高いモジュラリティと拡張性を持つ構文解析器

一 杉 裕 志†

高いモジュラリティと拡張性を持つ再帰下降パーザの構築方法について述べる．このパーザは，拡張可能 Java プリプロセッサ EPP に使われている． EPP は， plug-in を追加することで Java の文法を拡張し，機能を追加できるプリプロセッサである．本論文が提案する方法では，再帰下降パーザを細かい mixin に分割して記述する．複数の mixin を直列につなぎ合わせ，１つのクラスにすることによりパーザが構築される．新たな mixin を定義し追加することで，演算子の追加など広範囲に渡る文法拡張を行なうことができる．

Modular and Extensible Parser Implementation using Mixins

†

This paper describes a method to construct highly modular and extensible recursive descent parser. This parser is used in an extensible Java pre-processor, EPP. EPP can be extended by adding plug-ins which extend Java syntax and add new language features. The EPP's parser consists of small mixins. A recursive descent parser class is constructed by composing these mixins. The syntax accepted by the parser can be extended by adding new mixins.

1. はじめに

筆者は，部品を追加することで Java 言語[7] の文法を拡張し新たな機能を追加できる拡張可能 Java プリプロセッサ EPP[9),10)] を開発した．本論文では， EPP の実装に用いた，高いモジュラリティと拡張性を持つ再帰下降パーザの構築方法について述べる．

EPP 設計時の目標として，「非常に広い範囲の拡張が行なえること」，「拡張部品の実装の自由度が高いこと」，「複数の拡張機能を同時に利用できること（以下 composability と呼ぶ）」などがあった．

あるシステムの機能を拡張する手段として，最も拡張範囲が広く実装の自由度が高い方法は，そのシステムのソースコードを編集し，直接変更を加えることである．しかし，この方法では，独立して作られた２つ以上の拡張を，同時に使うことは簡単にはできない．すなわち composability がない．

多くの拡張可能言語では，宣言的記述により新しい構文とその意味を定義できる．この方法は composability は高いが，拡張の範囲と実装の自由度は，記述言語の記述力によって制限されてしまう．

EPP は，「ソースコードの編集」という方法が持つ拡張範囲と実装の自由度を一切失うことなく，できる限り高い composability を持つ言語拡張の実装を可能とするシステムである． EPP のパーザの実装においては，差分プログラミングの機構である mixin を利用し，普通の汎用プログラミング言語のモジュールとして，パーザの部品を記述するアプローチをとった．このパーザは，以下の特徴を持つ．

- 新しい構文，演算子などを定義する部品をあとから追加可能である．文法拡張の部品は差分プログラミングのスタイルで書かれているため，複数の拡張部品を同時に組み合わせて使うことができる．
- モジュラリティが高く，部品単位で分割コンパイルも可能である．ソースコードを公開せずに文法拡張部品を配布できる．
- 汎用手続き型言語による記述のため，文脈依存，大域脱出など， BNF 記法では書きにくい ad-hoc な処理が行ないやすい．
- エラー処理，行番号処理も実現できる．
- symbol と mixin のメカニズムさえあれば，どのプログラミング言語でも容易に実現できる．

本論文の構成は次のようになっている．2章では EPP の概要を述べる． 3章では本論文のパーザの実装に必要な言語機能である symbol と mixin について述べる．

† 電子技術総合研究所
Electrotechnical Laboratory

4章と5章では mixin による拡張可能なパーザの記述方法と，特殊な処理の実装方法について述べる．6章ではパーザ実装方法の評価を行ない，7章で関連研究について述べ，最後に8章でまとめを行なう．

2. 拡張可能 Java プリプロセッサ EPP

EPP は，Java 言語に拡張機能を追加できるようにするプリプロセッサである．使いたい EPP plug-in を Java ソースコードの先頭で `#epp` *name* と書いて指定することで，Java の文法を拡張し，新しい機能を追加することができる．plug-in 同士の衝突が起きなければ，複数の plug-in を同時に取り込むこともできる．プリプロセッサが出力する Java ソースコードは，普通の Java コンパイラを通り，普通にデバッグすることができる．

EPP は，Java 言語を拡張するためのプリプロセッサとしてだけでなく，言語研究者による新しい言語機能実験の道具，拡張 Java の実装のためのフレームワーク，Java 言語のソースコード解析・変換ツールのためのフレームワークとして用いることもできる．

EPP のソースコード自身も「EPP で拡張された Java」で書かれており，「Common Lisp[15] で書かれた EPP」でブートストラップされた．バイナリは Java が動く環境ならばどこでも動く．

3. symbol と mixin

この章では，本論文の拡張可能パーザの実装に必要な言語機能である symbol と mixin について述べる．

3.1 Java 上での symbol の実現

symbol は lisp などの言語が持つデータ型で，以下の特徴を持つ．

(1) C 言語の enum 文で宣言された定数に近いが，あらかじめ有限個の要素を宣言しておく必要がない．ソースコード中必要に応じて任意の名前（文字列）を持つ symbol を使うことができる．
(2) 文字列リテラルにも近いが，文字列とは違い，実行時にはポインタの比較だけで高速に同一性の判定ができる．
(3) 名前を指定して symbol を動的に生成することもできる．

図1は，EPP の “Symbol plug-in” を使った Java プログラムの例である．Symbol のリテラルは，コロンとそれに続く identifier または文字列リテラルで表される．

Symbol plug-in は，以下のようにして Java 上での symbol を実現している．プログラム中の symbol リテラルは，それぞれ個別の private static final 変数への参照に変換される．そして，その変数は，クラスファイルの load 時に，`Symbol.intern("`*name*`")` という static method 呼び出しの返値に初期化される．`Symbol.intern("`*name*`")` は，ハッシュ表を検索し，もし同じ名前の Symbol クラスのインスタンスがすでにあればそれを返し，もしなければ新たに生成してハッシュ表に登録してそれを返す．これにより，同一の名前を持つ symbol リテラルは，同じ identity を持つインスタンスを参照することが保証される．実行時には static 変数の値を参照するだけなので，ハッシュ表検索のオーバーヘッドはない．

```
#epp jp.go.etl.epp.Symbol
import jp.go.etl.epp.epp.Symbol;

public class TestSymbol {
  public static void main(String args[]){
    Symbol x = :foo;
    Symbol y = :"+";
    System.out.println(x == :foo); // true
    System.out.println(y == :foo); // false
  }
}
```

図1 Symbol plug-in を使ったプログラム例
Fig. 1 A program using Symbol plug-in

3.2 Java 上での Mixin の実現

3.2.1 mixin とは何か

mixin[4] は，抽象サブクラスとも呼べるものである．通常オブジェクト指向言語では，サブクラスを定義する時に特定のスーパークラスの名前を指定する．mixin は，定義時に特定のスーパークラスを指定しないで定義されたクラスである．mixin は，後で他のクラスによって多重継承され，直列化されることによりスーパークラスが決まることを想定して定義される．

Bracha は，mixin のメカニズムだけあれば，Smalltalk, BETA, CLOS が持つ継承機構と同じことができることを示した[4]．VanHilst は，この mixin-based inheritance をさらに発展させた方法で，従来のオブジェクト指向よりもプログラムの再利用性を向上させる設計法を提案している[16]．

なお，C++ は多重継承ができるが，スーパークラスが直列化されないので，それを使って，mixin-based inheritance はできない．C++ プログラミングにおいて，多重継承によって共通スーパークラスが生じないように定義されたクラスを mixin と呼ぶ場合があるが，これはここで言う mixin とは異なるので注意が必要で

```
class Foo {
  void m(char c){
    if (c == 'B') {
      doB();
    } else if (c == 'A') {
      doA();
    } else {
      doDefault();
    }
  }
}
```

図 2 ネストした if 文を含むメソッド定義
Fig. 2 A method definition which uses nested if statements.

ある.

mixin は Design Pattern[6] の中の decorator pattern と似ている. しかし mixin は, decorator のような実行時の部品のつなぎかえはできない. 一方で, decorator は, あらかじめ固定されたインターフェースしか持てないという制約があるが, mixin はクラスに新たなメソッドを追加することができる.

3.2.2 mixin を使ったプログラム例

EPP の "SystemMixin plug-in" を用いて mixin-based inheritance ができる☆. SystemMixin plug-in で拡張された文法を用いて, mixin の機能について説明する.

まず, 図 2は, ネストした if 文を含むメソッド `m` を普通の方法で記述したものである.

このプログラムを, 3つの mixin に分割して記述すると, 図 3のようになる. ここで `original` という名前のメソッド呼び出し式は, SystemMixin plug-in で導入されたもので, 通常のオブジェクト指向言語での super のメソッド呼び出しに相当する.

このように, mixin を用いることで従来は一枚岩でしか書けなかった1つのメソッドを, 複数の「メソッドの断片」に分割して記述することができる. 後で, 複数の mixin を選択して組み合わせることで, 1つのクラスを構築することができる.

3.2.3 Mixin の実現方法

EPP の SystemMixin plug-in は, mixin に含まれるメソッドの断片を全て Java のクラスに変換することで mixin を実現している. メソッド呼び出し式は, レシーバのオブジェクトが持つハッシュ表を検索し, メソッドの断片を実現するオブジェクトを取り出し, それの `call` という名前のメソッドを呼び出す式に変換される. この実現方法は, 実行効率が悪く, また mixin を使って定義されたクラスと本来の Java 言語のクラスを同じように扱えないという問題がある. しかし, mixin 単位の分割コンパイルが可能なので, この方法を採用した.

☆ SystemMixin plug-in が提供する機能は, 本来の mixin とは違い, 差分プログラミングの単位はクラスではなくシステム全体である. しかし, 本論文中ではクラスはシステム内に1つしかないため, 本来の mixin と同じと考えてよい.

```
SystemMixin Skeleton {
  class Foo {
    define void m(char c){
      doDefault();
    }
  }
}
SystemMixin A {
  class Foo {
    void m(char c){
      if (c == 'A') { doA(); }
      else { original(c); }
    }
  }
}
SystemMixin B {
  class Foo {
    void m(char c){
      if (c == 'B') { doB(); }
      else { original(c); }
    }
  }
}
```

図 3 mixin によるメソッド定義
Fig. 3 A method definition by mixins.

なお, C++ でも template を使って, superclass をパラメタ化することによって, mixin-based inheritance が実現できる. 図 3のプログラムは, C++ の template を用いて書くと図 4のようになる. `Skeleton`, `A`, `B` の3つを組み合わせたクラスは, `B<A<Skeleton>>` となる. ただし, template による mixin は, 分割コンパイルすることはできない.

3.3 EPP を構成する mixin

EPP が持つパーザは, `Epp` という名前のただ1つのクラスとして定義されている. ただしクラス定義は複数の mixin に分割されている. EPP を起動すると, まず「標準の Java 言語パーザを構成する mixin 」と,

```
class Skeleton {
public:
  void m(char c){ doDefault(); }
};
template<class Super>
class A : public Super {
public:
  void m(char c){
    if (c == 'A') { doA(); }
    else { Super::m(c); }
  }
};
template<class Super>
class B : public Super {
public:
  void m(char c){
    if (c == 'B') { doB(); }
    else { Super::m(c); }
  }
};
```

図 4 C++ の template による mixin 定義
Fig. 4 Mixins defined by the template mechanism of C++.

「ソースコードの先頭で指定された plug-in を構成する mixin 」をすべてつなぎ合わせて１つのパーザ（Epp という名前のクラス）を構築する．そしてそのクラスのインスタンスを１つ作り，起動メソッドを呼び出して，入力ソースコードの処理を開始する．

4. 拡張可能なパーザの実装方法

4.1 トークンの表現

字句解析ルーチンを拡張可能にする場合，字句解析ルーチンが返すデータ型の定義をどう拡張可能にするかという問題がある．これには２通りの方法が考えられる．

(1) 字句解析ルーチンを拡張する手段と同時に，トークンのデータ型定義を拡張する手段も提供する．
(2) 拡張の範囲をあらかじめ制限し，その範囲の中のすべてのトークンを表現できるデータ型を用意する．

前者の方法では，字句解析ルーチンの拡張にともないデータ型も変更されるため，それを処理する構文解析ルーチンも影響を受け，ソースコードの変更や再コンパイルが必要になる可能性がある．そこで， EPP では後者の方法を採用した．トークンのデータ型は，幅広い範囲の拡張を想定した構造になっており，たいていの言語拡張に対して，データ型を変更することなく対応できる．

具体的には，全てのトークンは，「リテラル」か「symbol」のいずれかのデータ型で表現される．

リテラルは，リテラルの種類を表すタグと，リテラルの中身を表す文字列から構成される．例えば， 123 という整数リテラルは int というタグと "123" という文字列を持つデータとして表現される．この枠組内で，もともとの文法にないリテラルも，新たなタグを割り当てるだけで表現可能である．

リテラル以外の全てのトークン，すなわち識別子，キーワード（if, while など），演算子，記号（セミコロン，かっこなど）はすべて symbol で表される．キーワードと識別子を区別していないので，新たな構文を追加する際，字句解析ルーチンを全く変更せずに，構文解析ルーチンを拡張するだけで実現できる．

4.2 再帰下降パーザ

まず， mixin を用いない従来のスタイルで，非終端記号の構文解析を行なうメソッドの実装方法を示す．（次の節でこのメソッドを mixin に分割する．）

例として，次のように，前置演算子，かっこ，右結合２項演算子，左結合２項演算子，後置演算子の選択肢を持つ生成規則を考える☆．

```
Exp → ++ Exp | ( Exp ) | Term += Exp | Term
        | Exp + Term | Exp ++
```

この生成規則は，書き換えによって再帰下降パーザで構文解析できる形，すなわち LL(1) 文法になる．（詳しくは付録を参照．）再帰下降パーザとは，各非終端記号ごとに，それを構文解析し構文解析木を返す関数を定義するスタイルのパーザである[3]．

上記の非終端記号 Exp を構文解析する（mixin を用いない）再帰下降パーザは図 5のようになる． expTop は前置演算子および右再帰でも左再帰でもない選択肢（かっこなど）， expRight は右結合演算子の選択肢， expLeft は左再帰の選択肢（後置演算子および左結合演算子）を構文解析して，抽象構文木を返すメソッドである．

ここで， lookahead は現在注目しているトークンを返すメソッド， match は現在のトークンが引数の値と異なる場合はエラーを出し，等しい場合は読み捨てて次のトークンを読むメソッド， matchAny は無条件にトー

☆ 実際の言語の文法では，このように１つの非終端記号にかっこと２項演算子，あるいは右結合２項演算子と左結合２項演算子が混在することはない．そのような文法では例えば， a + (b) や a + b += c が生成できず，人間の直感に反する言語になるからである．

クンを読み捨てるメソッドである．

このプログラムが作る抽象構文木の形は，生成規則から期待される通りの形になる．例えば，a += b += c は，(+= a (+= b c)) に，a + b + c は (+ (+ a b) c) という形の木になる．

4.3 拡張可能な再帰下降パーザ

図5のプログラムを複数の mixin に分割することにより，高いモジュラリティと拡張性を持った形にすることができる．図5のプログラムの if 文を取り除き，骨組みだけにすると図6になる．この mixin で定義される exp は，次の生成規則を構文解析するメソッドである．

Exp → Term

このプログラムの expTop，expRight，expLeft というメソッドを mixin を追加して拡張することで，非終端記号に新たな選択肢を追加できる．図7は，左結合２項演算子を追加する mixin の定義例である．

EPP では，数十種類の非終端記号が，図6のような骨組みを定義する mixin と，図7のような選択肢を追加する mixin の集合として定義されている．これらの定義を容易にするマクロが提供されており，図6の mixin は，次の１行で定義できる．

```
defineNonTerminal(exp, term());
```

また，図7の左結合２項演算子を追加する mixin も，次の１行で定義できる．

```
defineBinaryOperator(Plus, :"+", exp);
```

4.4 字句解析ルーチンの差分的拡張

字句解析ルーチンも再帰下降スタイルで書くことによって，差分的に拡張可能になる．EPP の字句解析ルーチンは，主に以下のメソッドから構成されており，これらの動作を mixin で拡張することができる．

- readToken
- readId
- readNumber
- readOperator
- readStringLiteral
- readCharLiteral
- readTraditionalComment
- readEndOfLineComment

例えば，図8は，// の直後に : がある場合は，それ以後をコメントとみなさないようにする mixin である．（この機能は，Java 言語と互換性を保ったまま機能拡張するために用いることができる．例えば

```
//: assert(predicate);
```

のように，通常の Java コンパイラには単なるコメントとして扱われ，EPP を通した時だけ assert マクロとして機能するようなプログラムを書くことができる．）

```
Tree exp() {
  Tree tree = expTop();
  while (true){
    Tree newTree = expLeft(tree);
    if (newTree == null) break;
    tree = newTree;
  }
  return tree;
}
Tree expTop() {
  if (lookahead() == :"++"){
    matchAny();
    return new Tree(:"preInc", exp());
  } else if (lookahead() == :"("){
    matchAny();
    Tree e = exp();
    match(:")");
    return new Tree(:"paren", e);
  } else {
    return expRight(exp1());
  }
}
Tree expRight(Tree tree) {
  if (lookahead() == :"+="){
    matchAny();
    return new Tree(:"+=",tree,exp());
  } else {
    return tree;
  }
}
Tree expLeft(Tree tree) {
  if (lookahead() == :"+"){
    matchAny();
    return new Tree(:"+",tree,exp1());
  } else if (lookahead() == :"++"){
    matchAny();
    return new Tree(:"postInc", tree);
  } else {
    return null;
  }
}
Tree exp1() { return term(); }
```

図5 再帰下降パーザの一部
Fig. 5 A part of a recursive descent parser.

```
SystemMixin Exp {
  class Epp {
    define Tree exp(){
      Tree tree = expTop();
      while (true){
        Tree newTree = expLeft(tree);
        if (newTree == null) break;
        tree = newTree;
      }
      return tree;
    }
    define Tree expTop(){
      return expRight(exp1()); }
    define Tree expRight(Tree tree){
      return tree; }
    define Tree expLeft(Tree tree){
      return null; }
    define Tree exp1(){
      return term(); }
  }
}
```

図 6 拡張可能なパーザの骨組み
Fig. 6 A skeleton of a extensible parser.

```
SystemMixin Plus {
  class Epp {
    Tree expLeft(Tree tree) {
      if (lookahead() == :"+") {
        matchAny();
        return new Tree(:"+",tree,exp1());
      } else {
        return original(tree);
      }
    }
  }
}
```

図 7 左結合 2 項演算子の追加例
Fig. 7 A mixin which defines a left associative binary operator.

4.5 パーザ部品の削除・再定義

EPP は，差分プログラミングによる文法拡張以外の拡張手段も提供している．

差分プログラミングによる文法の拡張は，新たなパーザ部品の追加しかできず，既存の部品の削除はできない．また，あらかじめ用意された hook （メソッド）を用いた拡張しかできない．これだけでは，文法拡張の範囲はかなり制限されてしまう．

```
SystemMixin CommentPragma {
  class Epp {
    Token readEndOfLineComment
                (EppInputStream in){
      if (in.peekc() == ':'){
        in.getc();
        return readToken(in);
      } else {
        return original(in);
      }
    }
  }
}
```

図 8 字句解析ルーチンの差分的な拡張定義例
Fig. 8 A mixin which extends the lexical analyzer.

差分プログラミングでない拡張方法として，まず mixin を追加する際に original メソッドを呼び出さずに，もとのメソッドを無視するという方法がある．例えば図 6のメソッド exp1 を再定義することにより，演算子の優先順位を変更することができる．

さらに， EPP では plug-in のロード時に，既存のパーザ定義部品 (mixin) を取り除く手段を提供している．この手段を用いれば，原理的に任意の文法の変更が可能となる．

しかし，これらの手段を用いた plug-in は，差分プログラミングだけを用いた拡張と比べて， composability が低いという欠点がある．拡張機能と composability のトレードオフを， plug-in プログラマーか， plug-in ユーザが行なう必要がある．

5. 特殊な処理

5.1 バックトラック処理

EPP では，字句解析ルーチンおよび構文解析ルーチンで，明示的にバックトラックを行なうことができる．

図 9は，バックトラックを使って，新たなトークン ** を追加する例である．

引数の EppInputStream は，任意の長さのバックトラックが可能な入力ストリームである．入力ファイル全体を文字の配列にしてメモリ上に持つことで，任意長のバックトラックを実現している．

5.2 文脈依存処理

再帰下降パーザでは，字句解析ルーチンおよび構文解析ルーチンにおける文脈依存の処理は容易に実現できる．大域変数 (static 変数) に文脈情報を入れておけば

```
SystemMixin NewOp {
  class Epp {
    Token readOperator(EppInputStream in){
      if (in.peekc() == '*') {
        int p = in.pointer();
        in.getc();
        if (in.getc() == '*') {
           return :"**";
        }
        in.backtrack(p);
        return original(in);
      } else {
        return original(in);
      }
    }
  }
}
```

図 9 トークンの追加例
Fig. 9 A mixin which defines a new token.

よい.

5.3 エラー復帰処理

構文解析時のエラーからの復帰は, Java の例外処理機構を使って素直に実現することができる. 例外を catch で受け取り, 特定のトークンが現れるまで読み飛ばしを行なえばよい.

5.4 行番号処理

構文解析時に生成する Tree に, 「その構文が開始した行番号」の情報を入れておけば, 後の意味解析時にエラーが起きたときに, ユーザにわかりやすいエラーメッセージを出力するために利用できる. また EPP ではこの情報を, ソースコードの各行を変換後も同じ行に出力するためにも用いている.

この機能は, lisp ならば動的スコープを持つ変数を, Java ならば static 変数, スタック, try-finally 構文を用いることで容易に実現できる. 図 10は, Java で実現する場合の, メソッド exp の定義例であり, すべての非終端記号のメソッドを同様に定義する. Tree のコンストラクタは, static 変数 LineNumber.stack のスタックの先頭を見ることで, その Tree が開始した行番号を得ることができる.

```
Tree exp(){
  LineNumber.stack
    .push(currentLineNumber());
  try {
    ... 図 6 と同様 ...
  } finally {
    LineNumber.stack.pop();
  }
}
```

図 10 行番号処理
Fig. 10 Managing line number information.

6. 評 価

6.1 Java 文法の記述

EPP は, 本論文で述べた方法で実装された, JDK1.1 対応の完全な Java 言語パーザを有する. Java 文法を定義する部分は, 105 個の mixin から構成される. そのうち, 29 個が図 6のような非終端記号の骨組みを定義する mixin である.

実装する際に, 以下の箇所で明示的なバックトラックを行なった.

- constructor と method, field との区別
- static method/field と static initializer の区別
- method と field の区別
- ローカル変数宣言と文の区別

文法を書き替えることで LL(1) で解析できるものもあるが, 文法の拡張性, モジュラリティが悪くなるため, バックトラックを用いた.

また, field アクセス構文と cast の構文は, モジュラリティの悪い実装となっている. これらの部分は, パーズ時に型情報が得られないとモジュラリティの高い実装は不可能である.

6.2 効 率

7218 行ある EPP のソースコード自身を EPP で処理する速度を計測した結果, 以下のようになった.

- MMX Pentium 233MHz, Windows95, Microsoft SDK2.01 : 約 30 秒
- UltraSPARC 200MHz, Solaris2.5.1, JDK1.1.3 : 約 40 秒

処理は, ソースの構文解析, 拡張構文のマクロ展開, 変換後のソース出力の 3 つ部分から成るが, 構文解析が処理時間の多くの部分を占めている. 実用上問題ないスピードであるが, パーザの速度としては決して速くない. 遅さの原因は 3 つ考えられる.

(1) Java インタープリタのオーバーヘッド.
(2) mixin のメソッド呼び出しのオーバーヘッド.
(3) 本論文が提案するパーザ記述方法の本質的遅さ. (if-then-else による逐次的な選択肢の探索, 中身のないメソッドの不必要な呼び出しなど.)

このうち現在の実装による mixin のメソッド呼び出し

オーバーヘッドは，通常の Java class のメソッド呼び出しと比べておよそ１桁遅くなっている．これは， mixin で定義されたメソッドの断片を小さい Java object で実現し，実行時にハッシュ表を検索してメソッド呼び出しを行なっているためである． mixin の実装方法については，今後，高速化の余地が残っている．

if-then-else による逐次的な選択肢の探索のオーバーヘッドもかなり大きい．これを高速化する方法としては， if-then-else をテーブルの検索に変換するなど，このパーザのソースコードに特殊化した最適化トランスレータの実装が考えられる．

一般にバックトラックが多く起きると構文解析の効率は悪くなるが，少なくとも Java の文法では，深刻な速度低下を起こすバックトラックは起こらない．JDK1.1.1 の java.util.Vector のソースコードを EPP で構文解析したみた結果，トークンは 1214 個含まれているが，バックトラックが 86 回起き，その結果メソッド readToken が 172 回余分に呼ばれている．従って，バックトラックによる readToken を呼ぶ回数の増加は約 14% であり，深刻な増加にはなっていない．

6.3 デバッグのしやすさ

宣言的記述によるパーザ生成システムは普通，文法の衝突や曖昧さを検出し，ユーザに警告する機能を持つ．しかし，本論文が提案するパーザは，このような機能は提供できない． EPP のように，すでに完成された既存の言語の文法を，差分的に拡張する用途では問題ないが，全く新しい言語の文法を設計する道具としては向いていない．

しかし，宣言的記述から，本論文で述べた mixin を機械的に生成するパーザ生成システムの構築も可能であろう．

なお，再帰下降パーザの局所的なデバッグは容易である．標準の Java デバッガを使って stack trace を見たり， print 文を入れることによって普通のプログラムと同じようにデバッグできる．

7. 関 連 研 究

LL(k) に基づく最近のパーザ生成システムとして，ANTLR[1], JavaCC[2] がある．これらのツールの作者は， LALR(1) のようなボトムアップパーザに対するトップダウンパーザのメリットとして，デバッグしやすい，構文解析時に下向き・上向きに属性値を渡しやすい，などを挙げている．なお， ANTLR は，継承によって差分的に既存の文法を拡張する機能を持っている．また， JavaCC は，生成規則の一部に Java のコードを直接書くことができ，宣言的記述で書きづらい処理も行なうことができる．

コンパイル時 MOP(Meta Object Protocol[12]) を提供することによって言語を拡張可能にするシステムに， MPC++[11], OpenC++[5], JTRANS[13], OpenJava[17] がある．これらのシステムは， EPP と同様に，構文解析後の抽象構文木に対して複雑な変換処理を行なうことを目的としている．文法も限られた範囲の中で拡張可能になっている．例えば MPC++ では，新しい演算子や文などを追加できるようになっている．

文法の定義のモジュール化が行なえるコンパイラ生成システムとして， Eli[8] がある． Eli は，属性文法に基づいた文法と意味の定義から，言語処理系を自動生成する．既存の定義部品から一種の継承を行なうことによって，新たな言語を定義することができる．

文法を変更できる「拡張可能言語」は従来から数多くあるが，その多くは on-the-fly ，すなわち処理対象のプログラム中で新たな文法拡張を定義するスタイルをとっている． lisp や C のマクロも on-the-fly の文法拡張機能と言える． on-the-fly による文法拡張は，文法拡張のコードをあらかじめコンパイルしておけないという効率上の問題がある．また，「文法拡張を行なう構文」そのものを変更・拡張すると混乱が生じやすい．EPP では on-the-fly による文法拡張の方法はとっていないため，このような問題はない．

EPP と同様に，部品を追加することで文法の拡張が可能なプリプロセッサとして， Camlp4[14] という Objective Caml のプリプロセッサがある．宣言的記述により文法を差分的に拡張することができ，拡張部品は分割コンパイルできる．

8. ま と め

mixin を使って再帰下降パーザを細かく分割して記述することにより，高い拡張性とモジュラリティを持つパーザを記述する方法を述べた．このパーザは，差分プログラミングによって実装された mixin を追加することにより， composability の高い形で広い範囲の文法拡張が可能である．また，既存の mixin を取り除く手段を用いれば，原理的に任意の文法の変更が可能である．

謝辞 本研究の初期に拡張可能なパーザの記述方法について議論してくれた大阪大学の松下誠氏， EPP に関する議論をしてくれた電総研客員研究員 Yves Rouder 氏に感謝します．

参 考 文 献

1) : ANTLR home page. “http://www.ANTLR.org/”.

2) : JavaCC home page. “http://www.suntest.com/JavaCC/”.
3) Aho, A., Sethi, R. and Ullmann, J.: *Compilers: Principles, Techniques and Tools.*, Addison-Wesley (1987).
4) Bracha, G. and Cook, W.: Mixin-based Inheritance, *Proc. of ECOOP/OOPSLA'90* (1990).
5) Chiba, S.: A Metaobject Protocol for C++, *Proceedings of OOPSLA'95*, ACM Sigplan Notices, Vol. 30(10), Austin, Texas, ACM Press, pp. 285–299 (1995). “http://www.softlab.is.tsukuba.ac.jp/~chiba/openc++.html”.
6) Gamma, E., H., R., Johnson, R. and Vlissides, J.: *Design Patterns*, Addison-Welsley (1995).
7) Gosling, J., Joy, B. and Steele, G.: *The Java Language Specification*, Addison-Wesley (1996).
8) Gray, R., Heuring, V., Levi, S., Sloane, A. and Waite, W.: Eli: A Complete, Flexible Compiler Construction System, *Communications of the ACM*, Vol. 35, No. 2, pp. 121–131 (1992).
9) Ichisugi, Y.: EPP home page. “http://www.etl.go.jp/~epp”.
10) Ichisugi, Y. and Roudier, Y.: The Extensible Java Preprocessor Kit and a Tiny Data-Parallel Java, *ISCOPE'97, California*, LNCS 1343, pp. 153–160 (1997).
11) Ishikawa, Y.: Meta-level Architecture for Extendable C++, Draft Document, Technical Report Technical Report TR-94024, Real World Computing Partnership (1994). “http://www.rwcp.or.jp/lab/mpslab/mpc++/mpc++.html”.
12) Kiczales, G., des Rivieres, J. and Bobrow, D. G.: *The Art of Metaobject Protocol*, MIT Press (1991).
13) Kumeta, A. and Komuro, M.: Meta - Programming Framework for Java, *The 12th workshop of object oriented computing WOOC'97, Japan Society of Software Science and Technology* (1997).
14) Rauglaudre, D.: Camlp4 home page. “http://pauillac.inria.fr/camlp4/”.
15) Steele, G.: *Common Lisp the Language, 2nd edition*, Digital Press (1990).
16) VanHilst, M. and Notkin, D.: Using Role Components to Implement Collaboration-Based Designs, *OOSPLA'96* (1996).
17) 立堀, 千葉: ユーザにとって直観的なコンパイル時MOP, *SPA'98*, 日本ソフトウエア科学会 (1998). “http:// www.softlab.is.tsukuba.ac.jp/~mich/openjava/”.

付録 生成規則の書き換え

次のように定義される生成規則を書き換えて，再帰下降パーザとして書ける形にする．

```
Exp → ++ Exp | ( Exp ) | Term += Exp | Term
      | Exp + Term | Exp ++
```

まず *ExpTop* を導入し，生成規則を２つに分ける．

```
Exp → ExpTop | Exp + Term | Exp ++
ExpTop → ++ Exp | ( Exp ) | Term += Exp | Term
```

ExpLoop を導入し， *Exp* の左再帰を除去する．また， *ExpRight* を導入し， *ExpTop* を書き換える．

```
Exp → ExpTop ExpLoop
ExpTop → ++ Exp | ( Exp ) | Term ExpRight
ExpRight → += Exp | ε
ExpLoop → + Term ExpLoop | ++ ExpLoop | ε
```

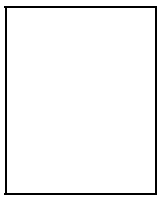
この文法は，すでに再帰下降で書けるが， *ExpLeft* を導入し *ExpLoop* をさらに以下のように書き直せる．

```
Exp → ExpTop ExpLoop
ExpTop → ++ Exp | ( Exp ) | Term ExpRight
ExpRight → += Exp | ε
ExpLoop → ExpLeft ExpLoop | ε
ExpLeft → + Term | ++
```

この生成規則に基づいた非終端記号を構文解析するプログラムが，図5である．ただし *ExpLoop* の末尾再帰呼び出しは loop に置き換えられ，メソッド exp に埋め込まれている．また，メソッド expLeft は，選択可能な選択肢がないことを特殊な値 null を返すことで表している．

なお最初の文法および書き換え後の文法は，いずれも曖昧である．例えば， ++ a + 1 という式は，（++ (+ a 1)）と（+ (++ a) 1）の両方に解釈できる．図5のプログラムでは，この式は（++ (+ a 1)）と構文解析される．

(平成1998年6月0日受付)
(平成1998年9月0日採録)



一杉 裕志（正会員）

1965 年生． 1990 年東京工業大学大学院情報科学専攻修士課程修了．1993 年東京大学大学院情報科学専攻博士課程修了．博士 (理学)．同年電子技術総合研究所入所．オブジェクト指向言語，並列・分散言語，拡張可能システムに興味を持つ．日本ソフトウエア科学会， ACM 各会員．