfload &rest data-files　　［関数］

関数 nfload を用いて、辞書・文法ファイルをローディングすると、ハッシュテーブルの初期化を行なわない。

＊注　lisp の load とは、修正記録をとらないという点で異なっている。従って defdict を多く含むファイルは、nfload でローディングするとかなり高速になる。

6．構文解析

（1） 構文解析

parse root bunsho &key :input :output :print :lexical :struct

bunsho（ストリング）の先頭から終了記号（3．2参照）まで（終了記号が指定されていない場合は最後まで）のサブストリングに対してキャリッジリターン（#¥newline）スペース（#¥space）を削除し、root（ストリング）が根となるようなすべての構文解析木を求め、その個数と、それぞれの構文解析木がもつ構造データのリスト、処理結果を multiple-values で返す。ここで処理結果とは、root を根とする構文解析木を作成の結果余った形態素解のリストであり、余りがない場合は対応する要素が ""（空ストリング）となる。

bunsho に nil を指定し、キーワード :input にファイルまたはストリームを指定するとそのファイルから1ストリングを読み込み、それを bunsho として処理する。:input で指定されたファイルが存在しない場合、ストリームが正しくない場合、ファイルにストリングが記述されていない場合は warning を出力し処理を中止する。また、input がストリングの場合でファイルの最後に到達している場合は、end_of_stream （シンボル）を返す。

:output にファイルまたはストリームが指定されていると、最終的な構造データをそのファイルに出力し、解の個数と処理結果を multiple-values で返す。:output で指定されたファイルがすでに存在している場合は付加えられる。

:print に t が指定されると、形態素解析、構文解析の解が見つかる度にその解を出力し、入力待ちとなる。そこでユーザの入力に従い次の形態素解析や構文解析に移行したり、処理を終了したりする。nil を指定すると結果出力なしにすべての解を求める。:print のデフォルトは t である。

:lexical に t が指定されると、構文解析木のリストイメージを構造データとして返す。nil であれば、root の構造を構造データとして返す。:lexical のデフォルトは nil である。

:struct に t が指定されると、中間出力の構文解析木に構造情報が出力される。nil であれば出力されない。:struct のデフォルトは t である。

－－例－－

(parse "文章" "これが構文解析木の例だ。")

```
Parser: (parse "文章" "これが構文解析木の例だ。")
 Count : 1 Morphene
| 代名詞(KORE MONO)]これ | 助詞(GA KAKUJOSHI)]が | 名詞(KOUBUN NOUN)]構文 | 名詞(KAISEKI NOUN)]解析 |
 名詞(KI NOUN)]木 |
 助詞(NO KAKUJOSHI)]の | 名詞(REI INSTANCE)]例 | 助動詞(DA JODOUSHI ("だ" "終止形"))]だ | 句点(MARU E
ND)]。 |
Exit(E) Next Morphene(N) Parser(P)>> p

 Count : 1 Parsing Tree
文章.(((((KORE MONO)) (GA KAKUJOSHI))
       (((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE)) (DA JODOUSHI (
"だ" "終止形"))))
      (MARU END))
   文.((((KORE MONO)) (GA KAKUJOSHI))
       (((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE)) (DA JODOUSHI (
"だ" "終止形"))))
      主部.(((KORE MONO)) (GA KAKUJOSHI))
         名詞句.((KORE MONO))
            代名詞.(KORE MONO)..これ
         助詞.(GA KAKUJOSHI)..が
      述部.(((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE)) (DA JODOUS
HI ("だ" "終止形")))
         名詞句.((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE))
            連体修飾語.(((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI))
               名詞句.((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN))
                  名詞.(KOUBUN NOUN)..構文
                  名詞.(KAISEKI NOUN)..解析
                  名詞.(KI NOUN)..木
               助詞.(NO KAKUJOSHI)..の
            名詞.(REI INSTANCE)..例
         助動詞.(DA JODOUSHI ("だ" "終止形"))..だ
   句点.(MARU END)...
Exit(E) Next(N)>>n
Exit(E) Next Morphene(N)>> n
1
((((((KORE MONO)) (GA KAKUJOSHI))
   (((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE)) (DA JODOUSHI ("だ"
 "終止形"))))
  (MARU END)))
("")
```

(parse "文章" "これが構文解析木の例だ。" :print nil)

```
Parser: (parse "文章" "これが構文解析木の例だ。" :print nil)
1
((((((KORE MONO)) (GA KAKUJOSHI))
   (((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE)) (DA JODOUSHI ("だ"
 "終止形"))))
  (MARU END)))
("")
```

(parse "文章" "これが構文解析木の例だ。" :lexical t)

```
Parser: (parse "文章" "これが構文解析木の例だ。" :lexical t)
 Count : 1 Morphene
| 代名詞(KORE MONO)]これ | 助詞(GA KAKUJOSHI)]が | 名詞(KOUBUN NOUN)]構文 | 名詞(KAISEKI NOUN)]解析 |
 名詞(KI NOUN)]木 |
 助詞(NO KAKUJOSHI)]の | 名詞(REI INSTANCE)]例 | 助動詞(DA JODOUSHI ("だ" "終止形"))]だ | 句点(MARU E
ND)]。 |
Exit(E) Next Morphene(N) Parser(P)>> p

 Count : 1 Parsing Tree
文章.(((((KORE MONO)) (GA KAKUJOSHI))
       (((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE)) (DA JODOUSHI (
"だ" "終止形"))))
      (MARU END))
   文.(((((KORE MONO)) (GA KAKUJOSHI))
       (((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE)) (DA JODOUSHI (
"だ" "終止形"))))
      主部.(((KORE MONO)) (GA KAKUJOSHI))
         名詞句.((KORE MONO))
            代名詞.(KORE MONO)..これ
         助詞.(GA KAKUJOSHI)..が
      述部.(((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE)) (DA JODOUS
HI ("だ" "終止形")))
         名詞句.((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE))
            連体修飾語.(((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI))
               名詞句.((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN))
                  名詞.(KOUBUN NOUN)..構文
                  名詞.(KAISEKI NOUN)..解析
                  名詞.(KI NOUN)..木
               助詞.(NO KAKUJOSHI)..の
            名詞.(REI INSTANCE)..例
         助動詞.(DA JODOUSHI ("だ" "終止形"))..だ
   句点.(MARU END)...
Exit(E) Next(N)>>n
Exit(E) Next Morphene(N)>> n
```

```
1
(("文章"
   (((((KORE MONO)) (GA KAKUJOSHI))
     (((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE)) (DA JODOUSHI ("だ
" "終止形"))))
    (MARU END))
   T)
  ((E QUOTE (MARU END))
   (S QUOTE
      ((((KORE MONO)) (GA KAKUJOSHI))
       (((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE)) (DA JODOUSHI (
"だ" "終止形"))))))
  (("文"
     ((((KORE MONO)) (GA KAKUJOSHI))
      (((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE)) (DA JODOUSHI ("
だ" "終止形"))))
      T)
    ((PP QUOTE (((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE)) (DA JO
DOUSHI ("だ" "終止形")
))
      )
     (SP QUOTE (((KORE MONO)) (GA KAKUJOSHI))))
   (("主部" (((KORE MONO)) (GA KAKUJOSHI)) T) ((P QUOTE (GA KAKUJOSHI)) (NP QUOTE ((KORE MONO)))
)
     (("名詞句" ((KORE MONO)) T) ((N QUOTE (KORE MONO))) (("代名詞" (KORE MONO) "これ")))) (("助詞"
 (GA KAKUJOSHI) "が"
))))
    (("述部" (((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE)) (DA JODO
USHI ("だ" "終止形")))
 T)
     ((P QUOTE (DA JODOUSHI ("だ" "終止形")))
      (NP QUOTE ((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE))))
     (("名詞句" ((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE)) T)
       ((N QUOTE (REI INSTANCE)) (R QUOTE (((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSH
I))))
        ((("連体修飾語" (((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) T)
          ((P QUOTE (NO KAKUJOSHI)) (NP QUOTE ((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN))))
          (("名詞句" ((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) T)
            ((N-3 QUOTE (KI NOUN)) (N-2 QUOTE (KAISEKI NOUN)) (N-1 QUOTE (KOUBUN NOUN)))
            (("名詞" (KOUBUN NOUN) "構文")) (("名詞" (KAISEKI NOUN) "解析")) (("名詞" (KI NOUN) "木")
)))
           (("助詞" (NO KAKUJOSHI) "の"))))
         (("名詞" (REI INSTANCE) "例"))))
       (("助動詞" (DA JODOUSHI ("だ" "終止形")) "だ"))))))
   (("句点" (MARU END) "。")))))
("")
```

(parse "文章" "これが構文解析木の例だ。" :struct nil)

```
Parser: (parse "文章" "これが構文解析木の例だ。" :struct nil)
 Count : 1 Morphene
| 代名詞(KORE MONO)]これ | 助詞(GA KAKUJOSHI)]が | 名詞(KOUBUN NOUN)]構文 | 名詞(KAISEKI NOUN)]解析 |
 名詞(KI NOUN)]木 |
 助詞(NO KAKUJOSHI)]の | 名詞(REI INSTANCE)]例 | 助動詞(DA JODOUSHI ("だ" "終止形"))]だ | 句点(MARU E
ND)]。 |
Exit(E) Next Morphene(N) Parser(P)>> p

 Count : 1 Parsing Tree
文章
   文
      主部
         名詞句
            代名詞.これ
         助詞.が
      述部
         名詞句
            連体修飾語
               名詞句
                  名詞.構文
                  名詞.解析
                  名詞.木
               助詞.の
            名詞.例
         助動詞.だ
   句点.。
Exit(E) Next(N)>>n
Exit(E) Next Morphene(N)>> n
1
((((((KORE MONO)) (GA KAKUJOSHI))
   (((((KOUBUN NOUN) (KAISEKI NOUN) (KI NOUN)) (NO KAKUJOSHI)) (REI INSTANCE)) (DA JODOUSHI ("だ"
 "終止形"))))
  (MARU END)))
("")
```

（２） その他

ａ．未知語処理

(i)　未知語処理のスイッチ

unknown　　［大域変数］

形態素解析時に辞書にない単語を未知語として処理できる。未知語処理は、大域変数 *unknown* が non nil の場合行なわれる。*unknown* の初期値は t である。

(ii)　未知語情報の登録

defunknown cat var struct check　　［マクロ］

unknown が non nil の場合、形態素解析時に未知語の切出しが行なわれるが、その際参照する未知語情報は、マクロ defunknown で登録する。それぞれの引数は defdict に準じる。var は 未知語の表記を受ける変数で、struct 中の var は未知語採用時に置き換えられる。なお、当然ではあるが、未知語情報が登録されていない時は、未知語処理を行なわない。

ｂ．語・文章の区切り箇所の指定

kugiri *end*　　［大域変数］

語、文章の切れる箇所を文章に記号を挿入することにより予め設定しておくことができる。（以下、後の切れる箇所を指定する記号を区切り記号、文の切れる箇所を指定する記号を終了記号という。）

区切り記号は、大域変数 *kugiri* にセットされているストリングのリスト、終了記号は大域変数 *end* にセットされているストリングのリストとする。（ユーザは自由に設定することができる。）

```
－－例－－
(setq *kugiri* '("," "、" "(" ")" "[" "]" .....etc))
(setq *end* '("。" "？" "！" "." "＊".....etc))
```

(i)　区切り記号の取扱について

形態素解析において、文章中の区切り記号を越えて語の切出しは行なわない。

(ii) 終了記号の取扱について

文章は、基本的に最後の要素が終了記号であるものとする。もし、文章の最後の要素以外に終了記号が出現した場合はそこまでの文章について処理を行い、残りは無視する。

なお、区切り記号、終了記号を指定されていない（*kugiri* *end* の値が nil）場合上記の処理は行なわれない。*kugiri* *end* の初期値は nil である。

7. 周辺ツール

(1)トレーサ

parse-trace [大域変数]

大域変数 *parse-trace* が non nil の場合、構文解析中トレース情報が出力される。*parse-trace* の初期値は nil である。
トレース情報は以下の通りである。

```
Parser:
    Enter ゴールのスタック
        [呼出回数] Enter Parser>>>
                        ##Word##          ワード
                        ##PendingGoals##  ゴールのスタック
    Fail  失敗したゴール
        [呼出回数] Fail Parser>>>
                        ##FailWord##  失敗したワード
                        ##FailGoal##  失敗したゴール
  ＊注  ワード:  表記（構造）品詞  [::=  <lex>(<struct>)<cat> ]

Assemble:
    Enter 現在のサブツリー
        [呼出回数] Enter Assemble>>>
                        ##Subtree##       現在のサブツリー
                      ( ##Word##          現在のワード)
                        ##Environment##   環境変数
    Fail  失敗したサブツリー
        [呼出回数] Fail Assemble>>>
                        ##FailSubtree##       失敗したサブツリー
                      ( ##FailWord##      失敗したワード)
                        ##Environment##   環境変数

  ＊注  サブツリー:  根の情報  -->  真下の情報...
        [ ::=  (<cat> <struct> <check>) --> (<cat> <struct> <check>)
                                            {(<cat> <struct> <check>)}*]
        変数環境:  変数名 = 構造  .......
        [ ::=  { <var> = <struct> }* ]

Update:
    Enter 現在の文法の成立状況
        [呼出回数] Enter Parser>>>
    ##Grammer## 右辺 :== 成立したサブゴール... <未成立のサブゴール>
    ##Environment## 変数環境
    Fail  失敗した文法の成立状況
        [呼出回数] Fail Parser>>>
    ##FailGrammer## 右辺 :== 成立したサブゴール... <未成立のサブゴール>
    ##Environment##  変数環境
```

```
Fire:
    Fail  失敗した評価式
        [呼出回数] Fail Fire>>>
            ##FailFrom##      失敗したＳ式
            ##Environment##   変数環境
```

(2)　リスト表示プログラム

print-list list
&key :output :base :tabs :current-part :write-func :rest-part
:nest-level :element　　[関数]

list（リスト；構文解析の結果など）を字下げ出力する。
キーワード引数は、以下の様に出力の方法を指定する。

:output
出力ストリームを指定する。デフォルトは *standard-output* である。
:base
書き出しの字下げを指定する。デフォルトは 0 である。
:tabs
字下げ出力の際のスペースの出力回数を指定する。デフォルトは 2 である。
:current-part
与えられた部分構造データの表示するデータを得る関数を指定する。デフォルトは car である。
:write-func
得られたデータを出力する関数を指定する。デフォルトは write である。
:rest-part
与えられた構造データより、部分構造データを得る関数を指定する。デフォルトは cdr である。
:nest-level
表示するリストのネストのレベルの上限を指定する。デフォルトは無限大である。省略の際は # を出力する。
:element
表示するリストの要素の上限を指定する。デフォルトは無限大である。省略の際は . を出力する。

付録３　情報検索における問い合わせ文解析の利用例

情報検索における問い合わせ文解析の特徴

知的情報検索システムにおいて、利用者が言った問い合わせ文を解析するために、当構文解析プログラムを使用している。この解析は、『～が欲しい』という文型における～の部分の名詞句の解析に主体を置いている。解析すべき問い合わせ文と解析結果の表現の例を下に示す。

処理対象文：「 Dempster-Shafer理論について調べたい。」
処理結果（格フレーム表現）：

```
((action retrieve)
 (object (subject (special "Dempster-Shafer理論"))))
```

この解析は、次のような特徴を持つ。

- 適用分野、および解析内容ともに応用に強く依存しているため、文法としては、通常の日本語の文法記述ではなく、意味情報が文法規則上に現れたような記述（意味文法）を行っている。適用分野が限定されている場合には、このような意味文法の方が文法記述をコンパクトに行うことができる。
- 本研究においては、特に名詞間の係受け関係の解析に主眼を置いているため、助詞の使い方を限定し（約２５０文例の分析結果、検索要求文では助詞の使い方を１０数個の意味的なパターンに分類できることを確認した）助詞を意味的なマーカとして文法を記述する。

当構文解析プログラムにおける文法記述の特徴

文法は、下記のような形式で記述される拡張文脈自由文法である。

```
／＊　左辺の品詞　⇨　右辺の品詞1　・　右辺の品詞n　＊／
(defgram　(左辺の品詞　構造作成用ＬＩＳＰ式　チェック用ＬＩＳＰ式)
          (右辺の品詞1 下位の構造を受ける変数　チェック用ＬＩＳＰ式)
                :
          (右辺の品詞n 下位の構造を受ける変数　チェック用ＬＩＳＰ式))
```

この文法において、次のような記述が可能である。

a）文法規則が成立したときに、上位の文法に渡すリスト構造を記述できる。これにより、単に構文木を出力するだけでなく、問い合わせ文の解析で必要となる様な格フレームを作成するLISPプログラムを文法中に記述できる。

b）文法規則が成立する条件を記述できる。これにより、単に構文的なマッチングだけでなく、意味的なマッチング条件を記述できる。この場合には、辞書に意味情報を記述する必要がある。また、右辺の品詞間の意味的な関連によるマッチング条件も記述できる。

解析例

前述の問い合わせ例における解析の仕方を述べる。

i）辞書とマッチングを行い、辞書に記述してある品詞（special, 助詞, 動詞, 要求助動詞）および上位に持ち上がる構造（(special "Dempster-Shafer理論"), (NITUITE), (SIRABERU 連用)(TAI)）からなる形態素解を作成する。

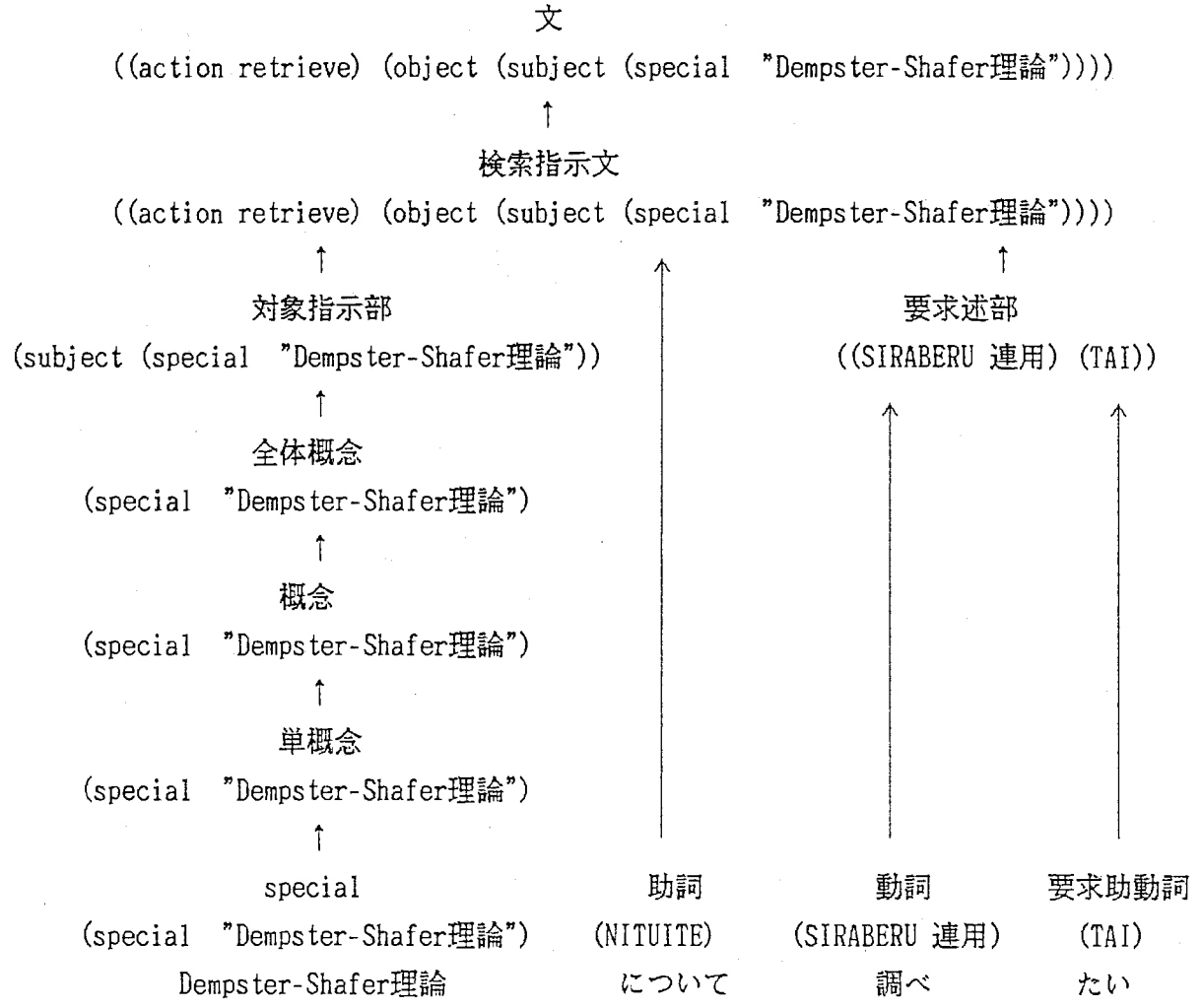
ii）それまでに作成した部分構造（あるいは形態素解）と文法の右辺とのマッチングを行い、マッチした場合には、文法の右辺に記述してある構造作成用ＬＩＳＰ式を評価し、更に上位の文法に渡す部分構造を作成する。文法の左辺が最上位の品詞である場合には、処理を終了する。この時、次のようなチェックを行うことが出来る。

a)文法右辺でのチェック

右辺の一つの品詞内での文法規則だけでは書けない構文的な制約あるいは変数により受け取る部分構造を使っての意味的な制約を記述する。

b)文法左辺でのチェック

右辺の複数の品詞に跨がる文法規則だけでは書けない構文的な制約あるいは意味的な制約を記述する。



(構文木)

(構文解析プログラムの処理例)

前述の例題を構文解析プログラムで解析した結果を下記に示す。

```
Parser: (parse "文"  "Dempster-Shafer理論について調べたい")

 Count : 1 Morphene
| special[(SPECIAL "Dempster-Shafer理論")]Dempster-Shafer理論 | 助詞[(NITUITE)]
について | 動詞[(SIRABERU "未然")]調べ | 要求助動詞[(TAI)]たい |
Exit(E) Next Morphene(N) Parser(P)>> p
Exit(E) Next Morphene(N)>> n

 Count : 2 Morphene
| special[(SPECIAL "Dempster-Shafer理論")]Dempster-Shafer理論 | 助詞[(NITUITE)]
について | 動詞[(SIRABERU "連用")]調べ | 要求助動詞[(TAI)]たい |
Exit(E) Next Morphene(N) Parser(P)>> p

 Count : 1 Parsing Tree
文..((ACTION RETRIEVE) (OBJECT (SUBJECT (SPECIAL "Dempster-Shafer理論"))))
   検索指示文..((ACTION RETRIEVE) (OBJECT (SUBJECT (SPECIAL "Dempster-Shafer理論
"))))
      対象指示部..(SUBJECT (SPECIAL "Dempster-Shafer理論"))
         全体概念..(SPECIAL "Dempster-Shafer理論")
            概念..(SPECIAL "Dempster-Shafer理論")
               単概念..(SPECIAL "Dempster-Shafer理論")
                  special..(SPECIAL "Dempster-Shafer理論")..Dempster-Shafer理論
         助詞..(NITUITE)..について
      要求述部..((SIRABERU "連用") (TAI))
         動詞..(SIRABERU "連用")..調べ
         要求助動詞..(TAI)..たい
Exit(E) Next(N)>>n
Exit(E) Next Morphene(N)>> n
1
(((ACTION RETRIEVE) (OBJECT (SUBJECT (SPECIAL "Dempster-Shafer理論")))))
("")
Parser:
```

問い合わせ文の解析における文法の記述例

```
／＊　文　⇨　検索指示文　＊／
  (defgram ("文" bun t)   ("検索指示文" bun t))

／＊　文　⇨　理解正否処理文　＊／
(defgram ("文" bun t)   ("理解正否処理文" bun t))

／＊　文　⇨　未知語処理文　＊／
(defgram ("文" bun t)   ("未知語処理文" bun t))

／＊　文　⇨　要求限定文　＊／
(defgram ("文" bun t)   ("要求限定文" bun t))

／＊　検索指示文　⇨　対象指示部・要求述部　＊／
(defgram ("検索指示文" (list (list 'action 'retrieve)
                             (list 'object  target)) t)
         ("対象指示部" target t)
         ("要求述部" dummy t))

／＊　検索指示文　⇨　間投詞・対象指示部・要求述部　＊／
(defgram ("検索指示文" (list (list 'action 'retrieve)
                             (list 'object  target)) t)
         ("間投詞" dummy1 t)
         ("対象指示部" target t)
         ("要求述部" dummy t))

／＊　対象指示部　⇨　全体概念・助詞　＊／
(defgram ("対象指示部" (cons 'subject (list concept))
                       (check-taishounosiji concept))
         ("全体概念" concept t)
         ("助詞" dummy t))

／＊　全体概念　⇨　type　＊／
(defgram ("全体概念" concept t)   ("type" concept t))

／＊　全体概念　⇨　概念　＊／
(defgram ("全体概念"  concept t)   ("概念" concept t))

／＊　全体概念　⇨　概念１　＊／
  (defgram ("全体概念"  noun t)   ("概念1" noun t))
```

```
／＊　全体概念　⇨　単概念　＊／
(defgram ("概念" noun t)　("単概念" noun t))

／＊　全体概念　⇨　連体詞・概念　＊／
(defgram ("概念" noun t)　("連体詞" dummy t)　("概念" noun t))

／＊　単概念　⇨　special ＊／
(defgram ("単概念" sterm t)　("special" sterm (or (equal (car sterm) 'special)
                                                    (equal (car sterm) 'unknown))))

／＊　単概念　⇨　general ＊／
(defgram ("一般概念" gterm t)　("general" gterm (equal (car gterm) 'general)))

／＊　一般概念　⇨　一般概念・単純限定・一般概念　＊／
(defgram ("一般概念" (make-subcase "単純限定" cspt2 cspt1) t)
        ("一般概念" cspt1 t)
        ("単純限定" dummy t)
        ("一般概念" cspt2 t))

／＊　単概念　⇨　special ・general ＊／
(defgram ("単概念" (list 'special (list (cadr sterm) (cadr gterm))) t)
        ("special" sterm (or (equal (car sterm) 'special)
                             (equal (car sterm) 'unknown)))
        ("general" gterm (equal (car gterm) 'general)))

／＊　概念　⇨　概念・併記・概念　＊／
(defgram ("概念" (make-subcase "併記" cspt1 cspt2)
                 (equal (depth cspt1) (depth cspt2)))
        ("概念" cspt1 t)
        ("併記" dummy t)
        ("概念" cspt2 t))

／＊　概念　⇨　概念・単純限定・概念　＊／
(defgram ("概念" (make-subcase "単純限定" cspt2 cspt1) t)
        ("概念" cspt1 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place)
                               (equal (car cspt1) 'time))))
        ("単純限定" dummy t)
        ("概念" cspt2 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place)
                               (equal (car cspt1) 'time)))))
```

```
／＊　概念　⇨　概念・単純限定・一般概念　＊／
(defgram ("概念" (make-subcase "単純限定" cspt2 cspt1) t)
        ("概念" cspt1 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place)
                               (equal (car cspt1) 'time))))
        ("単純限定" dummy t)
        ("一般概念" cspt2 t))

／＊　概念　⇨　time・単純限定・概念　＊／
(defgram ("概念" (list "時間限定" cspt2 cspt1) t)
        ("time" cspt1 t)
        ("単純限定" dummy t)
        ("概念" cspt2 (equal (car cspt2) 'general)))

／＊　概念　⇨　place ・場所の限定・概念　＊／
(defgram ("概念" (list "場所の限定" cspt2 cspt1) t)
        ("place" cspt1 t)
        ("場所の限定" dummy t)
        ("概念" cspt2 (not (or (equal (car cspt2) 'general)
                               (equal (car cspt2) 'place)
                               (equal (car cspt2) 'time)))))

／＊　概念　⇨　概念・対象の限定・概念　＊／
(defgram ("概念" (list "対象の限定" cspt2 cspt1) t)
        ("概念" cspt1 t)
        ("対象の限定" dummy t)
        ("概念" cspt2 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place)))))

／＊　概念　⇨　概念・手段の限定・概念　＊／
(defgram ("概念" (list "手段の限定" cspt2 cspt1) t)
        ("概念" cspt1 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place))))
        ("手段の限定" dummy t)
        ("概念" cspt2 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place)))))
```

```
／＊　概念　⇨　概念・目的の限定・概念　＊／
(defgram ("概念" (list "目的の限定" cspt2 cspt1) t)
        ("概念" cspt1 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place))))
        ("目的の限定" dummy t)
        ("概念" cspt2 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place)))))

／＊　概念　⇨　概念・状況の限定・概念　＊／
(defgram ("概念" (list "状況の限定" cspt2 cspt1) t)
        ("概念" cspt1 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place))))
        ("状況の限定" dummy t)
        ("概念" cspt2 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place)))))

／＊　概念　⇨　概念・因果関係・概念　＊／
(defgram ("概念" (list "因果関係" cspt2 cspt1) t)
        ("概念" cspt1 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place))))
        ("因果関係" dummy t)
        ("概念" cspt2 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place)))))

／＊　概念　⇨　概念・階層関係１・概念　＊／
(defgram ("概念" (list "階層関係1" cspt2 cspt1) t)
        ("概念" cspt1 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place))))
        ("階層関係1" dummy t)
        ("概念" cspt2 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place)))))

／＊　概念　⇨　概念・階層関係２・概念　＊／
(defgram ("概念" (list "階層関係2" cspt1 cspt2) t)
        ("概念" cspt1 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place))))
        ("階層関係2" dummy t)
        ("概念" cspt2 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place)))))
```

```
／＊　概念１　⇨　概念・対象の指示・概念　＊／
  (defgram ("概念1" (list "対象の指示" cspt2 cspt1) t)
        ("概念" cspt1 (not (or (equal (car cspt1) 'general)
                               (equal (car cspt1) 'place))))
        ("対象の指示" dummy t)
        ("一般概念" cspt2 t))

／＊　一般概念　⇨　time・単純限定・一般概念　＊／
(defgram ("一般概念" (list "時間限定" cspt2 cspt1) t)
        ("time" cspt1 t)
        ("単純限定" dummy t)
        ("一般概念" cspt2 t))

／＊　要求述部　⇨　動詞・要求助動詞　＊／
(defgram ("要求述部" (list verb dummy3) t)
         ("動詞" verb (equal (cadr verb) "連用"))
         ("要求助動詞" dummy3 t))

／＊　要求述部　⇨　形容詞　＊／
(defgram ("要求述部" dummy t)   ("形容詞" dummy (equal (second dummy) "終止")))

／＊　要求述部　⇨　補助動詞　＊／
  (defgram ("要求述部" dummy t)   ("補助動詞" dummy t))

／＊　要求述部　⇨　連用述部・要求述部　＊／
(defgram ("要求述部" (list dummy1 dummy2) t)
        ("連用述部" dummy1 t)
        ("要求述部" dummy2 t))

／＊　連用述部　⇨　動詞・接続助詞　＊／
(defgram ("連用述部" (list verb dummy) t)
        ("動詞" verb (equal (cadr verb) "連用"))
        ("接続助詞" dummy t))

／＊　連用述部　⇨　名詞・助詞　＊／
(defgram ("連用述部" (list noun dummy) t)
        ("名詞" noun (equal (car noun) 'bunken))
        ("助詞" dummy t))
```

```
／＊　理解正否処理文　⇨　正否回答　＊／
(defgram ("理解正否処理文" (list (list 'action 'answer)
                          (list 'response  reply)) t)
        ("正否回答" reply t))

／＊　理解正否処理文　⇨　正否回答・対象指示部・要求述部　＊／
(defgram ("理解正否処理文" (list (list 'action 'answer)
                          (list 'response  reply)
                          (list 'alternative target)) t)
        ("正否回答" reply (equal reply 'negative))
        ("対象指示部" target t)
        ("要求述部" dummy t))

／＊　未知語処理文　⇨　対象指示部・未知語述部　＊／
(defgram ("未知語処理文" (list (list 'action 'explain)
                             (list 'meaning  target)) t)
        ("対象指示部" target t)
        ("未知語述部" dummy t))

／＊　未知語述部　⇨　限定名詞・記述助動詞　＊／
(defgram ("未知語述部" (list verb dummy3) t)
        ("限定名詞" verb t)
        ("記述助動詞" dummy3 t))

／＊　未知語述部　⇨　終助詞　＊／
(defgram ("未知語述部" (list dummy3) t)   ("終助詞" dummy3 t))
```