TR-I-0360

LRパーザの応用法

田代 敏久
Toshihisa Tashiro

1993.3

概要

ATR音声翻訳システムASURAの音声認識部で利用されているLRパーザは、簡潔にまとめられた文脈自由文法解析ツールとなっており、各種の応用が可能である。本報告書では、まずLRパーザの特徴を説明し、次に、LRパーザの応用時に留意すべき点についてプログラムレベルで言及する。さらに、筆者の行なったLRパーザの応用事例を紹介する。

ATR自動翻訳電話研究所
ATR Interpreting Telephony Research Laboratories

もくじ

1　はじめに

ATR 音声翻訳システム ASURA[1] の音声認識部で利用されている LR パーザは、汎用的な文脈自由文法解析ツールとしても整備されている [2]。そのため、この LR パーザは、他の言語処理システムで使われている文法を利用したり、他の言語処理モジュールと組み合わせたりすることが容易に行なえるので、自然言語処理研究の様々な場面での有効活用が可能である。本報告書では、まず LR パーザの特徴を説明し、次に、LR パーザの応用時に留意すべき点についてプログラムレベルで言及する。 さらに、筆者の行なった LR パーザの応用事例を紹介する。

2 LR パーザの特徴

本節の目的は、LR パーザを応用するにあたっての最低限必要な知識を列挙・整理することである。LR パーザの詳しい動作原理については文献 [13, 3] 等を参照してほしい。また、一般に LR パーザといった場合、

- プログラム言語作成時に用いられる yacc 等の解を一つしか出力しないパーザ[1] (LR 文法しか受け付けないパーザ)
- 自然言語処理に用いられる一般化 LR パーザのように、曖昧性のある文法の場合は全ての解を出力可能なパーザ

に二分されるが、ここでは特に断りのないかぎり後者を想定する。

2.1 完全統制解析

LR は完全統制解析を可能にしており、非常に (時間的にも空間的にも) 効率のよい文脈自由文法解析アルゴリズムである。そのため各種の実験を手軽に行うことができる。特に他の大きなプログラムと組み合わせて使用する際、空間的に効率のよいことは大きな利点である。なお完全統制とはどういうものかについては、文献 [9] にわかりやすい解説がある。

2.2 簡潔なアルゴリズム

LR パーザは

- バックトラックしない (同じ計算を繰り返さない)。
- 適用すべき文法規則の予測はコンパイル時に前もって行なわれいる。

という特徴があるため、入力を左から右に一回だけ走査し、解析を決定的 (deterministic) に行なうことができる。この簡潔なアルゴリズムが解析の過程の見通しを良くし、ビームサーチのような直感的に理解しやすい枝刈り手法の導入を可能にしている。、しかし、これにも欠点があり、ベストファーストサーチやその他のヒューリスティックな探索を行なうのが困難になっている[2]。

[1] yacc では文法に曖昧性を許さないわけではなく、曖昧性がある場合には優先順位を付けることにより単一の解を出力するようになっている。

[2] スタックや入力の状態を保持しておく機構を設ければ可能であるが、プログラムの簡潔さや効率が損なわれることは避けられない。やはりこのような制御にはアクティブチャートパーザの方が向いていると思われる。

2.3 解析可能な文法のクラス

LR パーザは、 Earley パーザ [15] のようにすべての文脈自由文法をカバーしているわけではない。しかし、LR パーザは文脈自由文法のほとんどを解析することが可能である。解析対象外は次の 2 つのみであり、どちらもわずかな修正により解析可能となる。

- ϵ 規則 (空生成規則)
  これは、単に ϵ 規則を扱えるテーブルジェネレータを用意すれば解決できる問題である。ただし、規則の組み合わせによってはループが生じたり、非常に効率の悪い規則体系になる可能性があるが、これはまた別の問題である。

- 周期的 (cyclic) な規則の集合
  この場合、LR テーブルは正確に作られるので、パーザにループ検出機構を用意したり、 cfg 規則の評価時に副作用を起こす手続きを組み込む (拡張 CFG パーザにする) ことにより、プログラムの停止を保証すればよい。

3 LR パーザの応用時の留意点

本節では、LR パーザの応用時の留意点について、実際のプログラムの実装方法に触れながら説明する。解析の起動の方法や、文法のコンパイルの方法等については、文献 [2] を参照してほしい。なおこの章で扱うプログラムはこの文献に説明のある音韻列パーザを想定している。重要な点については巻末にプログラムのソースリストを掲載する。

3.1 終端・非終端記号の取り扱い

オリジナルの音韻列パーザは終端記号としてアルファベットを想定している。応用時には様々な終端・非終端記号を取り扱う必要があるが、その際に留意すべき点について説明する。

3.1.1 漢字を扱う場合

プログラムの中では、文法中のすべての記号はハッシュされて管理されている。端・非終端記号に漢字 (shift-jis, euc 等の 8bit 系漢字コード) を利用する場合、オリジナルのハッシュ関数は 8bit-code に対応していないのでこれを改める必要がある。またテーブル作成プログラム (slr) も同様なハッシュを行なっているので、これも改造する必要がある。なお、ハッシュ関数は完全ハッシュテーブルを作成できればどんなものでもよい。

3.1.2 終端記号の単位を変更する場合

オリジナルのプログラムは、入力は終端記号単位で空白によって区切られていることを想定している。これは入力をトークンに分解する際に空白を手掛かりにしているためであり、トークンに分解する機構さえ設ければどのようにも対応できる。漢字を含む日本語テキストを解析する場合には、単純に一文字づつ (2byte づつ) 分解するようにし、テキストには変更を加えない用にした方が利用しやすいだろう。

3.1.3 字句解析ツール (Lexical Analyser) の必要性

文字列以外の入力を解析する必要が生じる場合もある。例えば言語データベースの品詞情報を利用し、品詞を終端記号として解析を行ないたい場合等である。そのための変更自体は容易であるが、本格的にツールとして整備するためには、unix の yacc のようにパーザ本体と字句解析ルーチンを分離し、様々な入力に対して、字句解析部だけの変更で対応できるようにするのが望ましい。

3.2 統計処理用ツールのフロントエンドとしての使用

LR パーザーを各種の統計処理プログラムのフロントエンドとして利用することも可能である。これには大きく分けて二つの場合が考えられる。

3.2.1 CFG 規則の統計処理

確率文法や文法規則の適用過程の N-gram を計算する際には、解析時に適用された文法規則のリストを出力する必要がある。このためには還元動作時に使用された文法規則をスタック[3]に保持しておけばよい。LR パーザの解析過程は、最右導出の逆向きの作成過程であるので, 解析終了後にこのスタックの先頭から取り出したリストが、その解析木の最右導出過程になっている。最右導出以外の導出過程を求めるには一旦構文解析木を作成し (オリジナルのプログラムはデフォルトでは解析木を作成しない)、その木をトラバースすればいい。なお、解析木の表示ツールは最右導出のリストから構文解析木を作成し、最左導出過程に従い解析木のノードを表示することで実現している。

3.2.2 形態素の統計処理

品詞の N-gram を計算する際等には、品詞が割当てられた終端記号列を得る必要がある。これは形態素解析を行なうことと等価であり、LR パーザで実現できる[4]。この場合もやはり解析木を一回作成し、解析木の葉 (leaf) とその親を抽出すればよい。

3.3 拡張ＣＦＧ解析ツールとしての使用

オリジナルの LR パーザは純粋な CFG パーザ (明示的に解析木を作成しないことを考えると、単なる CFG-recognizer といってよい) であるが、自然言語を扱うには CFG の記述能力では普通足りない。CFG 以上の言語クラスを解析するための方法としては、

- 文脈依存文法を利用する
- 拡張文脈自由文法を利用する

[3]解析状態のスタックとは別であることに注意。

[4]本来は、形態素解析結果を得るためだけに CFG による構文解析を行なうのは無駄である。しかし、品詞や単語の区切り方等に依存されない形態素解析ツールはなかなか存在しない (少なくとも atr 内部には無い) のに対し、LR パーザは単純な CFG 規則さえ用意すれば簡単に利用できる。後の述べる部分木解析と組み合わせれば、効率もさほど悪くはない。

の2つが考えられるが、一般的には拡張文脈自由文法を利用することが多い。拡張文脈自由文法は、文脈自由文法の各規則に注釈を記述しておき、規則の書き換えの可否をチェックしたり、部分木の情報をより上位の構造へ伝播させたりすることにより実現できる[5]。

LR パーザで拡張文脈自由文法を扱うためには、

- 還元動作 (reduce) 時に何らかの副作用をもたらす手続きを起動する。その結果、句構造規則の適用が拒否されれば、その場で枝刈りを行なう。
- 解析終了後の作成後、解析木をたどりながら副作用を伴う処理を行なう。その結果、作成された解析木が拒否されることもある。

の2つの方法が考えられ、通常は後者の方法がより効率がよい。しかし、文法規則の性質によっては、後者の方法では曖昧性の数が急激に増えてしまうこともある。このような句構造規則とその注釈の評価時期による効率の変化に関しては文献 [11] が詳しい。また、ATR の LR パーザは、文献 [3] に述べられているようなノードのパッキング (共通の部分文字列を範囲とする同じ非終端記号を持つ部分木を共有すること) を行なっていないため、後者の方法のデメリットがより大きくなる場合もある。このノードパッキングと句構造規則の注釈の評価との関係については文献 [12] に説明がある。

3.4 部分木解析の方法

LR パーザは適格部分文字列 (well-formed substring) を保持しない。より大きな部分木に組み上がれなかった部分木のデータ構造は再利用されてしまう。LR パーザは解析過程が固定されているため、ある時点で部分木が全体の木の作成に貢献しないことがわかれば、その木を他の部分木が必要とすることはありえないためである。LR パーザはこの仕組みにより記憶領域を節約しているわけであるが、結果としてせっかく計算した適格部分文字列の情報を捨ててしまうことになる[6]。

このような理由で、本格的な部分木解析を LR パーザで行なうのは困難であるが、制限付きの部分文字列解析なら可能である。前に述べたように、LR パーザで問題なのは途中の解析結果を保持しないことであるから、これを改めればよいのである。 具体的には、次のような手続きで行なえばよい。

---

[5] CFG の各規則に注釈を記述しておくという方法は、自然言語処理に限るわけではない。本来はプログラム言語のコンパイラ作成時の技術 (構文主導定義) である [13]。

[6] これに対し、アクティブチャートパーザでは一旦チャートに登録された情報は捨てられることはない (というより、解析過程の制御が自由なので捨ててしまうわけにはいかない) ので、解析終了後、文として解析木が得られたかどうかにはかかわり無く、すべての適格部分文字列の情報を得ることができる。

- 還元動作時に、保持すべき部分木であるかどうかを判断する。例えば、名詞句は保持すべきだが、副詞句は残さなくてもよい、 等と判断する。すべての部分木を保存するのは効率を著しく落としてしまうだろう。

- 保持すべきと判断したら、その時の解析の状態 (スタックや適用された文法規則のリスト) をコピーし、テーブルに保存しておく。

- 文としての解析が失敗したり、得られた解析木が正しくない時には保持しておいた部分木の中から適当なものを選ぶ。必要なら得られた部分木に含まれない文字列を再パースしてもよい。

より実際的な応用としては、作成された構文木入力のすべてをカバーしていれば、どんな非終端記号の木になっていても解析成功とみなしてしまうという方法がある。入力を適当に分割する仕組みを設け、パーザはその分割された入力を処理すればよい。この場合、解析過程のコピーは文末まで解析が進んだときまでは行なわないので、それほど効率を落とさずに実現することができる。

3.5 形態素切りデータ作成ツールとしての使用

日本語の漢字かな混じり文を解析する際には、完全な形態素結果ではなく、単に単語単位で切り分けられたデータが有効なことも多い。その場合前述した形態素解析の方法に加え、同じ単語切りになっている解析結果をマージすることにより、効率的に作業を行なうことが可能である。

4 LR パーザの応用事例

本節では、筆者が行なった LR パーザの応用事例を紹介する。

4.1 単一化に基づく LR パーザの作成

4.1.1 作成意図

日本語構文解析部で使われているアクティブチャートパーザ [10, 4] は、

- 選言を含む素性構造が単一化可能。
- 解析過程の制御が可能。
- 弧の共有 (ノードパッキング) が可能。
- 素性記述の評価時期の制御が可能。

等、豊富な機能を持つパーザである。半面、その機能の豊富さが災いし、

- 手軽に改造を行なうのが困難。
- プログラムが巨大なため、実行可能な計算機が限定される。
- 全解出力が困難 (これはチャートパーザーが well-formed substring を保持していることに起因すると思われる)。

等の難点がある。特に、全解出力が困難であることは、文法規則の統計を取る場合等に不便である。その点、前節で述べたように、LR パーザは解析過程の制御等に難はあるものの、プログラムが簡潔で実行ファイルも小さくてすむ、という利点がある。そこで、LR パーザに単一化機構を組み込み、日本語構文解析部で用いられている単一化文法 [5] をそのまま利用できる仕組みを作ることにした。

4.1.2 実現方法

単一化 LR パーザの実現は、次のような手順で行なった。

1. 音韻列パーザの日本語対応
   これは前節で書いたとおりの方法で行なった。

2. 単一化機構のチャートパーザからの分離

   C版日本語構文解析プログラム [14] から単一化モジュールを分離した。もともと設計上は分離されていたので、わずかな修正で実現できた。

3. LR パーザと単一化機構のインターフェースの確立

   基本的には前節で述べた方法で行なった。実装時の問題点としては、チャートパーザーでは、句構造規則とその注釈をひとつのオブジェクトとして管理しているのに対し、今回は句構造規則と注釈を分離したほうが作業が容易になることが挙げられる[7]。この問題は句構造規則と対応する注釈のテーブルをパーザ内に持つことにより解決した。

4.1.3 結果

作成した単一化 LR パーザは当初の期待どおりの能力を示した。正確な測定は行なっていないが、モデル会話 1 2 会話の文法 (Mset 文法) を用いた場合、C 版チャートパーザが解析木をひとつ出力する時間内に、単一化 LR パーザは全解出力できる場合が多かった。しかし、機能試験文 600 文の文法 (Eset 文法) を用いた場合には、メモリ不足により解析不能となることが目立った。これは、LR パーザの解析効率の優位性は計算量の理論でいう定数係数を下げたことに起因しているだけなので、複雑な文法や長い入力文を与えれば、チャートパーザと LR パーザの差は無くなってしまうためだと思われる。

なお、この単一化 LR パーザは LRP という名前で一つの実行プログラムとしてまとめられており、

as23:/home/export/nadine-c/LR/BIN

に実行ファイルと README ファイルが存在しているので参照してほしい。単一化 LR パーザ (LRP) の動作例を以下に示す。

```
##############################################################################
csh > LRP こちらは会議事務局です

こちらは会議事務局です
Tree 1.
count = 1
```

[7] これは純粋に実装上の問題であって、単一化 LR パーザにおいても概念的には句構造規則と注釈はひとつのオブジェクトである。

```
<start2>
|--<sent>
   |--<v>
      |--<v-kernel>
         |--<p>
         |  |--<n>
         |  |  |--<n-anaph>
         |  |     |-- こちら
         |  |--<postp-topic>
         |     |-- は
         |--<v-kernel>
            |--<n-prop>
            |  |-- 会議事務局
            |--<auxv-copl-com>
               |-- です

(([[SEM ?X2[[RELN  だ -IDENTICAL]
            [ASPT  STAT]
            [OBJE ?X3[[LABEL  *SPEAKER*]]]
            [IDEN [[PARM ?X1[]]
                   [RESTR [[RELN  NAMED]
                           [ENTITY ?X1]
                           [IDEN  会議事務局 -1]]]]]]]
   [PRAG [[RESTR [[IN []]
                  [OUT []]]]
          [TOPIC [[IN [[FIRST [[FOCUS ?X3]
                               [TOPIC-MOD  HA]
                               [SCOPE ?X2]]]
                      [REST []]]]
                  [OUT []]]]
          [PRSP-TERMS [[IN []]
                       [OUT []]]]
          [SPEAKER ?X3]
```

```
[HEARER [[LABEL  *HEARER*]]]
[ASPE [[IN []]
       [OUT []]]]]]]]))
```

```
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
```

ＣＦＧ規則による解析木の数 = 1

単一化後の解析木の数 = 1

```
####################################################################################
```

4.2 単一化文法を用いた連続音声認識実験

4.2.1 ねらい

日本語構文解析部で使われている単一化文法を用いた連続音声認識実験のねらいは以下の２つについて検討することであった。

- 現在独立に開発されている音声認識文法 [6] と構文解析文法 [5] の将来の統合へ向けて、単一化文法を用いた連続音声認識の (システムとしての) 実現可能性。
- 単一化文法の持つ意味情報や語用論的情報を音声認識に用いることの有効性。

4.2.2 単一化文法を用いた連続音声認識プログラムの作成

単一化文法を用いた連続音声認識プログラムは、先に述べた単一化 LR パーザの作成方法と同様な手順で作成した。具体的には、以下のような手順である。

1. HMM-LR 音声認識プログラムと単一化機構のインターフェースの確立
   HMM-LR 連続音声認プログラム [7, 2] と、C 版日本語構文解析プログラム [14] から分離した単一化モジュールを接続し、還元動作時に単一化を行なうようにした[8]。
2. 日本語単一化文法の終端記号の書き換え
   音声認識時には、終端記号は音韻を表すアルファベットである必要があるので、単一化文法の終端記号を書き換えた。なおこの書き換えは単一化文法の句構造規則部分のみに適用し、句構造規則の注釈部に含まれる語形や語義の情報は漢字のまま残した。

作成した連続音声認識プログラムは正しく動作し、単一化文法を用いて音声を認識し、認識成功時には日本語意味構造を出力した。よって、実験のねらいの一つである、「単一化文法を用いた連続音声認識の (システムとしての) 実現可能性」は確認できた。

4.2.3 実験結果と考察

単一化文法を用いた音声認識実験は以下のような条件で行なった。

- 話者:MAU
- ビーム幅:100(global)、12(local)
- 文連続音声発声

[8]音声認識ではビームサーチを行なうため、解析木作成後にまとめて単一化を行なうわけにはいかない。

- モデル会話 (A,B,1 10) を対象
- 単一化 LR パーザを用いて計算した確率文法を使用。

音声認識率は以下のとおり。

・確率文法を用いない場合

1 位認識率 43.1%

5 位以内 44.0%

・確率文法を用いた場合。

1 位認識率 54.9%

5 位以内 55.7%

なお、認識時間は一文節の文でも CPU 時間で 60 秒、二文節の文で 400 秒程度かかった。三文節以上の文が認識できることはほとんど無かった。

ちなみに、日本語音声認識文法を用いた文連続音声認識 [8] では、確率文法を用いない場合、5 位以内認識率は 60% 弱、確率文法を用いた場合で 70% 程度であり、単一化を用いた場合より高い認識率を示している。

単一化文法による音声認識の性能が良くない原因としては、

- 文法の性質の違い

  音声認識文法は、(当然のことながら) 音声認識に使われることを想定して、開発・改良されているのに対し、構文解析文法は認識終了後の文字列からできるだけ早く意味構造を生成することを主眼にして開発・改良されている。そのため、構文解析文法は、音声認識文法が持つ音声認識に有効な制約 (特に局所的な制約) が充分に記述されていないと考えられる。

- 探索手法の問題

  単一化文法の利点は、制約伝播機構のおかげで、純粋な CFG では記述できない長距離の依存関係に関する制約を記述できるところにある。しかし、現在の HMM-LR 音声認識ではビームサーチを用いているため、局所的な制約が充分でない場合には、長距離の依存関係が出現する前に正しい候補が枝刈りされてしまうことが多い。単一化文法による音声認識がうまくいかない本質的な原因は、この探索手法に起因すると思われる。

以上のように、単一化文法を用いた連続音声認識実験では、システムとしての実現可能性の確認はできたものの、単一化文法が持つ意味情報や語用論的情報を音声認識で有効活用することはできなかった。しかし、LR パーザとビームサーチという手法の組み合わせの本質的な弱点が明らかになったという点では、意義のある実験であったと言ってもよいだろう。

なお、この単一化を用いた連続音声認識プログラムは Hmmlr.save という名前で一つの実行プログラムとしてまとめられており、

as23:/home/export/hmm-lr

に実行ファイルと README ファイルが存在しているので参照してほしい。

Hmmlr.save の実行例を以下に示す。

```
csh > Hmmlr.save  LBL/lblaa
[ HMM-LR Continuous Speech Recognizer, Separate-VQ Version. Unification ]
    <<< Copyright(C) 1988,1989,1990,1991 K.Kita & T.Hanazawa >>>

 ### Hmmlr.save (Unification) ###
 Created on Fri Mar 27 16:23:04 1992


 Trellis algorithm && Duration control.

 Grammar-> honyaku
 Total cells = 2000.
 ThresholdQ = 30.000000.
 Threshold1 = 20.000000.
 Threshold2 = 15.000000.
 Global beam = 100.
 Local beam = 15.
 First = 0

 Input-> LBL/lblaa
 Frame period = 9.0 msec
```

```
 Add time = 54.0 msec
 End_Free = 27.0 msec
 Mabiki = 1

 DurationQ:  9.0 - 243.0 msec

(001) MAU_MA1_01    246.0   884.0   |moshimoshi|  72 frames
*****************************************************************
Recognition time: CPU-time = 42498 msec, Elapsed-time = 104 sec.
Total-verify = 5384, Depth = 14
  1: moshimoshi                (prob =  8.82206)
  2: moshimoshiga              (prob =  9.57832)
  3: moshimoshine              (prob =  9.71281)
*****************************************************************
```

5 おわりに

本報告では、LR パーザを効果的に応用するために、知っておいた方がいい LR パーザの特徴について述べた後、LR パーザ応用時に留意すべき点についてプログラムレベルで述べ、最後に筆者の行なった LR パーザ応用事例を紹介した。

今後検討すべき点としては、以下のようなことが挙げられる。

- 共有資源としての整備
  LR パーザは改造が容易なので、ATR 内の多くの人が様々な形で LR パーザを改造して利用していると思われる。それらの改造には、汎用性の高いものも多いと予想されるので、共有資源として整備することにより、研究の効率化が可能だろう。

- LR パーザ以外の解析ツールの必要性
  今後、より自由な発話を許容する頑健な音声言語処理システムを開発する際には、LR パーザの、

  - 解析過程の制御が困難
  - 適格部分文字列を保持しない

  という弱点が次第に目立つようになると思われる。

一方、ASURA の日本語構文解析部で利用されているチャートパーザ [10, 4] は、アクティブチャート解析を行なっているので、上記のような問題は解決している。しかし、このパーザは単一化文法を効率よく解析することを目的に開発・改良されてきたために、一般的な CFG 解析ツールとして整備されていない。解析ツールとして整備されたチャートパーザは、より高度な音声言語処理研究を可能にするだろう。

参考文献

[1] 竹沢, 森元, 谷戸, 鈴木, 嵯峨山, 椹松 (1993-03): ATR 音声言語翻訳実験システム ASURA', 情報処理学会第 46 回全国大会, 6B-5.

[2] 北:HMM-LR ユーザーズ・マニュアル, ATR テクニカルレポート, TR-I-0246,1992

[3] Tomita, M.:An Efficient Context-free Parsing Algorithm for Natural Languages, Proc. 9th Intertational Joint Conference on Artifical Intelligence, 1985

[4] 永田、田代、松尾、高橋: 単一化に基づく構文解析：実践編, ATR テクニカルレポート, TR-I-0333,1993

[5] 永田、田代、衛藤、坂口: 日本語解析文法解説書, ATR テクニカルレポート, TR-I-0335,1993

[6] 永田、衛藤、保坂: 音声認識のための構文規則ガイドブック, ATR テクニカルレポート, TR-I-0240,1992

[7] 北、川端、斉藤:HMM 音声認識と拡張 LR 構文解析法を用いた連続音声認識, ATR テクニカルレポート, TR-I-0082,1989

[8] Kita,Morimoto,Ohkura,Sagayama: Continuously Spoken Sentence Recognition by HMM-LR ICSLP,1992、

[9] 松本: 統語解析の手法, 自然然言語理解, 知識工学講座 8 , オーム社,1998

[10] 小暮: 解析過程の制御を考慮した句構造文法解析機構の検討, ATR テクニカルレポート,TR-I-0064,1988

[11] Nagata, M:An Emprical Study on Rule Granuarity and Unification Interleaving Toward an Efficient Unification-Based Parsing System,Proc.COLING-92,pp.177-183

[12] Maxwell,J and Kaplan, R:The Interface between Phrasal and Functional Constraints, unpublished manuscript(1992).

[13] Aho,Sethi,Jeffrey:Compilers Principle,Techiques,and Tools,1986

[14] (株) 漢字情報サービス,C 版日本語構文解析系 (ACP) 詳細仕様設計書,ATR 内部資料,1992

[15] Earley,J:An Efficient Context-Free Parsing Algorithm,Comm.ACM,Vol.13,No.2, pp.100-105,1970

A　プログラムリスト

LR パーザの応用を行なう際に特に重要と思われる関数について、プログラムリストとコメントを載せる。

A.1　記号処理関数

本文 3.1節に関連する関数を紹介する。

なお、ここで紹介する関数のソースファイルは

as23:/home/export/lr

に存在する。

A.1.1　HashSymbol

```
/*
  8bit-code を扱えるハッシュ関数。
  文字列を入力とし、ハッシュテーブルに格納して、そのインデックスを返す
*/
HashSymbol(s)   /* Return index of null slot or index of symbol 's'. */
char    *s;
{
        char    *p;
        int     old_index, index, n;

        for(index = 0, p = s; *p != NULL; )

                /* 絶対値を取るようにして、8bit code に対応する */
                index += abs(*p++);

    retry:
        old_index = index = index % MAXSYMBOLS;
        for(n = 0; SymbolTable[index] != NULL; ++n, index = (old_index+n*n) % MAXSY
        {
                if(n != 0 && index == old_index)
```

```
            {
            warning("Bad hashsize for SymbolTable. Modify hash size!!!\n");
                    ++index;
                    goto retry;
            }
            if(strcmp(SymbolTable[index], s) == 0)
                    break;
    }
    return(index);
}
```

A.1.2 lr_parse

```
/*
  空白やタブで区切られた文字列を入力にとり、
  文字列をトークンの列に分解し、解析本体の関数である parse を呼ぶ。
  解析が成功した場合には 1 を、失敗した場合には 0 を返す。
*/
lr_parse(buf)
char    *buf;
{
        register int    i;
        register char   *p, *q;
        int     ok;
        char    *symbol[MAXSYMS];

        Cell    *cellp, *firstcell();
        Cell    *cp;

        /* 入力を正規化する。(タブをスペースに変換し、先頭の空白を詰める。)
   様々な入力を扱うためには、この位置で調整すればよい */
        format(buf);
        if(strlen(buf) == 0)
                return;

        /* 文字列の終りを示す "$" を付ける */
        strcat(buf, " $");  /* END-SYMBOL */

        /* スペースを手掛かりに文字列をトークン列に分解する。 */
        for(i = 0, p = q = buf; *p != (int)NULL; p++) {

                if(*p == ' ') {
                        *p = (int)NULL;
                        if(i >= MAXSYMS - 1) {
                                fatal_error("Input too long.\n");
```

```
                        return(0);
                }
                symbol[i++] = q;
                q = p + 1;
        }
    }
    /*
     * Start parsing.
     */
    ok = 1;
    init_cell();
    cellp = firstcell();

/* トークン毎に解析を行なう。 */
    for( i = 0 ; symbol[i] != NULL; i++) {
        if(!parse(symbol[i], &cellp)) {
            ok = 0;
            break;
        }
    }

    /* 解析成功の場合、解析木を表示する。
  ここで解析木を評価して、拡張文脈自由法に対応してもよい */
    if ( ok ) {
        print_result_trees(cellp);
    }
    return(ok);
}
```

A.2 構文解析木操作関数

解析木を作成、表示する関数を紹介する。

3.2節に関係が深い。

なお、ここで紹介する関数のソースファイルは

as23:/home/export/lr

に存在する。

A.2.1 make_result_tree

```
/*
  解析の状態を表す構造体であるセルから、LR パーザの文法規則適用過程のリスト
  （最右動出過程である）を取り出し、そのリストから木構造を作成する。
*/
struct tree *
make_result_tree( cp )
Cell    *cp;
{
        Rule    *rp;
        struct  tree    *root_tree;
        struct  tree    *next_tree;
        int     next_point;
        int     *p;
        int     rule;
        int     i;
        short   int     gra_array[GSTKSIZE];
        int     gstkptr;

        /* 最右導出過程の抽出 */
        gstkptr = cp->gstkptr;
        for ( i = 0 ; i < gstkptr ; i++ ) {
                gra_array[i] = cp->gstk[i];
        }
        rule = gra_array[gstkptr - 1];
```

```
        /* 文法規則を示すインデックス (rule) から文法規則の構造体 (rp) を得る */
        rp = GrammarTable[rule];

        /* ルートノードの作成 */
        root_tree = make_tree( SymbolTable[rp->lhs]);
        root_tree->rule = rule;
        root_tree->rhs_len = rp->length;

        /* 最右導出なので逆から回す必要がある */
        for( i = rp->length-1 ; i >= 0 ; i -- ) {
                p = rp->rhs + i ;
                /* 右側の子供を得る */
                next_point = reverse_search( *p, gstkptr - 1, gra_array);
                /* 再帰する */
                next_tree = make_result_tree_internal( *p , next_point ,
                                                       gra_array);
                root_tree->rhs[i] = next_tree;
        }
        return  root_tree;
}
```

A.2.2 print_tree

```
/*
  解析木を受けとり、最左の行きがけ順 (pre-order) にたどりながら、
  木構造を表示する。
*/
print_tree( tree )
struct  tree    *tree;
{
        char    string[512];

        if (! tree )
                return;

        /* top */
        printf( "%s \n" , tree->left_symbol );

        bzero( string, 512 );
        print_tree_internal( tree , string );
        printf("\n");
}

print_tree_internal( tree , string )
struct   tree    *tree;
char            *string;
{
        int     i;
        int     j;
        char    new_string[512];

        if (! tree )
                return;

        if ( i = tree->rhs_len ){
```

```
        /* terminal */
        if ( tree->rhs[0]->rhs_len == 0 ) {
                printf( "%s%s%s" , string, "|--",
                                tree->rhs[0]->left_symbol );
                for (j = 1 ;  j < i ; j++ ) {
                        printf( "%s" , tree->rhs[j]->left_symbol );
                }
                printf("\n");
                return;
        }
        /* nonterminal */

        /* 木を表示する際には最左導出の方が見やすい。
           左の子供から再帰的に木を辿る. */
        for ( j = 0 ; j < i ; j++ ) {
                printf( "%s%s%s\n" , string, "|--",
                                tree->rhs[j]->left_symbol );
                strcpy( new_string , string );

                if ( j < i - 1 )
                        strcat( new_string , "|   ");
                else
                        strcat( new_string , "    ");

                if ( tree->rhs[j]->rhs_len )
                        print_tree_internal( tree->rhs[j] ,new_string );
        }
    }
}
```

A.3 パーザ本体

LR パーザのパーザ本体関数 (parse) を紹介する。

A.3.1 parse

```
/*
 トークンと現在の解析状態の集合（セルのリスト）をうけとり、
 LR 解析を行なう。
 解析に成功のときは 1 を、あるいはまだ成功する可能性があれば 2 を
 もう可能性がなければ 0 を返す。
 */
parse(str, top)
char    *str;
Cell    **top;
{
        char    *p;
        Cell    *cp, *cp2, *clist, *clist2;
        Action  *ap;
        int     input, state, state2, action, n, accepted = FALSE;
        Cell    *getcopy();

        clist = *top;
        clist2 = NULL;

        /* 入力されたトークンがシンボルテーブルに登録されているか */
        if(SymbolTable[input = HashSymbol(str)] == NULL) {
            error("Bad input symbol: %s\n", str);
            goto End;
        }

        /* 一つ一つのセルに対して */
        for(cp = clist; cp != NULL; cp = cp->next) {
            /**********
             * Reducer
             **********/
            /* 現在のセルのスタックのトップを取り出す。 */
            state = StackTop(cp);
```

```
/* 動作表を引く
   なお、このパーザは動作表と行き先表をひとつにまとめている */

for(ap = ActionTable[state]; ap != NULL; ap = ap->next) {
    int     id;

    /* 動作表の一つのエントリにつき */
    action = ap->action;
    /* もし還元動作なら */
    if(IsREDUCE(action)) {

        /************************************************
        この位置で、何らかの手続き（単一化など）を用意すれば、
        還元を認めるかどうかかを調べることができる。
        還元を認めない場合には、ここで
        ループの先頭に戻る（continue）すればよい。
        ************************************************/
        /* 新たなセルを作る */
        cp2 = getcopy(cp);
        /* 文法規則の右辺の数を求める */
        n = GrammarTable[REDUCE(action)]->length;
        /* スタックから右辺の数だけポップする。 */
        PopN(n, cp2);
        /* 行き先の状態を得る */
        state2 = GotoState(ActionTable, StackTop(cp2),
                    GrammarTable[REDUCE(action)]->lhs);
        /* 新たな状態をプッシュ */
        Push(state2, cp2);

        /* 使われた文法規則を文法スタックにプッシュ */
        GPush(REDUCE(action), cp2);
```

```
                cp2->next = cp->next;
                cp->next = cp2;
            }
        }
    }

    for(cp = clist; cp != NULL; cp = cp->next) {
        state = StackTop(cp);

        for(ap = ActionTable[state]; ap != NULL; ap = ap->next) {
            action = ap->action;

            /* シフト時、およびアクセプトには、入力との照合を
               行なう。 */
            if(ap->input == input) {
                /*********
                 * Accept
                 *********/
                if(IsACCEPT(action)) {
                     accepted = TRUE;
                    cp2 = getcopy(cp);
                    Append(cp2, clist2);
                }
                /********
                 * Shift
                 ********/
                else if(IsSHIFT(action)) {
                    /* 新たなセルを作る */
                    cp2 = getcopy(cp);
                    Push(SHIFT(action), cp2);
                    /* 新たな状態をプッシュ */
                    Append(cp2, clist2);
                }
```

```
            }
        }
    }
End:
    /* 古いセルを回収する。
       これで還元もシフトもされなかったセル（これ以上解析が進展しない
       セル）は再利用されてしまう。
       部分木の解析等を行ないたい場合には、意図的にセルをコピーして
       おく必要がある。 */
    for(cp = clist; cp->next != NULL; cp = cp->next)
        ;
    cp->next = freecellp;
    freecellp = clist;

    *top = clist2;
    for(n = 0, cp = *top; cp != NULL; cp = cp->next)
        ++n;

    /* 何も残っていない。 */
    if(n == 0 && !accepted)
        return(_FAIL);
    /* ACCEPT された */
    else if(accepted)
        return(_ACCEPT);

    /* 解析途中 */
    return(_OK);
}
```