情報処理学会論文誌　プログラミング　Vol. 1　No. 3　1–10 (Oct. 2008)

Yayacc——自動エラーリカバリ機構付きパーサジェネレータ

山根雅司[†1]　山崎淳[†2]　阿部正佳[†3]

本発表では，自動エラーリカバリ機構を持つパーサジェネレータ yayacc の設計と実装を述べる．yacc や bison をはじめとして，現在広く使われているパーサジェネレータでは，文法中に特殊なエラーリカバリ動作を指示するトークンを手動で挿入させることで，エラーリカバリパーサを生成している．この古典的な手法は本来の文法定義を変えてしまうことに起因する深刻な問題があり，正しいエラーリカバリを行うパーサを生成させるには，多くの勘と経験が必要とされる．一方，yayacc ではエラーリカバリのために，文法を修正するというこがいっさい不要であり，生成されるパーサは従来の error トークン手動挿入によるパーサでは理論的に不可能な優れたエラーリカバリを自動的に行う．さらに yayacc では意味動作の Undo も自動で行うことが可能である．これは先行研究における自動エラーリカバリパーサでは扱われていないが，字句解析が完全に分離できない文法，たとえば C 言語に対しても自動的なエラーリカバリを行う場合に必要となる重要な機能である．yayacc は筆者らが開発している SCK コンパイラキットのツールの 1 つとして作成されたものである．現在 ANSI C 言語，および小規模な関数型言語のパーサは yayacc により自動生成されたものを使っており，きわめて優れたエラーリカバリが行われている．

Yayacc——A Parser Generator with Automatic Error Recovery

Masashi Yamane,[†1] Jun Yamazaki[†2] and Seika Abe[†3]

In this presentation, we describe the design and implementation of a parser generator named yayacc, which generates parsers with automatic error recovery. Widely used parser generators, such as yacc and bison, require parser writers to insert by hand special tokens indicating error recovery actions for generating error recovery parsers. This antique method involves serious problems due to the fact that the method actually modifies the original definition of given grammar. As a consequence, much of experience and inspirations are needed to generate parsers which can truly recover from errors. On the other hand, yayacc ruquires no such ad hoc modification of grammars for error recovery, yet the generated parsers do excellent error recoveries which could not theoretically be achieved by the antique method. Undoing semantic actions are also possible in yayacc. Previous studies and their experimental parser generators did not implement this feature, but it is indispensable to generate error recovery parsers for certain languages where a parser must semantically interact with its lexical analyzer to parse input programs, such as C language. We have developed yayacc as a tool of SCK compiler kit. Currently, parsers of ANSI C and a small functional language in SCK are yayacc generated ones and demonstrate excellent error recoveries.

1. はじめに

現在広く使われているパーサジェネレータでは，文法中に特殊なエラーリカバリ動作を指示するトークンを手動で挿入させることで，エラーリカバリパーサを生成している．この古典的な手法は本来の文法定義を変えてしまうことに起因する深刻な問題があり，正しいエラーリカバリを行うパーサを生成させるには，多くの勘と経験が必要とされる．

パーサジェネレータのような自動化ツールにおいては，このような本質的な問題解決が重要であると考える．yayacc のエラーリカバリ手法は，既存手法をふまえたうえで，それらでは不可能であった，あるいは取り上げられていなかったエラーリカバリの問題を，コンフィギュレーションのスナップショットリストにより解決した．

このしくみは新たなエラーリカバリ手法の研究，および既存手法の追試にも効果を発揮すると期待できる．

2. エラーリカバリ手法

ここでは主なエラーリカバリ手法を概説し，yayacc のリカバリ手法の特徴である属性値付きのコンフィギュレーションの必要性について説明する．

自動的ではないが，エラーリカバリがシステマティックに導入された初期のコンパイラとして Pascal の P4 コンパイラがある．そこでは手書きの再帰的下向き構文解析系では，各

†1 株式会社ワイズケイ
Y'sK Corporation
†2 株式会社ケイブ
CAVE Co., Ltd.
†3 株式会社数理システム
Mathematical Systems Inc.

解析関数の入口と出口でそこに許されるトークン集合までスキップし同期をとるという方法により，実用レベルのエラーリカバリが実現されていた．LL(1) ベースの手書きのパーサの場合，そもそも手書きであることから細かいエラーリカバリを導入しやすいというメリットがあるように思われる．

一方，自動的なパーサジェネレーションに向いている LR 系の構文解析では，現在でも，エラー処理用の特別な `error` トークンを挿入するという手法[1] が一般的である．経験上この方式は非常に難しく，勘と経験が必要とされる．また入力トークンを捨てることでのみ同期をとるという原理的な限界がある．

LR 系構文解析では，現在の構文解析の状態が LR オートマトンの状態のリスト（スタック）とこれから処理すべきトークンのリストのペア，すなわちコンフィギュレーションにより定まるので，ここでのエラーリカバリ手法はこのコンフィギュレーションの変換として説明することができる．

2.1　ローカルリカバリ

エラーを起こしたトークンの付近に対して削除，挿入，置換といった細かい修正を施し，いくつかの候補から最も妥当なものを選択するという手法[2] である．妥当性の基準としては，修正後どこまでエラーを起こさずに解析できるか（そのトークン数）が主な要素である．この手法はエラーリカバリのサーベイ[4] における比較でも，非常に優れていることが実証されている．

2.2　グローバルリカバリ

エラーが起こったコンフィギュレーションから，左（状態のスタック部分）と右（未処理トークン列）の適当な部分を削除するというものである．アイディアとしてはローカルリカバリよりも以前からり，広く研究されている[10]．Burke ら[2] ではローカルリカバリが失敗した場合の非常手段としてこのグローバルリカバリを利用している．なお，グローバルリカバリという用語は著者により使い方が異なることに注意されたい．

2.3　エラーリカバリの問題点

エラーが検出された箇所よりも以前に，本当のエラーの原因があるというケースが実際によく起こる．このようなケースを考慮している代表的な先行研究として文献 2)，3) があるが，パーサジェネレータが生成するパーサは純粋に構文解析をするだけではなく，一般には意味動作，属性値の計算が必要であることが，こういう処理を困難にしてきた．これら先行研究では Deferred parsing あるいは Double parsing と呼ばれる手法が使われている．それは属性値の計算を行わず，純粋に構文解析を行う部分を独立に設け，属性値計算を保留にしたままでエラーリカバリを行いつつ，ある限界を超えたところで属性値を計算していくというものである．

一方でそのような困難に対処することはあきらめ，すなわち過去は振り返らずに，今見えている先読みトークン以降をなるべく精密に修正するという試みがあった．そのようなアプローチとして以前には文献 7)，最近でも文献 6) がある．しかし，構文解析処理にかかる時間がコンパイラの中で占める割合はごくわずかであるし，コンピュータの性能が向上している現在，このような前提はあまり望ましくないと考えられる．実際，以前に遡らなければ修正できないエラーは，先読みトークン以後をいかに修正しようとも適切なリカバリにはならない．

2.4　Deferred parsing の必要性

LR 系の構文解析でエラーが検出されるのは，先読みトークンがステートスタック先頭の状態と矛盾している場合であり，先読みトークンが誤りであるとされる．しかし実際に誤っているトークンがその先読みトークンでない場合もある．たとえば以下の C プログラムにおいて，

```
int f(int x);
{
  return x;
}
```

明らかに最初の ‘`;`’ が余計なのであるが，一方でこの ‘`;`’ まででもプロトタイプ宣言として正しく解析できるので，実際にエラーが検出されるのはその次の ‘`{`’ になってしまう．ブロックがトップレベルに現れる構文はないので，このままエラーリカバリを継続しても良い結果は得られない．明らかにエラーが検出されたトークンより前のトークンまで遡ったエラーリカバリが必要である．しかしながら通常の LR パーサでは最初の ‘`;`’ を読んだ時点で還元動作が起こるので，もはや手遅れである．通常還元動作は属性値の計算を含み，その計算にはテーブル操作等の副作用が含まれうる．

Burke ら[2] では還元動作を一時的に保留しつつ解析を進める Deferred parsing という手法により，この問題を解決している．

2.5　typedef 問題

しかしながら，少なくとも C 言語では Deferred parsing の導入でも解決できない以下のような問題がある．この場合，最初の ‘`}`’ を削除するリカバリが好ましいが，そのためにはこのエラートークン削除後，次に現れる ‘`a`’ を型名と解釈し直す必要がある．

```
int f(int x)
```

```
{
  typedef int a;
  }
  a b = 1;
}
```

すなわち，C 言語を構文解析するためには，型名と変数名がトークンとして区別されていなければならない．実際には構文解析部の属性計算時に型名を記録しているテーブルをレキシカルアナライザが参照することで，この処理は実現されている．すなわち，C 言語では Burke らの Deferred parsing により，エラートークンより前のトークンを修正するような処理は不可能なのである．

2.6　属性値の Undo の必要性

この typedef 問題の本質は，パーサのアクションにおける属性値の計算が純粋な関数ではなく，テーブルのような大域的な変数への修正も含んでいることに起因する．yacc の例でいえば，計算済みの属性値 `$1,$2,...` から属性値を計算し `$$` へ代入するが，そのアクション中で行われるサイドエフェクトが属性文法の枠組みから外れているため，エラーリカバリ処理で状態を前に戻した場合に同期がとれなくなってしまう．

我々はこのサイドエフェクトも，アクション実行の属性値の一部と考え，それらのスナップショットを保持するという，自然な方法でこの問題を解決した．この方法ではエラーリカバリによりバックトラックが発生しても，サイドエフェクトにより修正された環境を以前の状態に正しく戻すことが可能であり，上記の typedef 問題を解決することができる．たとえば簡単な変数への代入のある電卓プログラムを yayacc で記述する場合を考えよう．

```
> a = 1;
result : 1

> a = 2;* a;
syntax error, delete ";"

> a = 2 * a;
result : 2
```

この例では 2 番目の `a = 2;*a;` という入力でエラーとなり，最初の ; を削除することでエラーリカバリしているが，エラーが発見されるのは `*` を読んだときであり，すでに `a` に 2 が代入されるアクションが実行されているが，yayacc ではエフェクトも Undo されるので，最後は期待される結果となっている．

もちろんサイドエフェクトの記述の仕方は任意であり，ユーザが記述したアクションからそれを自動的に検出することは困難である．yayacc ではサイドエフェクトの部分の記述に制約を課すことで，この機構を実現している．

3.　yayacc のエラーリカバリ手法

本章では yayacc のエラーリカバリ手法をある程度抽象的に定式化して説明する．より詳細な説明は「実装」の章で行う．

yayacc では細かな修正を試みるローカルリカバリと，それらがすべて失敗した場合でも解析を継続するためのグローバルリカバリの両方を実装している．これらのリカバリ手法の基盤となるのが，コンフィギュレーションのスナップショットリストである．

LR 構文解析におけるコンフィギュレーションとは，すでに解析済みのトークン列により遷移，あるいは還元を行った結果を表す LR オートマトンの状態のリスト（スタック）と，未処理のトークン列の組であり，次に処理されるトークンを特に先読みトークンという．このコンフィギュレーションが，構文解析のある時点での状態を完全に表している．

`State`	=	LR オートマトンの状態 と属性値の組み
`Token`	=	入力トークン
`Config`	=	(`State`...,`Token`...)
`recovery`	:	`Config` → `Config`

先に述べたとおり，yayacc では意味動作により生成された属性値の Undo も行う必要があるので，状態のスタックにはその属性値も記録されている．属性値の扱いについては，後の実装の章で説明する．このコンフィギュレーションを保持しておけば，構文解析の状態を意味動作も含めて，その保持された時点での状態に復元することが可能となる．

この環境の下では，エラーリカバリとはエラーコンフィギュレーションを引数とし，何らかの方法でエラーを修正したコンフィギュレーションを返す関数 `recovery` として簡潔に定式化できる．

3.1　スナップショットリスト

コンフィギュレーションのスナップショットはスナップショットリストに保持されていく．このリストの長さは「実装」の章で説明するパラメータで指定され，それを超えた場合は古いものから廃棄されるので，正確にいえば長さが一定のキュー構造である．

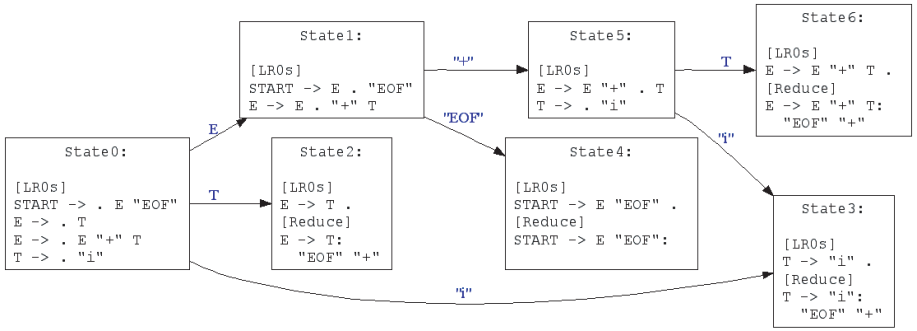


図 1　LR オートマトン
Fig. 1　An LR automaton.

スナップショットは構文解析のシフト動作が行われた直後にのみ記録される．たとえば以下の文法の例では，**図 1** の LR オートマトンが生成される．

```
E  →  E * T
E  →  T
T  →  i
```

このパーサで `i * i * * EOF` という（エラー）入力を解析した場合のコンフィギュレーションの遷移は以下のようになる．

```
c1 :  0 | i * i * * EOF
c2 :  0 -i->3 | * i * * EOF
      0 -T-> 2 | * i * * EOF
      0 -E-> 1 | * i * * EOF
c3 :  0 -E-> 1 -*-> 5 | i * * EOF
c4 :  0 -E-> 1 -*-> 5 -i-> 3 | * * EOF
      0 -E-> 1 -*-> 5 -T-> 6 | * * EOF
      0 -E-> 1 | * * EOF
c5 :  0 -E-> 1 -*-> 5 | * EOF
      ⋮
```

このなかで番号のついたコンフィギュレーション（$c1$ から $c5$）のみがスナップショットリストに記録される．コンフィギュレーション c の n（< 0）個前のスナップショットコンフィギュレーションを $c[n]$ で表す．上の例でいえば $c_5[-1] = c_4$ である．

3.2　ローカルエラーリカバリ

エラーリカバリ関数 `recovery` は，与えられたエラーを含むコンフィギュレーションに対して，まずこのローカルリカバリ `localfix` を試みる．

```
localfix   :  (FixPos, LFixOp) →
              Config → Config
  FixPos   =  Z
parsedist  :  Config → ParseDist
  LFixOp   =    LFDel
              | LFIns Token
              | LFSubst Token
              | LFSpell
ParseDist  =  N
```

これはローカルリカバリの種類（適用すべき場所 `FixPos` と操作 `LFixOp`）により定まるエラーコンフィギュレーションの修正関数である．`FixPos` は先読みトークンが 0，その 1 つ前を -1，と数える．`LFixOp` はローカルな修正の種別であり，それぞれトークンの削除，挿入，置換，およびミススペリング修正を意味する．たとえばエラーコンフィギュレーション c_5 については以下のような修正が可能である．

```
localfix (0, LFSubst"i") c5
   ⇒  (0 -E-> 1 -*-> 5 | i EOF)
```

一般には多くの可能な修正候補があるが，ローカルリカバリではそれらのすべてを試した後，その中から最も妥当なものを選択する．その基準として重要なものが解析距離 `ParseDist` である．これは修正したコンフィギュレーションから解析を継続し，何トークン先までエラーなしで解析できたかの，そのトークン数であり，これを求めるのが `parsedist` 関数である．

この関数は事実上構文解析プログラムそのものであるが，エラーが起こるごとに可能な限り解析を続けるのは効率のうえで問題なので，「実装」の章で説明されるようなパラメータ

によりユーザが解析距離の上限を定めることが可能となっている．また，たとえローカルな修正をしたとしても，その解析距離があまりに短い場合，妥当な修正とは見なせない．この下限も同様にユーザがカスタマイズ可能である．これら，パラメータで制御される要素は以下の関数として定式化可能である．

$$
\begin{aligned}
\texttt{LFixResult} &= (\texttt{FixPos}, \texttt{LFixOp}, \texttt{ParseDist}) \\
\texttt{lfsuccess} &: \texttt{LFixResult} \to \boldsymbol{Bool} \\
\leq_{\texttt{LF}} &\subset \texttt{LFixResult} \times \texttt{LFixResult}
\end{aligned}
$$

ここに，LFixResult はローカルな修正をした結果の評価基準となるデータである．そのようなローカルなリカバリを適用された各コンフィギュレーションは lfsuccess でテストされ，パスしたものの集合上で，これもユーザがカスタマイズ可能な全順序関係 $\leq_{\texttt{LF}}$ において，最も大きな値を持つようなコンフィギュレーションが選択される．

この lfsuccess の存在から明らかなように，ローカルリカバリは失敗する可能性がある．すなわち，すべての修正候補が lfsuccess により拒否される場合である．

3.3　グローバルエラーリカバリ

すべてのローカルリカバリが失敗した場合は，以下に述べるグローバルリカバリを試みる．これはエラーが発生した周辺を思い切って削除するという手法である．よって，それを行う関数 globalfix は切り取るべき範囲と操作を引数としてとり，エラーコンフィギュレーションの修正を試みる．

$$
\begin{aligned}
\texttt{globalfix} &: (\texttt{FixPos}, \texttt{FixPos}, \texttt{GFixOp}) \to \\
&\quad \texttt{Config} \to \texttt{Config} \\
\texttt{GFixOp} &= \texttt{GFDel} \mid \texttt{GFSubst}
\end{aligned}
$$

置換操作 GFSubst は置換，すなわち切り取った後に状態スタックを参照しつつ可能な記号を挿入するという手法[3] である．yayacc では試験的に実装してみたが，有効かどうか疑問であり，現在は正式にはサポートしていない．ここでは枠組みを定義するのが目的なので書き添えてある．

グローバルリカバリではとにかく正しいコンフィギュレーションを返す必要があるので，ローカルリカバリの lfsuccess のような基準は設けず，試みたグローバルリカバリすべてを候補として，全順序関係 $\leq_{\texttt{GF}}$ で最大のものを選択する．

$$
\begin{aligned}
\texttt{GFixResult} &= (\texttt{FixPos}, \texttt{FixPos}, \texttt{GFixOp}, \\
&\quad \texttt{ParseDist}) \\
\leq_{\texttt{GF}} &\subset \texttt{GFixResult} \times \texttt{GFixResult}
\end{aligned}
$$

4. 実　　装

yayacc は SCK プロジェクト[9] の一部として開発した LALR(1) ベースのパーサジェネレータである．SCK は Emacs Lisp（GNU Emacs エディタのマクロ定義用 Lisp 処理系）で記述された Emacs 上のコンパイラ開発環境であり，yayacc も Emacs Lisp で実装されている．yayacc と同様に以下のような文法とアクション（属性値を計算する関数）の組の並びから，Emacs Lisp で書かれたパーサを生成する．

```
.....
((iteration_stmt
  "for" "(" exp ";" exp ";" exp ")" stmt)
  (sck-cparse-make-for $2 $4 $6 $8))
 .....
```

アクションは Emacs Lisp の任意のフォームが許される．この例では sck-cparse-make-for という Emacs Lisp 関数を呼び出している．すでに計算されている i 番目の属性値は yacc と同様 $i で参照する．そして，フォームの評価結果が還元時の属性値となる（yacc では $$ への代入で表現する）．

LR オートマトンテーブルの作成は bison で使われている高速な手法[5] を利用している．ここでは yayacc のエラーリカバリの詳細とリカバリ制御パラメータについて説明する．以下，パーサメインの擬似コード**図 2** およびエラーリカバリ処理の擬似コード**図 3** を参考にされたい．

4.1　エラーリカバリ処理

パーサはシフト動作が起こった際先読みしていたトークンを実際に読み込み，新しく次のトークンを見にゆく．次のトークンを初めて見たこのタイミングで現在の状態を記録したスナップショットを作成する．エラーリカバリが発生したとき，パーサはこのスナップショットの状態にもどしてこれから読むトークンを修正する動作を行う．もしこのタイミング以外の状態にもどした場合，修正される先読みトークンの影響でパーサは還元動作を行ってしまっているのでスタックや属性値評価によるサイドエフェクトは正しい状態にもどされない

```
// Config のスナップショットリスト
// 末尾が最新の Config
configQue : List of Config;

main()
{
  while (1) {
    // 今のコンフィギュレーションのスナップショットを撮り，キューに記録
    conf = makeSnapShot();
    configQue = configQue ++ conf;

    // 1 トークン読み，LR オートマトンを進める
    // GOTO，REDUCE アクションは parseOneStep() 内部で処理
    action = parseOneStep();

    if (action == ERROR) {
      // エラー発生
      fixresult = errorRecovery();
      // fixresult に従って configQue を修正する
      fixUpConfigQue(fixresult);
    }
    else if (action == ACCEPT) {
      // パース終了
      break;
    }

    if (length(configQue) > undo)
      // キューの長さをエラーリカバリ制御パラメータ undo
      // 以下にメンテナンス
      configQue = cdr(configQue);
    }
  }
}
```

図 2　擬似コード（main）
Fig. 2　Pseudo code (main).

ので正しく動作しない．通常時パーサはシフト動作がおこるごとにパーサの状態を記録してゆく．

次にエラーが発生したときの動作について説明する．我々のパーサはローカルリカバリとグローバルリカバリをサポートしており，後者は前者がうまくいかなかった場合のみ試みられる．

```
FixResult errorRecovery()
{
  fixList = [];
  foreach_reverse (conf, configQue) {
    if (conf に対してのローカルリカバリが可能)
      fixList = fixList ++ localRecovery(conf);
  }

  if (fixList == []) {
    // ローカルリカバリが不可能だったので，グローバルリカバリを試みる
    // global-fix はエラーリカバリ制御パラメータ
    for (l = 0; l <= global-fix.left; l++) {
      conf = configQue[l]; // FIFO キューの最後から l 番目
      for (r = 0; r <= global-fix.right; r++) {
        // エラーポイントの左に l 右に r の範囲でグローバルリカバリを試みる
        if (グローバルリカバリが可能)
          fixList = fixList ++ globalRecovery(conf, l, r);
      }
    }
  }

  if (fixList == []) {
    // エラーリカバリを諦め終了
    exit;
  }

  // リカバリ制御パラメータから定まる優先順位に従って
  // 最も良い FixResult (= LFixResult ∪ GFixResult) を返す
  return selectBestFix(fixList);
}
```

図 3　擬似コード（errorRecovery）
Fig. 3　Pseudo code (errorRecovery).

ローカルリカバリでは順番にスナップショットリストを遡ってゆき，それぞれの状態において 1 トークンだけ修正を行い，パーサがエラーの発生した位置からの解析距離を求める．このとき，あらかじめ決めておいた閾値を超えた修正だけ候補として残してゆく．こうして集めた候補を「修正動作」「解析距離」「スナップショットの位置」を基に最も適切な修正候補を決定する．この判定関数はユーザが定義することもできるがデフォルトではまず解析距離の一番長いものを選び，複数ある場合あらかじめ決めておいた修正動作の優先順位の高い方を選び，それでも複数あればスナップショットを遡る距離が一番短かったものを選んで

いる．なお，我々のパーサでは修正動作の種類として「削除」「挿入」「置換」，さらに「置換」の特殊なケースとしてミススペリング修正機能を用意している．話は前後するが「修正動作」とは上で述べた種類とトークンのペアのデータのことで，あらかじめ優先順位を設定できるようになっている．

グローバルリカバリではエラーが起こった位置から前後に複数のトークンを削除する処理を行っている．エラーの位置より先は単純にその数だけ削除するが，エラーが起こった位置より前のトークンの削除に関しては削除するトークン数だけスナップショットを遡りいったん状態を戻してから遡った数だけトークンを削除している．

4.2　属性値の Undo 処理と effect

属性の計算には純粋な関数として記述できるものばかりではなく，一般には，大域的なテーブル（C 言語の型名テーブルといった）参照等が含まれる．したがってそういった副作用による環境の変更も Undo するための機構が必要となる．たとえば先にあげた電卓の例では，yayacc での記述は以下のようになる．

```
((exp  var "=" exp)
 ;; <exp> ::= <var> = <exp>
 (assign $0 $2))
```

アクション関数 assign は変数 $0 に式の値 $1 をサイドエフェクトとして変数値テーブルを更新し，純粋な関数値として $1 を返す．この純粋な関数値自体は，リカバリによるバックトラックからの起動でもまったく問題はないが，変数値テーブルについてはケアが必要となる．しかし，先に述べたように関数の中を解析し，サイドエフェクトを自動的に抽出するのは困難である．

そこで，yayacc では大域的なデータをサイドエフェクトとして修正する場合，ユーザは yayacc のスペックに effect というグローバル変数以下に，それらがたどれるような形ですべてのグローバルデータを保持する，という規約を設けた．すなわち，assign の定義は以下のようになる．

```
(defun assign (var val)
  (let ((a (assoc var effect)))
    (if a
        (setf (cadr a) val)
      (push (list var val) effect)))
  val)
```

この場合，大域的なデータは変数値テーブルのみなので，それを直接 effect 変数に保持しているが，一般には関連するすべての大域的データを何らかのデータ構造で表現したものを effect に設定することになる．

```
(error-recovery
  ((undo 5)
   (parse-check (2 10))
   (misspelling-rate 0.3)
   (local-fix ((misspelling ("sizeof" "return" "break" "break" ...))
               (deletion (not "id" "int_const" "char_const" ...))
               (insertion (not "id" "string" "int_const" ...))
               (substitution (not "id" "string" "int_const" ...))
               (deletion ("id" "string" "int_const" ...))
               (insertion ("id" "string" "int_const" ...)
               (substitution ("id" "string" "int_const" ...))
   (global-fix (4 4)))))))
```

図 4　エラーリカバリ制御パラメータ
Fig. 4　Error recovery controlling parameters.

これにより，ユーザは大域的データの参照をつねに effect から行うことになる．一方 yayacc 側では Undo が起こったときに，保持されていた以前の effect 値をリストアすることにより，属性値の Undo を実現している．

各スナップショットでは effect 変数のディープコピーがとられ，状態スタックに保持される．そして，リカバリでバックトラックが発生した場合は，そのコンフィギュレーションに保持されているエフェクトのディープコピーが effect に設定された後，ユーザの指定したアクションが実行される．このような処理を一般的に記述するのは難しいが，Lisp では容易である．なお，この処理の詳細は煩雑になるので擬似コードでは省略した．

4.3　エラーリカバリの制御パラーメータ

ユーザが設定可能なエラーリカバリのパラメータについて C 言語の場合を例として説明する．**図 4** は yayacc の文法記述ファイル中で，エラーリカバリの制御に関する指定部分の引用である．設定パラメータは (<パラメータ名>　<データ>) という A-List 型式となっている．

undo パラメータにはスナップショットリストの長さの上限を設定する．この数よりスナップショットが増えると古いものから捨てられる．またエラーリカバリの際，このスナップショットの数だけリカバリの検査を行う．

parse-check パラメータには修正を行ったときにパーサが何ステップ進めば候補として採用するかという最低ステップ数と検査を何ステップまで行うかという最大ステップ数を設

定する．なおパーサが入力を受理した場合は最低ステップ数より少なくても，最大ステップ数進んだものと同じ扱いにしている．

`misspelling-rate` パラメータにはミススペリングの検査用の閾値を設定する．edit distance（levenshtein distance[8]）を文字列の長さで割った値がこの閾値以下だった場合にミススペリングだと判断している．

`local-fix` パラメータにはローカルリカバリでの検査方法を設定する．データは (<検査の種類> <検査するトークン>) という構造になっている．<検査するトークン> の最初の要素に `not` を入れるとその後に指定したトークン以外という意味を表す．検査の種類には misspelling deletion insertion substitution があるが，misspelling だけは <検査するトークン> の指定にトークン id とトークン文字列のリストを指定する．トークン id とトークン文字列が同じ場合，リストにしないで指定可能である．また，ここで指定した順番が検査のプライオリティを表している．検査するトークンもリストの初めの方がプライオリティが高くなっている．C 言語の設定例ではキーワードの編集をする動作の優先順位が高く設定してある．

`global-fix` のパラメータにはエラートークンの位置から前後どれだけの数のトークンをグローバルリカバリによって編集を行うかを指定する．ここで指定された数だけ総当りで検査する．たとえば (2 2) と設定した場合，前後に (0 1) (0 2) (1 1) (1 2) (2 1) (2 2) の組合せを検査する．検査の種類としては deletion だけをサポートしている．

このようにエラーリカバリ動作のカスタマイズをパーサジェネレータの入力パラメータによって事前に設定することが簡単にできるようになっている．

5. 評　　価

エラーのない，100 行程度のプログラム `sample.c` および付録に掲載したようなエラーを含んだプログラム `error.c` について実行時間を計測した（**表 1**）．undo は制御パラメータ undo で指定される `configQue` の長さであり，長いほど広範囲でリカバリを試みる．特に undo = 0 のプログラムではエラーリカバリに関連するコードをバイパスするようにしている．また GC はガベージコレクションの回数，Time は解析に要した時間である．

この結果から，エラーが起こらない限り，`configQue` の長さは実行時間に影響を与えないことが分かる．もちろんキューの長さが長ければ常時使用しているメモリ量は増えるが，それはキューの長さと `effect` のメモリ使用量の積であり，通常の構文解析では問題にならないと考えられる．

表 1　実行時間
Table 1　Elapsed time.

Source	undo	GC	Time (sec)
sample.c	0	19	0.55
sample.c	5	19	0.58
sample.c	10	19	0.52
sample.c	20	19	0.54
sample.c	50	19	0.55
error.c	0	0	0.02
error.c	5	18	0.66
error.c	10	33	1.38
error.c	20	58	2.61
error.c	50	104	5.03

一方で，エラーが発生した場合は著しい速度低下が起こっている．undo = 50 は非現実的な大きさであり，通常は付録のテスト結果で使用した undo = 5 でも十分なリカバリが可能である．

6. おわりに

我々は当初文献 2)，3) と同様のアプローチを検討していたが，先に説明した C 言語の typedef 問題によりその可能性は困難であることが分かった．結局，属性値の Undo は必要なのである．このためにユーザはすべてのアクションを，純粋な関数として記述するか，そうでなければすべての大域的なデータ（C 言語の型名テーブルといった）を 1 つのグローバル変数 `effect` 以下（それからたどれる形で）に作成しなければならない，という制約が課せられたが，少なくとも Lisp ではこの制約は大きなものではない．他の言語でも effect 以下のディープコピーを作成するような関数をユーザが提供すればよい．

むしろ，このように属性値を含めたコンフィギュレーションと，そのスナップショットリストという枠組み自体に意味があると考える．このフレームワークにより，さまざまなエラーリカバリ手法の研究が Deferred parser のような複雑な処理を基盤にするよりも容易に行うことが可能になったと考えている．

また，このようなフレームワークの提供は母体プロジェクト SCK[9] の考えにも沿っている．SCK プロジェクトにおけるソース言語，C 言語および MinCaml のパーサは yayacc により生成されたものが使われている．今後は現在のエラーリカバリ機構の，より広範囲の言語における評価が必要である．

最後に，SCK プロジェクトを未踏ソフトウエア創造事業 2007 II 期に採択していただき，また貴重な意見や励ましをくださった，九州大学大学院システム情報科学研究院の竹田正幸氏に感謝いたします．

参　考　文　献

1) Aho, A.V. and Johnson, S.C.: LR Parsing, *ACM Computing Surveys*, Vol.6, Issue 2 (1974).
2) Burke, M.G. and Fisher G.A.: A Practical Method for LR and LL Syntactic Error Diagnosis and Recovery, *ACM TOPLAS*, Vol.9, No.2, pp.164–197 (1987).
3) Charles, P.: A Practical method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery, Ph.D. thesis, New York University (May 1991).
4) Degano, P. and Priami, C.: Comparison of Syntactic Error Handling in LR Parsers, *Software Practice and Experience*, Vol.25, No.6 (1995).
5) DeRemer, F. and Pennello, T.: Efficient Computation of LALR(1) Look-Ahead Sets, *TOPLAS*, Vol.4, Issue 4 (1982).
6) Mckenzie, B.J., Yeatman, C. and De Vere, L.: Error Repair in Shift-Reduce Parsers, *TOPLAS*, Vol.17, No.4 (1995).
7) Pennello, T.J. and DeRemer, F.A.: A Forward Move Algorithm for LR Error Recovery, *Conf. Record ACM Symposium on Principles of Programming Languages* (Jan. 1978).
8) Wagner, R.A. and Lowrance, R.: An Extension of the String-to-String Correction Problem, *JACM*, Vol.22, No.2, pp.177–183 (1975).
9) SCK Home Page. http://www14.plala.or.jp/gazico/sck/
10) Sippu, S. and Soisalon-Soininen, E.: A Syntax-Error-Handling Technique and its Experimental Analysis, *TOPLAS*, Vol.5, No.4, pp.656–679 (1983).
11) Eijiro Sumii: MinCaml. http://min-caml.sourceforge.net/index3.html

付録　エラーリカバリの例

```
int a b;
————————
insert ","
int a , b ;
```

この例では b というトークンに出会ったときにパーサはエラーとなる．ここでエラーリカバリが動き，b の位置に ‘,’ を挿入するという動作を行うことでパーサが解析可能な状態に復旧している．修正候補として実際には b の削除等も出てくるがキーワード以外の削除よりもキーワードの挿入の優先順位が高く設定してあるため，int a; と修正されずに int a, b; に修正されている．

```
int  f(int x);{return x;}
————————————————————
delete ";"
int f ( int x ) { return x ; }
```

この例では ‘{’ に出会ったときにパーサはエラーとなる．ここでエラーリカバリは ‘;’ というトークンに出会った状態まで戻り ‘;’ を削除する動作を行うことで復旧している．このように以前の状態に戻って修正を行うことで初めてこの例のようなエラーリカバリを行うことができる．

```
chara a;
————————————————————
chara a ;
edit misspelling chara to char
char a ;
```

この例では a というトークンに出会ったときにパーサはエラーとなる．ここでは 1 つ前の状態に戻って chara というトークンを char に置き換えている．これは misspelling の検査によって置換する動作が選択されている．chara と char の edit distance は 1 となり edit distance を文字列の長さで割った値が 0.2 となり閾値として設定してある 0.3 より小さいため修正候補とになっている．また misspelling による修正動作の優先順位が最も高く設定してあるのでこの動作が選択されている．

```
int ))a;
————————————
remove phrase ") ) "
int a ;
```

これはローカルリカバリがすべて失敗し，グローバルリカバリを適用する例である．最初に ‘)’ に出会ったときにパーサはエラーとなる．ここでエラーリカバリはエラートークン ‘)’ とその次のトークン ‘)’ をまとめて削除する動作を行うことで復旧している．

```
int f(int x)
{
  typedef int a;
```

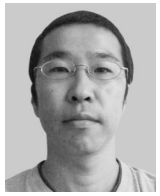
```
  }
  a b = 1;
}
```

————————————

```
a b = 1 ;
delete "}"
typedef int a ; a b = 1 ;
```

この例では最初に b というトークンに出会ったときにパーサはエラーとなる．ここでエラーリカバリは初めにトークン‘}’を見た部分まで戻り‘}’を削除している．注意して見てもらいたいのはパーサはエラーが起こる前最初に‘}’を読んだときに a を int 型に型定義している typedef の効果を取り消すサイドエフェクトが発生している．したがって次の a b という並びで a を型とは認識できずにエラーとなっている．ここでパーサのエラーリカバリ処理は‘}’を初めて見た状態まで戻って修正するのであるが typedef によるサイドエフェクトによる影響ももとの状態に戻しているので‘}’を削除することで a の型定義が取り消されることなく処理が継続されている．

(平成 20 年 4 月 22 日受付)
(平成 20 年 7 月 28 日採録)



山根　雅司

1992 年岡山大学中退．2007 年度未踏ソフトウェア（SCK）プロジェクトにおいて，本論文で紹介したパーサジェネレータ yayacc を担当．ゲームプログラミング等における物理シミュレーションに興味を持つ．現在，（株）ワイズケイに勤務．

山崎　　淳

1989 年日本電子専門学校電子情報処理科卒業．ソフトウェア会社勤務を経て，同未踏プロジェクトに参加．OS や言語処理系の実装に興味を持つ．現在，（株）ケイブに勤務．

阿部　正佳

1984 年東京理科大学理工学部数学科卒業．ソフトウェア会社勤務を経て，2005 年東京大学大学院博士課程終了．理工学博士．同未踏プロジェクトに参加．現在，（株）数理システムに勤務．