LINGOL　コンパイラ

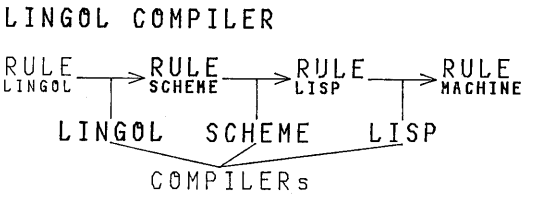
元吉　文男
（電子技術総合研究所）

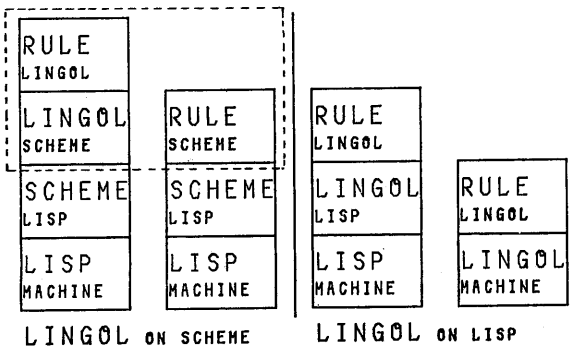
□　はじめに

我々のところでは自然言語処理を行う際に拡張LINGOLを使用してきた。この拡張LINGOLは、文脈自由文法で記述された規則をもとに構文解析を実行するプログラムであるLINGOLに、規則を適用しようとするときに、その規則を適用してよいかどうかを判断するプログラムを起動できるようにしたものである。

拡張LINGOLはLISPで記述されており、文法規則に従って、入力された文字列を構文解析するが、これは次のようにみることもできる。LINGOLはプログラム言語であり、文脈自由文法で記述された規則はLINGOL言語で書かれたプログラムであり、LINGOLが文法規則を「インタープリト」しながら構文解析を実行していると考えることができる。このように考えると文法規則を「コンパイル」すれば、今までのものより高速に構文解析を行うプログラムができることになる。この場合、目的コードはLISPのプログラムとなる。

ここで紹介するのは、文法規則からLISPの方言であるSCHEMEのプログラムを生成するものである。現在のところは、SCHEMEはLISPで記述されたものしかないが、機械語で記述されたSCHEME言語処理系は現在作成中であるので、LISPと同等のレベルでSCHEMEを使用することができるようになる。また、SCHEMEのプログラムをLISPのプログラムに「コンパイル」することも可能でこれは別に発表する。このLISPのプログラムをLISPのコンパイラにかけると、結局、機械語で書かれた構文解析のプログラムができることになる。この次々にコンパイルされる様子を右の図に示しておく。





なお、上に述べたことを模式的に書くと、右図のようになる。この図では一つの箱が一つの言語処理系に対応していると考える。小さな文字で示された言語で大きな文字で示された言語を記述していると考える。コンパイルするとは、二つの上下に接している箱を一つの箱にすることである。大まかにみると、箱を積み上げた高さが、処理に要する時間に比例すると考えることができる。

□ LINGOL

ここで簡単にLINGOLの動作を説明することにする。LINGOLでは構文解析を始める前にまず「辞書」と「文法規則」と呼ばれる構造を読み込んでおく。「辞書」は、文脈自由文法で記述されたもののうち、終端記号を含むものである。(ただし、その終端記号が直接に上のカテゴリーに連なるものだけをいう。このカテゴリーを「品詞」と呼ぶことにする。「文法規則」は、辞書記述で使用した品詞を終端記号だと思って文脈自由文法で構文規則を記述したものである。文法規則の記述例を右に示しておく。これは

```
BUN ⟶ S  END
  S ⟶ V
  S ⟶ ADJ
  S ⟶ NP S
 NP ⟶ N  P
 NP ⟶ S  NP
```

```
1  (BUN ((S NIL) (END NIL)) T)
2  (S ((V NIL)) NIL)
3  (S ((ADJ NIL)) NIL)
4  (S ((NP NIL) (S NIL)) NIL)
5  (NP ((N NIL) (P NIL)) NIL)
6  (NP ((S NIL) (NP NIL)) NIL)
```

という文脈自由文法による記述をLINGOLに入力する形で書いたものであり、簡単な日本語の文法を表わしているつもりである。この図でNILと記述されているところは、拡張したLINGOLで使用したり、でき上がった構文解析木から「意味解釈」を行ったりするときのために、プログラムを書いておくことができるが、ここでは省略する。

構文解析を行うときには、入力文字列を左から順に調べて、辞書にエントリーがあるとその品詞に文法規則を適用させて構文解析木を大きくしていく。この方法はボトムアップ法であるが、単なるボトムアップ法では効率が悪くなるので、現在どのカテゴリーが必要とされているかを常に覚えておいて、辞書を引くときや文法規則を適用するときにむだな操作をしないようにしている。例えば上の例ではPがゴールのときには、辞書を引くところでNやV等の品詞になるものは調べる必要のないことがわかる。これを行うためには、文法規則が与えられた時点で、あるカテゴリーが与えられたゴールにつながるかどうかを調べておけばよい。それには、それぞれのカテゴリーについてそれをトップダウンに展開したときに左端に出現する可能性のあるカテゴリーを集めておけば、それ以外のカテゴリーが表われたところでゴールにはたどりつけないことがわかるので、余分の探索をしないですむ。

例として上の文法規則が与えられたときに、Sをゴールにするカテゴリーを調べてみることにする。まず規則2によりVが求まる。規則3によりADJも求まる。この二つはこれ以上展開できないので先へは進まない。規則4からはNPが求まるが、このNPはさらに展開することができる。規則5、6からそれぞれN、Sが求まる。Nはこれ以上展開できないし、Sは現在調べているゴールなので再帰的になりこれ以上調べても新しいカテゴリーは見つからない。結局Sをゴールとするものは、V、ADJ、NP、N、Sであり、これ以外のものがでてきたときにはそこから先の探索は必要のないものなので省略することができる。

なお、もとのLINGOLでは以上の探索を横型に行っているが、これを日本語処理用に自動的に単語の切りだしを行うようにしたものでは、単語の同じ切り方に対する探索は横型で行うが、別の切り方を試すときには縦型探索で行っている。

後で述べるコンパイラでは簡単のために全部を縦型に探索するようなコードを出力するようにしている。普通はこの方法で実行しても結果に変りはないが、辞書にない「未定義語」まで処理させようとして、未定義語の数を最小にするように構文解析をしようとすると、縦型探索では単純なバックトラック法ではうまく実行することができない。さらに未定義語の位置を決めるのに余分の時間を必要とするようになる。

□ SCHEME

ここではSCHEMEの簡単な紹介をしておく。SCHEMEはLISPの方言であるが、LISPと最も異っているのは静的スコープを採用していることである。このために変数の束縛にA-listを使用してもその深さは静的に定まった一定のレベルを超えることがなく、それほど効率を落とさずにすむ。またコンパイルすれば、Algol流のディスプレイを使用することも可能でさらにshallow bindingとの差を埋めることが可能となる。

A-listによる変数束縛を利用しているために、SCHEMEでは関数引数の問題として知られている環境問題は解決される。これに関連して関数引数の書き方がLISPとは異っている。LISPでは

(FUNCTION (LAMBDA (X) ...))

と書くところをSCHEMEでは

(LAMBDA (X) ...)

と書いてLAMBDAを関数の一種と考えるようになっている。このほうが自然であると見ることもできる。というのは、この式を関数部分に書くときには、LISPでも

((LAMBDA (X) ...) A)

と書いて

((FUNCTION (LAMBDA (X) ...)) A)

とは書かないので、全部にFUNCTIONを付けないで書くという書式は統一がとれている。

次の大きな相違点は、SCHEMEではEVALにおいて関数の場合に関数の部分を1回、しかも1回だけEVALするようになっているところである。このために、CAR, ATOMなどの基本関数でもそのアトムの値として関数本体を持っていることである。ここで一般には関数本体とは、関数の定義とそれを評価する環境との対のことである。このために次のような使い方もできる。

((LAMBDA (U CAR) (CAR U)) '(A . B) CDR)

この答はSCHEMEではBとなるが、この方が入計算に忠実であるといえるのではないかと思う。

次は関数としてLISPのFEXPRに相当するものがなくLAMBDA, QUOTE, IF, LABELS, CATCHだけがこれと似た働きをする。それ以外は全てマクロで定義して使うようになっている。(なおSCHEMEではCONDではなくてIFが組込みの条件分岐用の式である。)従ってPROGなどもないので利用者がマクロで定義するか、組込みのマクロであるLETやDOを使用しなければならない。

LISPとの主な違いは以上であるが、関数引数がフルにサポートされているために、この機能を十分に利用してコンティニュエーションとして使用すると今ま

でのLISPでは使わなかった方法が使えるようになる。

たとえば、値を複数返す関数は、LISPではリストにして返すか、副作用を利用するかして返していたが、コンティニュエーションとして多引数のLAMBDAを使用することによって実現できる。右の図は多値関数の例としてフィボナッチ数を計算するSCHEMEのプログラムである。LINGOLコンパイラで生成されるプログラムでは、このようなコンティニュエーションを利用しているので、この例を詳しく説明しておくことにする。FIBという関数は内部でFIB1という補助関数を利用しているが、そのために2行目のLABELSでFIB1を定義して10行目でFIB1を呼びだしている。FIB1は2引数であるが、第二引数をコンティニュエーションとして使用している。この例の10行目を読むときには、FIB1は1引数だと思い、2つの値がAとBに返されると思うのが理解し易い。同様に6~9行目ではFIB1を呼んで元の値をAとBにもらい、A+BをAを値として帰すと読むことができる。(コンティニュエーションの関数——今の例ではC——を実行することは値を返すことだと思えばよい。)

```
(DEF FIB (N)                                          1
  (LABELS                                             2
    ((FIB1                                            3
       (LAMBDA (N C)                                  4
         (IF (LESSP N 1) (C 1 0)                      5
             (FIB1                                    6
               (SUB1 N)                               7
               (LAMBDA (A B)                          8
                 (C (PLUS A B) A)))))))))             9
    (FIB1 N (LAMBDA (A B) A)))))                      10
```

上の例でコンティニュエーションとしての使用法は理解できたと思うが、このコンティニュエーションを利用すると、PROLOG等で使用している後戻り操作も容易に記述することができる。それには、関数を呼ぶときに2つのコンティニュエーションを渡してやり、成功したときと失敗したときとで「戻ってくる場所」が異なるようにして、しかも元のコンティニュエーション自身が別のコンティニュエーションを引数として受け取るようにしておくと、2つの関数をコルーチン的に動作させて後戻り操作を行うことができる。LINGOLで縦型探索を行うときの後戻り操作を、この2つのコンティニュエーションを利用して行うプログラムを生成するのが次で述べるLINGOLコンパイラである。

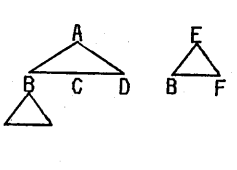
□ LINGOLコンパイラ

LINGOLコンパイラはLINGOLプログラム(これはLISP)が文法規則を見ながら構文解析を行っていく部分を、それと同じ働きをするプログラムを生成するように作ればよい。

具体的には、まず文法規則を読み込みながら、文脈自由文法で記述した規則の右辺の第1項の同じものをまとめておく。下の例では点線で囲んだ部分が、規則の右辺の初項がBであるものである。全部を読み込んだ時点で、まとめた項のそれぞれについて関数を作り出して行くことになる。右図の例で、関数Bの中で行なうことを説明することにする。Bが呼び出されると、Cをゴールとして新たな探索を開始するための関数

```
関数B
1   Cをゴールとして探索
2   Dをゴールとして探索
3   Aを呼びだす
1'  Fをゴールとして探索
3'  Eを呼びだす

A → B C D
E → B F
```



を呼ぶ。このとき、その探索が成功したときに実行するコンティニュエーションと失敗したときに実行するコンティニュエーションを引数として渡してやる。失敗したときには、1'以下の処理をするようなコンティニュエーションになっている。成功した場合には2以下を実行するようになっているが、その場合に引数として2つのコンティニュエーションを受け取り、ここで失敗が起った場合には1の「Cをゴールとして探索する部分」まで後戻りできるようにしておく。次に2のDをゴールとして探索を開始する。ここでも1と同様に2つのコンティニュエーションを渡す。この場合失敗したときのためには2の最初で受けとった「Cをゴールとして探索する部分」に後戻りするコンティニュエーションを渡す。成功した場合のためには3以下を実行するものを渡す。これも成功してるで1つのコンティニュエーションを受けとると、今度は文法規則の左辺にあるAという関数を呼び出す。そのときには、やはり2つのコンティニュエーションを渡す。以上のようにして、Aの文法規則を実行するプログラムができたことになる。1'以下はAの探索が全て失敗した場合に起動されるようになっており、実行することはAの規則の場合と同じである。

上の例の説明からわかるように、Bという関数はAを呼び出したところと同様の場所から呼ばれるので、引数として2つのコンティニュエーションを取る。このほかに現在のゴールのカテゴリーと構文解析木を作るための部分解析木の2つの計4つの引数を取る。上の説明ではゴールがどこで達成されたかがわからないが、これにはBが呼ばれた直後に現在のゴールがBであるかどうか調べて、等しければすぐに、引数として受け取った成功した場合のコンティニュエーションを起動する。このときに渡す失敗した場合のコンティニュエーションは、1以下の探索をするようにしなければならないことに注意してほしい。

さらに前節でも述べたように、あるカテゴリーがゴールにつながる可能性があるかどうかを調べるための情報を利用すると効率よく構文解析をすることができる。前節ではゴールとなるカテゴリーについてそれにつながる可能性のあるものを集めたが、ここでは逆にあるカテゴリーから行きつけるゴールを集めたほうがプログラムが容易になる。前と同じ右上の文法規則を例にとると、Sから行きつく可能性のあるものは、NP、S、BUNとなる。

これらの情報を利用して、実験にLINGOLの規則をコンパイルしたものが右下の図である。実線で囲んだ部分の規則に対応するものが、その下のプログラムになっている。

```
(BUN ((S NIL) (END NIL)) T)
(S ((V NIL)) NIL)
(S ((ADJ NIL)) NIL)
(S ((NP NIL) (S NIL)) NIL)
(NP ((N NIL) (P NIL)) NIL)
(NP ((S NIL) (NP NIL)) NIL)
```

```
(Ⓢ ((NP NIL) (S NIL)) NIL)

(DEF NP (G PO SUC FAILO)
  ((LAMBDA (FAILX)
     (IF (EQ G 'NP) (SUC PO FAILX)
         (FAILX)))
   (LAMBDA NIL
     (IF (MEMQ G '(BUN S NP))
         (GOAL
          'S
          (LAMBDA (P3 FAIL4)
            (Ⓢ G (LIST Ⓢ NIL
                       (LIST PO P3) NIL)
               SUC FAIL4))
          FAILO)
         (FAILO)))) )
```

前に述べたことからわかるように，この規則だけからプログラムが生成されるのではなく，全体の文法規則を読み込んで，ゴール関係を調べたあとでプログラムの生成が行われる。（なお，この例ではSが規則の両辺に表われていて，どちらのSかわからなくなるので，文法の左辺にあるSは丸印で囲んでおいた。）

なお，このコンパイルされたプログラムを走らせるには，GOALという実行時関数を必要とするが，このGOALで，入力された文字列の切り出しを行なって辞書を引き，辞書に見つかったカテゴリー名の関数を呼び出す操作を行っている。参考までにその定義を右に示しておく。この中で使用されている関数についての説明は省略する。

```
(DEF GOAL (G SUC FAIL)
   (LABELS
    ((GOAL1 (LAMBDA (LS)
              (IF (NULL LS) (FAIL1) (GOAL2 (JISHOBIKI LS) LS))))
     (GOAL2 (LAMBDA (DICTS LS)
              (IF (NULL DICTS) (GOAL1 (CUTLAST LS))
                  ((GETOP (CAAR DICTS))
                   G
                   (CAR DICTS)
                   (LAMBDA (P FAIL2) (SUC P FAIL2))
                   (LAMBDA () (GOAL2 (CDR DICTS) LS))))))
     (FAIL1 (LAMBDA () (RESAVE) (FAIL))))
    (GOAL1 (GETFIRST))))
```

□ PROLOG

以上の説明のように，文脈自由文法をボトムアップに構文解析するには，文法規則の右辺の第一項を関数名として，第二項以後をゴールとして探索して，それらが全て成功したときに，左辺のカテゴリー名の関数を呼び出すようにすればよい。PROLOGでは後戻り操作を自動的に行うので，LINGOLの規則からPROLOGのプログラムは容易に作成できる。前と同じ文法規則をPROLOGになおしてみるとSに関する部分は右のように書くことができる。なお，ここでは簡単のために，部分木の部分と入力文字列に関する部分の引数については省略してある。

```
s(s).
s(G) :- goal(end),bun(G).
s(G) :- goal(np),np(G).
```

また，LINGOLのSCHEMEへのコンパイルのところでの説明から理解できるように，成功・失敗の制御の流れはPROLOGの制御の流れとほとんど同じである。異っているのは，カットがある場合であるが，全てのコンティニュエーションが見えているので，どこでも好みの場所へ後戻りをすることができる。更に関数呼出しのマッチングをする部分をつけ付えることにより，PROLOGのプログラムをSCHEMEのプログラムにコンパイル（トランスレート？）することが可能であり，実際にその変換プログラムも動作している。

□ おわりに

以上述べたようにして，LINGOLの規則をSCHEMEのプログラムにコンパイルすることができるが，さらにこれを変換してLISPのプログラムに，機械語のプログラムに落すこともできる。またPROLOGのプログラムについても同様にして機械語まで落すことが可能であり，SCHEMEの中でPROLOG的な使用をしたいところでは，PROLOG流のプログラミングが可能となる。筆者は現在SCHEMEの言語プロセッサを作成中であり，この上に問題解決システムを走らせることを計画中である。