再帰的下向き構文解析における演算子順位構文解析

中田育男† 山下義行†

加算や乗算などの演算子を含んだ通常の式の形をしたものの文法は，生成規則だけによる定義より，演算子順位を使った定義のほうが一般には分かりやすく，構文解析の効率もよい．LR 構文解析の中で演算子順位を利用した構文解析をする方法はよく知られているが，再帰的下向き構文解析の中での方法は，演算子の順位を示す数値を手続きの引数として渡す方法があるようであるが，あまり報告されていない．本論文では，後者の方法として，再帰的下向き構文解析法における再帰的手続きの呼び出しの引数として，文法の LL(1) 性などを調べるのに使われる Follow 集合の部分集合を使う方法を提案する．これは，引数として演算子の順位を示す数値を渡す方法より一般的である．また，この部分集合は，再帰的下向き構文解析法でエラー処理のためによく使われる集合に近いものであり，この方法を再帰的下向き構文解析に導入するのは容易である．

Operator Precedence Parsing in Recursive Descent Parsers

IKUO NAKATA† and YOSHIYUKI YAMASHITA†

A definition of the syntax for expressions with several kinds of operators is best expressed by a simple, but ambiguous, grammar with disambiguating rules composed of operator precedence relations, and an efficient parser can be derived from such a definition. While LR-parsing techniques for this kind of definitions are well known, corresponding LL-parsing techniques are rarely reported. This paper describes a simple technique for parsing expressions in a recursive descent parser. It uses subsets of *Follow* sets which are usually used to check whether a given grammar is LL(1) or not. A recursive procedure that parses an expression takes an appropriate subset as a parameter. Our method is more general than the one that passes an operator precedence value as a parameter.

1. はじめに

コンパイラにおける構文解析の手法の代表的なものは，LR 構文解析，LL 構文解析，演算子順位構文解析の三つである．それらは，それぞれに利点，欠点があるが，加算や乗算などの演算子を含んだ通常の式の形をしたものの構文解析については，演算子順位構文解析がもっとも効率がよい．なぜならば，演算子の種類（優先順位の異なるもの）が多い場合，再帰的下向き構文解析では再帰的呼び出しのネストが深くなり効率が落ちるし，LR 構文解析では，単純な生成規則（A→B で A も B も非終端記号の形）に対する還元が多くなり効率が悪い．たとえば，C言語のように，演算子の優先順位のレベルが17もある（C の ISO 規格では primary-expression から expression まで，式のレベルが 17 ある）場合は，通常の再帰的下向き構文解析法に従えば，17 個の手続きが作られ，最も簡単な式（例えば変数一つだけからなる式）を構文解析する場合でも，少なくとも 17 回の手続き呼び出しがおこる．

このような場合，LR 構文解析の手法をとっている yacc[1] では，生成規則ではすべての演算子を同等に表現し，別途優先順位を宣言することにより，構文解析の効率が上がるようにしている[2]．この文法表現により文法が見やすくなる効果もある．yacc の生成する構文解析プログラムでは，この優先順位の情報を使ってシフト／還元の競合を解決している．LR 構文解析も演算子順位構文解析も共にシフト／還元による構文解析であるから，このような両者の併合は自然にできる．

本論文では，同じような表現を，LL 構文解析の代表的なものである再帰的下向き構文解析の場合に取り入れる方法を提案する．すなわち，文法としては，左再帰性のあるものも，演算子の優先順位の宣言も許す方法を考え，その文法に対する再帰的下向き構文解析法を提案する．その構文解析法としては，手続き呼び出しの引数として演算子の優先順位を渡す方法も考え

† 筑波大学電子・情報工学系
Institute of Information Sciences and Electronics, The University of Tsukuba

られるが，ここではそれをもう少し一般化して，文法の LL(1) 性などを調べるのに使われる Follow 集合の部分集合を使う方法を提案する．これは，文献 3) の PL/0 コンパイラで，エラー処理のために手続き呼び出しの際にパラメータとして渡している follow 集合に近いものである．

以下，2章で簡単な例を用いて提案する方法を説明し，3章で一般的な方法を述べ，4章で実現法に関することを述べ，5章で従来の方法と比較する．

2. 簡単な例

次の文法は，加算と乗算からなる式を定義しており，コンパイラの教科書でよく扱われているものである．

G1: E→E '+' T|T
T→T '*' F|F
F→i|'(' E ')'

この文法で与えられる式の再帰的下向き構文解析のプログラムは，この文法を

G2: E→T{'+'T}
T→F{'*'F}
F→i|'(' E ')'

と書き換えたもの（{α} は α の0回以上の繰り返しを表す）に対するプログラムとして，通常，**図1**のように与えられる[4]．なお，一般に構文解析プログラムではエラー処理のために，**if** に対応して **else** error といった部分があるが，本論文では，プログラムを短く表現するためにそれは省略する．get (next) はソースプログラムから次のトークンを変数 next に読み込む手続き呼び出しである．

このような式で，演算子の種類が多くなり，その演算の優先順位が複雑になると，多くの非終端記号が導入されるので，文法が読みにくくなり，構文解析時にはそれらの非終端記号に対応する手続きの呼び出しのネストが深くなるので，解析の効率も悪くなる．その問題を解決するために，生成規則として

```
procedure E:
begin    call T;
         while next='+' do
              begin get(next); call T end
end;

procedure T:
begin    call F;
         while next='*' do
              begin get(next); call F end
end;

procedure F:
begin    if next='i' then
              begin get(next); return end
         else if next='(' then
              begin get(next); call E;
                   if next=')' then
                        begin get(next); return end
              end
end;
```

図1 文法 G2 の再帰的下向き構文解析プログラム
Fig. 1 Recursive descent parser for G.

G3: E'→'$' E '$'
E→E '+' E
E→E '*' E
E→i|'(' E ')'

または，これの左再帰性を除いた

G4: E'→'$' E '$'
E→(i|'(' E ')'){'+' E|'*' E}

のような表現を許したとして，それに対する構文解析法を考えることにする．ただし，この文法はあいまいであるから，そのあいまい性を除去するために，この生成規則に付随して別途，'+' と '*' は左結合性を持つ演算子であり，'+' より '*' のほうが演算の優先順位が高いことが宣言されているものとする．このような文法に対して，演算子の優先順位を利用した構文解析をするプログラムを以下のように考えて作ることにする．

再帰的下向き構文解析の中で演算子順位解析を実現するためには，通常の演算子順位解析で使われる演算子のスタックを，なんらかの形で模擬する必要がある．それには，手続きの呼び出し系列の情報に，その手続きにパラメタとして渡される演算子を含んだものをスタックに対応させればよいと考えられる．

演算子順位解析の場合は，スタックの先頭の演算子 p が次の演算子 n より優先順位が高いときにある非終端記号 D に還元されるのであるが，それを導出の列として表現すると，（たとえば，p が2項演算子なら）

$$S \Rightarrow \alpha Dn\beta \Rightarrow \alpha ApBn\beta$$

となる．この場合，次の演算子 n は非終端記号 D の Follow 集合に含まれると考えられる．Follow 集合とは，文法の出発記号を S，終端記号の集合を V_T としたとき，

$$\mathrm{Follow}(A) = \{x \mid S \Rightarrow \alpha Ax\beta,\ x \in V_T\}$$

で定義されるものである．

上記の文法での式を考える場合，'+' と '*' が演算子であり，'$' や ')' は演算子とは考えないのが通常である．もちろん，すべてを演算子順位解析で解析する場合は，'$' や ')' も演算子と見なして優先順位を設定するのであるが，'$' や ')' は本来 Follow 集合の記号と考えたほうが自然である．そこで，ここでは逆に，

'+' や '*' の演算子も Follow 集合の記号と考え，両者を統一的に扱うことにする．

ところで，上記の文法でのEの Follow 集合は，どのEについても {'$', '+', '*', ')'} であり，これをそのまま使っては演算子の優先順位の違いなどを表現することはできない．しかし，たとえば，上記の

'$' E '$'

のEの次に来るのは '$' しかありえない．これは，E の Follow 集合の部分集合である．そこで，このように，文脈に依存した部分集合を使うことにする．文脈は構文解析の進行にともなって決まっていくものであるから，手続きの呼び出しによって解析をすすめる再帰的下向き構文解析では，手続きの呼び出しごとに決まる集合（Follow 集合の部分集合）を渡すようにすればよい．たとえば，

'$' E '$'

のEの構文解析をするための手続きを呼び出す（以後，誤解の恐れのない場合は「Eの構文解析をするための手続き」を単に「E」と書くことにする）ときは '$' だけを渡し，

'(' E ')'

のEを呼び出すときは ')' だけを渡せばよい．この部分集合の求め方は文献 3) でエラー処理のために渡している follow 集合や，LR(1) 構文解析のための LR(1) 項の先読み要素の求め方とほぼ同じである[5]．

以上の二つの場合は，Eの次に来るものが決まってしまうので簡単であるが，

'+' E (1)

のEを呼び出すときには演算子の優先順位を考慮した集合を渡す必要がある．'+' が左結合性を持つことを考えれば，'+' はそのEに渡すべきものに入る．なぜならば，そのEの解析中に別の '+' を見たら，その '+' よりも (1) の '+' を先に還元しなければならないからである．しかし，'*' は (1) のEに渡すべきでない．なぜならば，そのEの解析中に '*' を見たら，その '*' は (1) の '+' より優先順位が高いから，そのEの中で還元すべきであるからである．したがって，このEを呼び出すとき新たに渡すべき演算子の集合は {'+'} だけである．

以上の考察から，上記の文法 G4 に対する構文解析プログラムは**図 2** のようにすればよいことがわかる．図2の行3の **call** E ({'$'}) はEの follow 集合として {'$'}，すなわち，この場合のEの直後に来るのは '$' だけであることを指定している．行13ではそれが ')' だけになる．行17は，次のトークン next が，今Eが呼ばれたときの follow 集合に入っていなかったら以下を繰り返し，入っていたら，このEを終了して戻ることを示す．行19の **call** E(follow∪{'+'}) は，これから解析するEの直後に来るのは，そのEの直前で読んだ '+' より演算の優先順位の低い '+'（左にある '+' よりも，右にある '+' のほうが低い）か，今解析中のEの直後に来るものとして指定されているもの（follow として渡されているもの）かのどちらかであることを指定している．follow に含まれるトークンを読んで戻った場合は，続いて現在のEからも戻ることになる．行21では，同様に，'*' より優先順位の低い '+' と '*' が follow 集合に加えられている．

3. follow 集合を使った再帰的下向き構文解析

前章の例で示したように，演算子の優先順位に従った解析法を再帰的下向き構文解析に組み込むにあたって，本論文では，演算子の優先順位だけを陽に使うのでなく，前章のような follow 集合を使う方法を提案する．（ここでは，Follow と follow を区別して使っている．）これは，文法の LL(1) 性を Follow 集合と First 集合から判定する際，衝突があったら，それを Follow 集合の部分集合で解決しようとする方法である．この方法は，演算子の優先順位だけを使う方法に比べて，より自然な再帰的下向き構文解析の拡張になる，より適用範囲が広くなる（その例は本章の後半で示す），エラーからの復帰に Follow 集合を使う場合との親和性もよい，などの利点がある．

```
1   procedure E':
2   begin    if next='$' then
3                 begin get(next); call E({'$'});
4                       if next='$' then
5                           begin get(next); return end
6                 end
7   end
8
9   procedure E(follow):
10  begin    if next='i' then
11                get(next);
12            else if next='(' then
13                begin get(next); call E({')'});
14                      if next=')' then
15                          get(next);
16                end;
17            while next∉follow do
18                begin if next='+' then
19                            begin get(next); call E(follow∪{'+'}) end
20                      else if next='*' then
21                          begin get(next); call E(follow∪{'+','*'}) end
22                end
23  end
```

図 2 文法 G4 の再帰的下向き構文解析プログラム
Fig. 2 Recursive descent parser for G4.

一般に，

$A \rightarrow A\alpha_1 | A\alpha_2 | \cdots | A\alpha_n | \beta_1 | \beta_2 | \cdots | \beta_m$

のように，左再帰性のある生成規則に対して，再帰的下向き構文解析を可能にするために

$A \rightarrow (\beta_1 | \beta_2 | \cdots | \beta_m)\{\alpha_1 | \alpha_2 | \cdots \alpha_n\}$ (2)

なる書き換えが行われる．ここで，一般には

$\varepsilon \in \mathrm{First}(\{\alpha_1 | \alpha_2 | \cdots | \alpha_n\})$

であるから，この(2)にも通常の LL 構文解析の方法を適用するためには，すくなくとも

$\mathrm{Follow}(A) \cap (\mathrm{First}(\alpha_1) \cup \mathrm{First}(\alpha_2) \cup \cdots \cup \mathrm{First}(\alpha_n)) = \phi$ (3)

でなければならない．以下では，主として，(2)の形で(3)が成り立たない場合について述べるが，一般に，

$A \rightarrow \alpha_1 | \alpha_2 | \cdots | \alpha_n,\ \varepsilon \in \mathrm{First}((\alpha_1 | \alpha_2 | \cdots \alpha_n))$ (4)

の形で(3)が成り立たない場合にも同様のことが言える．

ところで，前章の例のような文法では，(3)は成り立たない．たとえば，

E→E '+' E

E→i

で，'+' は左結合性を持つと宣言されている場合，これから得られる

E→i {'+'E}

においては，

Follow(E)∩First('+'E)={'+'}

である．今，この文法の文

"a+b+c"

の構文解析をする場合，最初にまずEが呼ばれ，a が i と認識される．その次の '+' は {'+'E} の '+' と見られて，{'+'E} のEが呼ばれることになるが，この2回目に呼ばれたEの中で b を i と認識した後の '+' を2回目のEの中の（{'+'E} の）'+' と見てしまうと，先に "b+c" が認識されることになってしまって '+' の左結合性に反する．この場合は，'+' を見たところで2回目のEから戻らなければならない．すなわち，この場合 '+' は，First('+'E) の '+' ではなく，左辺の E に対する Follow(E) の要素と考えなければならない．

以上の考察から，上記(3)が成り立たないときには，通常は，以下の方法で解決することにする．（本章の最後に示すように，この方法がとれない場合もある．）

（i） Follow(A) の代わりにその部分集合を使う．それをここでは follow(A) と書くことにする．follow(A) は構文解析中にAを呼び出すとき，その時のAの直後に来る可能性のある終端記号の集合である．

（ii） follow(A) と First(α_1) では，follow(A) を優先させる．

（iii） follow(A) はAが呼ばれるときに，動的に決める．

LL(1)文法の場合，(2)のAの通常の構文解析手続きは

```
procedure A
begin
    parse(β1|β2|…|βm);
    while next∈First(α1)∪First(α2)
      ∪…∪First(αn) do
    parse(α1|α2|…|αn);
end
```

の形であるが，上記(i)～(iii)に従う構文解析手続きは次のようにすればよい．ここで，

parse($\alpha_1 | \alpha_2 | \cdots | \alpha_n$)

は，($\alpha_1 | \alpha_2 | \cdots | \alpha_n$) の構文解析プログラムの略記法であるとする．

```
procedure A(follow)
begin
    parse(β1|β2|…|βm);
    while next∉follow
        and next∈First(α1)∪First(α2)
          ∪…∪First(αn) do
        parse(α1|α2|…|αn);
end
```

また，(4)の形の場合は，

```
procedure A(follow)
begin
    if next∉follow
            and next∈First(α1)∪First(α2)
              ∪…∪First(αn)
        then parse(α1|α2|…|αn);
end
```

とすればよい．そして，Aの呼び出しは

```
call A(follow)
```

という形になる．このように follow 集合を優先させる場合は，その呼び出しのときに実際にAの直後に現れうる終端記号だけの集合を与えるようにする必要がある．この follow 集合は次のようにして決めればよい．

Aがある生成規則の右辺で

$\gamma A\delta$

という形の中で現れたとする．α_i や β_i の中では $\gamma \neq \varepsilon$ としてもよい．この $\gamma A\delta$ という文脈でAを呼び出すときは，

（1） $\varepsilon \notin$First(δ) の場合：follow 集合は First(δ) であり，呼び出しは次のようになる．

call A(First(δ))

（2） $\gamma A\delta$ が前出の α_i や β_i の中でなく，$\varepsilon \in$First(δ) の場合：$\gamma A\delta$ を右辺にもつ生成規則の左辺の非終端記号をBとして，呼び出しは次の形とする．

call A(First(δ)∪follow$_B$)

ここで，follow$_B$ は手続き B が follow 集合のパラメータを持つ場合はそのパラメータであり，持たない場合は通常の Follow(B) である．

（3） $\gamma A\delta$ が α_i や β_i の中にあって $\delta=\varepsilon$ の場合：Aの直前に読んだ演算子（通常は γ の最後の終端記号である．複数の可能性がある場合は，**case** 文などで分岐する）をaとしたとき，aより優先順位の低い演算子の集合 less_than(a) を，このAの follow 集合に加える．すなわち，

less_than(a)＝{b|b∈T, a＞b}

として，呼び出しは次の形とする．

call A(follow∪less_than(a))

（4） $\gamma A\delta$ が α_i や β_i の中にあって $\varepsilon \in$First(δ) の場合：呼び出しは次の形とする．

call A((First(δ)−ε)∪follow∪less_than(a))

演算子順位で解決できない例

ストリング定数は，「'」または「''」で囲まれた文字列であるとし，「'」（または「''」）で囲んだ場合は文字列の中には「''」（または「'」）が入っていてもいいとする．その文法は，aを「'」とし，bを「''」とし，cをその他の文字として，

S→aAa|bBb

A→{b|c}

B→{a|c}

と表現できるが，前記の方法を使えば，似たような手続きになるAとBを一つにまとめることができる．すなわち，もとの文法を

S→aDa|bDb

D→{a|b|c}

のように書き換える．このままでは First(D)＝{a, b, c, ε} で Follow(D)＝{a, b} であるから，LL(1) ではないし，もとの文法とは異なる言語を定義している．ここで，aDa と bDb のDに対してそれぞれ，**call** D({a}), **call** D({b}) とすることによって，もとの文法と同じものを認識できる．これは演算子順位では解決できない例である．

ところで，（3）式が成り立たなかったときの解決法として，いつでも上記の follow 集合を優先させればよいというわけではない．その文法でどんな言語を定義したいかによって解決法は異なる．次の例でそれを示す．

follow(A) を優先させられない例

（3）式が成り立たないとき，いつでも follow を優先させればよいわけではない．次のよく知られているあいまいな文法がその例である．

S→'if' C 'then' S

S→'if' C 'then' S 'else' S

この 'else' には，それにもっとも近い，まだどの 'else' にも対応していない 'then' を対応させる，と約束すれば曖昧性はなくなる．通常の再帰的下向き構文解析では，これを実現するのに，まず生成規則をまとめて

S→'if' C 'then' S ['else' S]

として，その構文解析手続きを**図3**のようにしている．この文法の場合，Follow(S)∩First(['else' S])＝{'else'} であるが，図3のプログラムは，上記の約束に合わせるために，First(['else' S]) を優先させたものになっている．

4. 実 現 法

再帰的下向き構文解析を使った手書きのコンパイラを書く場合で，その中に演算子順位による解析も入れる場合は，前章の方針にしたがって書けばよい．手続き呼び出しのパラメータを追加するのは，再帰的下向き構文解析法の場合，一般に容易であり，follow 集合をパラメータとすることに問題はない．この方法は，文献 3) の PL/0 コンパイラのように，エラー処理のために follow 集合を使っている場合には，その拡張

```
procedure S;
begin if next='if' then
          begin get(next); call C;
                if next='then' then
                    begin get(next); call S;
                          if next='else' then
                              begin get(next); call S end
                          else return;
                    end
          end
end
```

図 3 if 文の再帰的下向き構文解析プログラム
Fig. 3 Recursive descent parser for if-statement.

として，ごく自然に取り入れることができる．

再帰的下向き構文解析のプログラムを生成するコンパイラ生成系で前章の方法を取り入れる場合は，たとえば，yacc のような演算子の順位の宣言法を導入する必要がある．さらに，ある非終端記号に対して，それから始まる部分文法（与えられた文法の中で，その非終端記号から生成されるものを規定している部分）を演算子順位も併用して規定することを宣言するようにしてもよい．また，前章の最後のような例もあるから，(2)式が満足されないものについて，それがfollow集合でなく First 集合を優先すべきものである場合に，そのことを宣言する必要がある．前章の less_than(a) の求め方は，演算子の優先順位の宣言の方法をどのように決めてあるかによって決まる．yacc のように，演算子の優先順位と左または右結合性を宣言する場合，less_than(a) は，a が左結合性を持つ場合は，a より優先順位の低いものと a からなる．右結合性の場合は a 自身は入らない．

生成系で(2)式が満足されない場合が見つかったら，その旨のメッセージを出力して本方式による処理に入る．さらに，less_than(a) を求める必要が生じたとき，a についての優先順位が宣言されていなかった場合は，

（1） 優先順位の宣言を要求する

（2） less_than(a) の値として適当な値を設定する(その旨のメッセージは出力する)

などが考えられる．(2)で，たとえば less_than(a) の値として空集合を設定するということは，a を右結合性を持つ優先順位の低い演算子とみなすことを意味する．

演算子の種類が多くなると，follow 集合として実行時に渡すべき情報が多くなる．それを効率よく実現するためには，たとえば，集合をビットベクトルで表現すればよい．

なお，yacc では，演算子が結合性を持たないことを宣言することもできる．たとえば，演算子「>」についてそれが宣言されたら，

```
a>b>c
```

といった式は許されない．この場合は，

```
parse('>' A)
```

の部分を，

```
if next='>' then
    begin get(next); call A(follow∪less_
            than('>')∪{'>'});
        if next='>' then error;
    end;
```

のようにすればよい．

5. 従来の方法との比較

再帰的下向き構文解析法の中に演算子順位による解析法も入れるということは，実際のいくつかのコンパイラの中で行われているようではあるが，論文として発表されているものはあまりない．1992年5月に comp.compilers というニュースグループで，LL 構文解析と LR 構文解析の優劣が論じられたとき，LL 構文解析の中での演算子順位による解析法も議論され，その実現法が5件報告されているが，論文の紹介は1件だけであった．

その論文 6) の方法は，2章の最初に上げた文法 G2 に対して，本質的には図1のようなプログラムを与えるものである．ただし，図1で手続きEと手続きTがほとんど同じ形をしていることに着目して，演算子の種類が多くなってこのような非終端記号が増えた場合でも，手続きは一つだけ作成し，その手続きへのパラメータとして非終端記号を同定するような番号を渡している．これによって，手続きは一つで済みはするが，構文解析時に手続き呼び出しネストが深くなるのは解消されていない．

その他の報告の多くは，プログラムの形は図2とほとんど同じであるが，パラメータとしては演算子の優先順位を示す数値を渡し，図2の **while** 文にあたるところで，それと次の演算子の優先度との大小を判定するものである．

それに対して，本論文では，通常の LL 構文解析ができない部分を，従来 LL(1) 性の判定のために使われていた Follow 集合の部分集合を動的に求めることによって解決するという，より一般的な方法を与えている．

なお，文献 2) の後半では，3章の最後に上げた if 文の問題を扱っている．そこでは，生成規則がある条件を満足する場合に Follow 集合でなく First 集合を優先させることになっているが，それは納得できない．if 文の生成規則を見ただけではそれは決定できないはずである．この場合にも follow 集合を優先させることはできる．ただし，その場合は，First 集合を優先させた場合とは異なる言語を定義したことになる．

6. おわりに

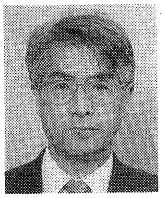
本論文では，再帰的下向き構文解析の枠組みの中に，演算子の優先順位による構文解析法を取り入れる一つの方法を提案した．演算子順位構文解析法を取り入れることによって，演算子の種類の多い文法における式を効率良く構文解析することができる．また，生成規則だけでなく演算子順位も使って文法を表現することによって，文法を分かりやすくする効果も期待できる．従来の実現法では，演算子の優先順位を示す数値を再帰的手続きにパラメータとして渡すものが多かったが，本論文では，Follow 集合の部分集合を渡すという，より一般的な方法を提案した．この部分集合は，再帰的下向き構文解析でエラー処理のために良く使われる集合とほぼ同じものであるので，そのようなエラー処理方式をとる再帰的下向き構文解析プログラムの中にはごく自然に取り入れることができる．

謝辞 本論文の内容に対して貴重なご意見を頂いた，東京工業大学佐々政孝教授に感謝する．

参考文献

1) Johnson, S. C.: Yacc—Yet Another Compiler Compiler, *Comp. Sci. Tech. Rep.*, No. 32, Bell Lab. (1975).
2) Aho, A. V., Johnson, S. C. and Ullman, J. D.: Deterministic Parsing of Ambiguous Grammars, *Comm. ACM*, Vol. 18, No. 8, pp. 441-452 (1975).
3) Wirth, N.: *Algorithms+Data Structures=Programs*, Prentice-Hall (1976).
4) 中田育男：コンパイラ，産業図書 (1981).
5) Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers, Principles, Techniques, and Tools*, Addison-Wesley (1986).
6) Hanson, D. R.: Compact Recursive-descent Parsing of Expressions, *Softw. Pract. Exper*, Vol. 15, No. 12, pp. 1205-1212 (1985).

(平成4年7月3日受付)
(平成4年11月12日採録)



中田 育男（正会員）

1935年生．1958年東京大学理学部数学科卒業．1960年同大学大学院修士課程修了．1960～79年(株)日立製作所中央研究所，同システム開発研究所勤務．1979年4月より筑波大学電子・情報工学系教授．理学博士．プログラム言語，言語処理系，ソフトウェア工学などに興味を持っている．著書「コンパイラ」(産業図書)，「基礎FORTRAN」(岩波書店)．ソフトウェア科学会，電子情報通信学会，ACM, IEEE 各会員．

山下 義行（正会員）

1959年生．1982年大阪大学理学部物理学科卒業．1982年～86年日立マイクロコンピュータエンジニアリング(株)勤務．1987年～89年筑波大学博士課程工学研究科在学．1989年東京大学大型計算機センター助手．1992年4月より筑波大学電子・情報工学系講師．工学博士．プログラミング言語，コンピュータグラフィックスなどに興味を持つ．情報規格調査会 SC 22/Fortran WG 委員．日本ソフトウェア科学会会員．