オープンソースの再利用による迅速なソフトウェア開発のための一手法——逆エンジニアリングツール開発の経験から

前 田 和 昭†

本稿では，既存の情報システムの一部を再利用するのに必要となることがあるツールの基本機能であるパーザを，オープンソースコンパイラを使って迅速に開発する手法について述べる．主要なプログラミング言語のソースコードを解析する逆エンジニアリングツールを従来の方法で開発するには，かなりの工数を必要とする．そこで，その工数を大幅に減らすために，パーザを 2 つに分ける手法と実装のためのパーザ生成系 MJay を考案した．MJay は，従来の LALR パーザである第 1 パーザを生成すると同時に，還元–還元競合を持たない第 2 パーザのための文法規則を生成する．これまでに MJay を使った経験では，C# や Visual Basic の逆エンジニアリングツールを短時間で開発することができた．

Rapid Software Development Using Open Source Software —From Experiences of Developing Reverse Engineering Tools

Kazuaki Maeda†

This paper describes a new source code analysis method which enables the rapid creation of parsers from source code of popular free and open source compilers. It is time-consuming and laborious to create a reverse engineering tool when the traditional parser development is used. MJay was developed to cut down the workload for such a work. The syntax analysis is performed by two parsers. MJay generates the first parser (which is a commonly used LALR parser) from traditional grammar rules. In addition to this, MJay generates other grammar rules for the second parser, which do not contain any reduce/reduce conflicts. The second parser is created rapidly using the generated grammar rules. The author's experiences found that the use of MJay enabled the rapid development of the reverse engineering tools for C# and Visual Basic using the source code of their compilers.

1. はじめに

最近，情報システムを開発するための技術としてサービス指向アーキテクチャ（Service Oriented Architecture）が注目されている[1),2)]．この技術を使い，既存のソフトウェアを取り込んで情報システムを開発する場合は，既存のソフトウェアの全体か一部をサービスという単位でラッピングし，新しいシステムのコンポーネントとして再利用する．しかしながら，元のソースコードを見直したり，修正したりすることがありうる．

既存のソースコードを再利用する場面では，逆エンジニアリングツールが有効だといわれている[3)]．逆エンジニアリングツールは，対象システムの構成要素とそれらの間の関係を明らかにし，抽象度の高いレベルでシステムを表現したものを生成する．たとえば，設計仕様書がなくなっていたり，ソースコードと不一致となっていたりする場合に，ソースコードを解析して設計仕様を自動的に生成する逆エンジニアリングツールが利用される．また，最近の商用の UML エディタの中には，逆エンジニアリング機能を持ち，ソースコードを読み込むことで，UML の一部を自動的に生成するものがいくつかある（Poseidon for UML[4)]，Magic Draw UML[5)]，Visual Paradigm for UML[6)] など）．

逆エンジニアリングツールを作成するには，プログラミング言語のソースコードを解析するプログラム（以下，パーザと記す）が必要となる．品質の高いパーザを開発するには非常に手間がかかるため，既存のコンパイラのソースコードを修正して利用することが多い．たとえば，JavaML[7)] コンバータの作成では，Java コンパイラとして有名な Jikes[8)] が使われてい

† 中部大学経営情報学部
College of Business Administration and Information Science, Chubu University

る．この作成のために，Jikes 内部でソースコードを表現する木を定義するために使われているクラスを多数書き換えている．既存のソースコードを修正する場合，逆エンジニアリングツールを使ったとしても，ソースコードをある程度読解してから修正するという手間のかかる作業が必要となる．

本稿は，容易に入手可能なコンパイラのソースコードから迅速にパーザを開発する手法について述べる．このパーザ開発手法は，逆エンジニアリングツールでソースコードを解析するためのもので，情報システムの開発・拡張において，既存のソースコードを有効に利用するために使われる．もし，プログラミング言語の仕様が変わったとしても，パーザを迅速に開発することができれば，新版の逆エンジニアリングツールを素早く提供できる．その結果，新しいプログラミング機能を活用した情報システムの開発を促進することになる．

ソースコードの有効活用という点では，最近，フリー☆なオープンソースソフトウェア（Free and Open Source Software 以下，FOSS と記す）の開発で，華々しい活躍を続けるプロジェクトがいくつも登場し注目を集めている．FOSS の代表的な例として，Linux[9]，Apache[10]，PostgreSQL[11]，OpenOffice.org[12]，Eclipse[13] などがある．それらの開発プロジェクトでは，地理的に世界中に分散するソフトウェア開発者たちが，公開されたソースコードを閲覧し，品質と機能の向上のために絶えず開発作業を続けている．

FOSS に関する書籍や報告書を数多く見かけるようになった．たとえば，日本国内では 2003 年 1 月に日本総合研究所による報告「オープンソースソフトウェアのセキュリティ確保に関する調査」が Web 上で公開され[14]，続いて 2003 年 8 月には，経済産業省の企画のもとでまとめられた報告書「オープンソース・ソフトウェアの現状と今後の課題について」が Web 上で公開された[15]．また，2003 年 3 月に 1 回目が行われた Asia Open Source Software Symposium[16] の開催回数はすでに 7 回に及んでいる☆☆．今までビジネスには縁遠かった FOSS が，ようやく表舞台で活動を始め，全世界でソフトウェアビジネスが変わろうとしているように思える．

現在の一般的な商用ソフトウェアでは，オブジェクトコードのコピーのみが利用者に提供されている．企業が高価な人件費をかけて作り上げたソースコードの中には，社外には漏らしたくない重要なアイディアや，生産性を上げるために長年にわたって苦労して開発したプログラム部品が含まれる．したがって，競合する他社に対して優位性を維持するためには，ソースコードを隠しておくのが当然とされてきた．FOSS は，これまでのビジネスの慣例とはまったく異なり，オブジェクトコードのコピーを無料（または安価）で公開し，かつ，ソースコードを無料公開する．最近の FOSS の広がりは，上記の特徴を反映し，「コスト削減」と「ベンダーロックインからの脱却」という 2 つが主な理由と考えられる[17]．

「Free and Open Source Software」という名前の由来は，ソースコードが公開され，そのソースコードを自由に閲覧・修正・利用できるところにある．しかし，ソースコードを読解し，たとえば他のソフトウェアの作成や修正に利用するなど，FOSS のソースコードをそれ自体の修正や拡張以外の目的で有効に使う場合はまれである．

そこで本稿では，広く使われているフリーなオープンソースコンパイラ（以下，FOSS コンパイラと記す）のソースコードを利用することで，パーザを迅速に開発する効果的な手法を提案する．提案するパーザ開発手法のことを，「2 段階パージング」と呼ぶことにし，その特徴は以下のとおりである．

- パーザを 2 つのステップに分割する．
- ステップ 1 では，FOSS コンパイラのソースコードを修正して利用する．ただし，パーザの処理が終了したところでコンパイラの処理を止めることにより，構文解析までを実行する．
- FOSS コンパイラの開発で使われているパーザ生成系を，新たに考案したパーザ生成系と交換することで，ステップ 2 のための文法規則を自動生成する．また，ステップ 2 を駆動するための情報を XML 文書として記録できるようにする．
- ステップ 2 では，ステップ 1 で記録した XML 文書を使って，ステップ 1 で実行した構文解析と同じ動きを再現する．ただし，自動生成された文法規則から，通常のパーザ生成系を使ってステップ 2 のパーザを生成するため，生成されたパーザの機能は構文解析のみであり，抽象構文木を生成する機能は持たない．

2 段階パージングを使ったパーザを含む逆エンジニアリングツールは，2006 年春から市販製品（.NET 用統合開発環境）の一部として組み込まれて出荷されて

☆「フリー」は free の訳語として使う．free には，無料と自由の 2 つの意味があるが，ここでは自由の意味とする．
☆☆ 2006 年 4 月現在.

いる☆．出荷後約半年の間，いくつかのバグ報告を受けた．しかし，報告を受けたバグは，2 段階パージングの実装に関するものではなく，元となる FOSS コンパイラの修正に誤りがあったことが原因となっている．これは 2 段階パージングによるパーザ開発の注意点を示すもので，5 章で考察する．

次の 2 章では，パーザ開発に関する関連技術と 2 段階パージングの動機について述べる．3 章では，2 段階パージングを使った迅速なパーザ開発について詳細を述べる．4 章では，性能評価のための実験について述べ，さらに実験結果を検討する．5 章で考察を行ったあと，6 章で本稿をまとめる．

2. 関連技術

2.1 Yacc を使ったパーザ開発

1970 年以前，パーザを開発するには複雑な作業が必要だった．ところが，1970 年代に入り，パーザ生成系の Yacc[18] が開発されたことによって，パーザの開発効率が格段に向上することとなった．Yacc は，ユーザが定義した文法規則を読み込み，C 言語で書かれたパーザを生成する．生成されたパーザは，基本的に，字句解析を通して作られたトークン列を読み込み，与えられた文法規則に従っているかどうかをチェックする機能を持つ．もし，構文上の誤りがある場合は，エラーメッセージを出力する．

通常のコンパイラでは，構文を解析するときに抽象構文木を作り出すことが多い．抽象構文木とは，基本的に，操作を表すノードと，操作に対する引数を表す子ノードからなる木である．Yacc の入力となる文法規則には，適合したときに起動されるアクションを付加することができる．このアクションを利用することで，構文を解析していく途上で抽象構文木を作り上げる．できあがった抽象構文木は，コンパイラの次のフェーズ（意味解析やコード生成など）に渡すのが一般的となっている．

Yacc が生成する C 言語で書かれたパーザを実行するには，関数 yyparse() を一度だけ呼び出せばよい．yyparse() の内部では，字句解析のための関数 yylex() が必要に応じて呼び出され，トークンが 1 つずつ取り出される．もし，構文の正しさをチェックするだけであれば，yyparse() を呼び出し，その返却値を確認すれば，すぐに処理を終了してよい．

パーザを開発するときのデバッグ機能として，Yacc にはパーザの内部動作を出力する機能がある．たとえば，以下のような単純な C#プログラムの文を構文解析することを考える．

```
using System;
```

この文を入力として，Yacc を使って開発した C#パーザを実行すると，**図 1** のような出力を得ることができる．この出力から，トークン USING，IDENTIFIER，SEMICOLON を順番に読み込みながら，パーザの動作であるシフト（shift）と還元（reduce）を繰り返し，いくつかの状態を遷移していくことで，構文解析が正しく進んでいることが分かる．

```
state 0, reading 336 (USING)
state 0, shifting to state 3
state 3, reading 431 (IDENTIFIER)
state 3, shifting to state 30
state 30, reducing by rule 326
    (opt_type_argument_list :)
after reduction, shifting from state 30 to state 72
state 72, reducing by rule 325
    (member_name : IDENTIFIER opt_type_argument_list)
after reduction, shifting from state 3 to state 33
state 33, reducing by rule 322
    (namespace_or_type_name : member_name)
after reduction, shifting from state 3 to state 31
state 31, reducing by rule 28
    (namespace_name : namespace_or_type_name)
after reduction, shifting from state 3 to state 32
state 32, reading 358 (SEMICOLON)
state 32, shifting to state 74
state 74, reducing by rule 21
    (using_namespace_directive : USING namespace_name
                                                SEMICOLON)
```

図 1 Yacc のデバッグ情報の一部
Fig. 1 A fragment of debug information generated by Yacc.

本稿で提案する 2 段階パージングでは，パーザの内部動作を XML 文書で表現して記録する．記録した XML 文書は，次のステップで読み込まれ，その内容に従って，前のステップの構文解析を再現するために使われる．

2.2 パーザにおける還元–還元競合

生成されたパーザがトークンを読み込むとき，複数の文法規則に適合して複数の文法規則を還元する可能性があれば，還元–還元競合（reduce/reduce conflict）があるという[19]．還元–還元競合の例として，プログラミング言語 C#☆☆のための文法規則の一部を**図 2** に示す．この規則の中では，大文字の単語（NEW，PUBLIC など）はトークンを示し，小文字の単語（class_modifier，interface_modifier など）は非終端記号を示す．パー

☆ 開発に参加している各社との契約上，製品名は記さない．

☆☆ 以下，単に C# とだけ書いたときは，プログラミング言語 C# のことを指すものとする．

```
class_declaration
   :opt_class_modifiers opt_partial
    CLASS ID opt_type_parameter_list ...
   ;
opt_class_modifiers
   :
   |class_modifiers
   ;
class_modifiers
   :class_modifier
   |class_modifiers class_modifier
   ;
class_modifier
   :NEW | PUBLIC | PROTECTED | INTERNAL
   |PRIVATE | ABSTRACT | SEALED | STATIC
   ;
interface_declaration
   :opt_interface_modifiers opt_partial
    INTERFACE ID opt_type_parameter_list ...
   ;
opt_interface_modifiers
   :
   |interface_modifiers
   ;
interface_modifiers
   :interface_modifier
   |interface_modifiers interface_modifier
   ;
interface_modifier
   :NEW | PUBLIC | PROTECTED | INTERNAL
   |PRIVATE
   ;
```

図 2 プログラミング言語 C# の文法規則の一部
Fig. 2 A fragment of grammar rules for the C# programming language.

ザがトークン PUBLIC を読み込むとき，2 つの規則（class_modifier と interface_modifier）に適合し，その後両方の規則を同時に還元しようとするので，還元–還元競合となる．還元–還元競合のある文法規則から Yacc が生成するパーザは，文法規則の定義で先に書かれた規則，すなわち，class_modifier の規則をつねに選択し，interface_modifier の規則を選択することはけっしてない．したがって，還元–還元競合は可能な限り取り除くべきである．

還元–還元競合のない文法規則を書くことは非常に難しい．C# は，Ecma インターナショナルによって標準化されている[20]．筆者が C# の文法規則を記述した経験と論文 21) によれば，Ecma による C# の仕様に従って文法規則を素直にそのまま書くと，800 以上の文法規則が必要となり，しかも，600 以上の還元–還元競合を持つ定義ができあがる．C#プログラムを正しく解析するパーザを開発するには，この 600 以上の還元–還元競合をすべて除去する必要がある．

還元–還元競合の除去に関するこれまでの経験では，

- 共通部分を括り，または，特定の文法規則を他の文法規則の中に展開するなど文法規則を書き換える，
- 字句解析でフラグを設定する，または，意味的な情報を利用することで，字句解析から構文解析へ渡すトークンの種類を文脈に合わせて変更する，

などの方法がある．しかし，これらの方法は還元–還元競合の除去作業を進めるための経験則でしかなく，方法論として定まっているわけではない．また上記の方法で還元–還元競合をすべて除去しきれないこともあり，その場の思いつきで解決する場合もある．論文 21) では，C# のためのパーザの開発を始めてから第 19 版までの修正作業により，約 600 カ所あった還元–還元競合を 1 カ所まで減らすことができたと述べられている．しかし，この作業に膨大な時間を費やしたことは容易に推測できる．

還元–還元競合の除去のための修正作業を進めるうえで，すべての文法規則の正しさを確認するために，多くのテストケースを作成してテストを実行しないと，隠れたエラーを発見することができない．また，C# のような新しいプログラミング言語の場合は，その言語仕様がいまだに変化し続けているため，パーザの保守にもかなりの時間を必要とする．

このように，正しい文法規則を完成させるのは容易な作業ではない．本稿で提案する 2 段階パージングは，新しく文法規則を書くことなく，なるべく簡単にパーザを開発するための手法である．

2.3 Mono と C#

Mono は，Novell によって商業的に支援されている FOSS であり，プラットフォーム独立な.NET 実行環境と開発ツールを提供する[24]．Mono の主たる目的は，Linux 開発者が.NET アプリケーションを開発し，配備できるようにすることである．Boston で開催された Linux World 2006 でベストアプリケーション開発プラットフォーム（Best Application Development Platform）賞を受賞[25] したことからも分かるように，Mono プロジェクトでは優れた製品が開発されている．

Mono は，主要なオペレーティングシステム（Linux, Mac OS X，Windows など）で動作し，その配布物には 2 種類の C#コンパイラと Visual Basic コンパイラ，そして，CLI 仮想マシン（クラスローダ，ジャストインタイムコンパイラ，ガーベジコレクションなどが実装済み）が含まれる．

本稿では，Mono バージョン 1.1.13.2 として公開されている配布物を使うことを前提として議論を進める．Mono バージョン 1.1.13.2 のソースコードを調べ

ると，配布物全体は 200 万行からなり，その一部である Mono C#コンパイラは 6 万行の C#プログラムで記述されている．

逆エンジニアリングツールを開発する立場としては，C# の XML ドキュメンテーションコメントが興味深い[26]．C# の XML ドキュメンテーションコメントは，/// で始まるコメント行に，XML を使って記述して，クラスやメソッドに関する情報を埋め込むためのものである．Java における javadoc の利用を考えたドキュメンテーションコメントに相当する機能である．たとえば，**図 3** に示すように，C#プログラム中に XML ドキュメンテーションコメントを書くことができる．この C#プログラムを XML ドキュメンテーションファイルの出力指定をしてコンパイルすると，**図 4** に示す XML 文書が出力される．このような C# の XML ドキュメンテーションコメントのために，約 20 種類のタグが準備されている．準備され意味が決まっているタグ以外のタグはそのまま出力されるため，出力後の XML 文書を変形して利用するなど，数多くの応用例がありうる．逆エンジニアリングツールを開発する場合には，この XML ドキュメンテーションコメントを無視することなく，重要な情報として取り出せるようにするべきである．

```
using System;
public class Hello
{
    /// <summary>Entry point</summary>
    /// <param name="args">
    ///    Command line arguments </param>
    /// <returns>void</returns>
    public static void Main(string[] args){
      Console.WriteLine("Hello");
    }
}
```

図 3　XML ドキュメンテーションコメントの例
Fig. 3　Examples of a set of XML documentation comments.

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>Hello</name>
  </assembly>
  <members>
    <member name="M:Hello.Main(System.String[])">
      <summary>Entry point</summary>
      <param name="args"> Command line arguments </param>
      <returns>void</returns></member>
  </members>
</doc>
```

図 4　生成された XML ドキュメンテーションファイルの例
Fig. 4　An example of a generated XML documentation file.

2.4　パーザの開発

逆エンジニアリングツールの基本機能の 1 つに，ソースコードからクラス図を生成する機能がある．たとえば，筆者が開発した C# で書かれたソースコードのための逆エンジニアリングツールを使うと，**図 5** に示すクラス図を生成できる．このクラス図は，Mono C#コンパイラのソースコードから生成したクラス図の一部であり，長方形はクラスを意味している☆．

逆エンジニアリングツールを開発した経験から，ソースコードを解析するためのパーザを開発するには，以下の 3 つの手法が考えられる．

（1）　プログラミング言語の仕様書を読んでゼロからパーザを開発する．
C# の仕様書[20] に従ってパーザ開発を進める場合，前に述べたように約 800 以上の文法規則が必要となり，還元–還元競合をすべて除去した文法規則を完成させるには，かなりの期間が必要となる．アジャイル開発が一般に広がり短期間の開発が当然となってきている状況を考えると，パーザ開発だけで時間を使いすぎるべきではない．

（2）　パーザ開発の情報が公開されている Web サイトから文法規則をダウンロードして利用する．
何種類ものプログラミング言語の文法規則を公開している Web サイトがあり[22]，その文法規則を使えば，

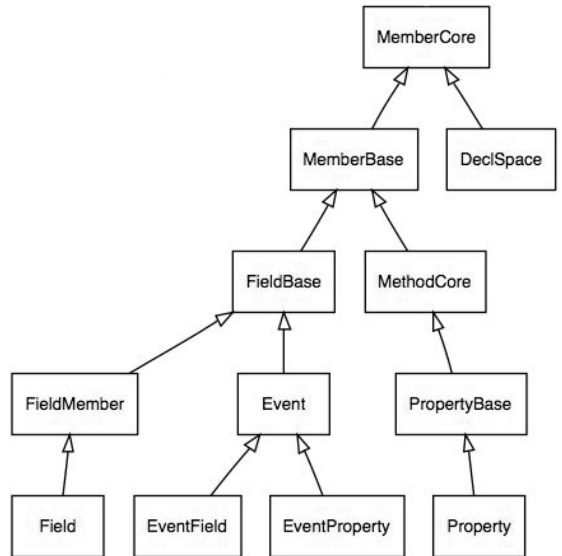

図 5　Mono C#コンパイラから生成されたクラス図
Fig. 5　A class diagram generated from the source code of the Mono C# compiler.

☆ 簡潔に表記するために，図 5 ではメソッドとフィールドを意図的に省略した．

パーザ開発の手間をかなり省くことができる．ところが，その Web サイトにある多くの文法規則は，厳格に正しいことが保証されているわけではなく，ある程度のエラーが含まれる可能性がある．その結果，文法規則の正しさを確かめ，エラーがあれば修正するために多くの作業時間が必要となる．さらには，プログラミング言語として人気のある C# や Java は，時が経過するにつれて言語仕様が激しく変化していくため，そのときどきの言語仕様に準拠するようにパーザを保守していくことも手間のかかる作業となる．

（3） FOSS コンパイラからパーザのソースコードを抜き出す．
商用の製品をコンパイルするときに使われるほど高品質の FOSS コンパイラがいくつも公開されている（たとえば，GNU コンパイラコレクション[23] や Mono C#コンパイラ[24] など）．これらのコンパイラの開発ではパーザ生成系が使われているものの，ソースコードを入力しオブジェクトコードを出力することだけを考えて開発されていて，そのコンパイラ内部では，パーザが他のモジュール（たとえば，字句解析，記号表操作，意味解析）と密に結合している．したがって，一般的には，コンパイル以外の目的のために正しく動作する文法規則だけを抜き出すことはできない．

本稿では，C#パーザの開発を対象として，上記の 3 つとは違った新しい手法 2 段階パージングを提案する．この手法では，新しく考案したパーザ生成系 MJay を使い，Mono C#コンパイラで使われている Jay を MJay に置き換え，さらに Mono C#コンパイラに若干の修正を加えることで，Mono C#コンパイラの一部をパーザとして利用する．3 章では，2 段階パージングの詳細について述べる．

3. 2 段階パージングを使ったパーザの開発

3.1 提案する手法の概要

Mono C#コンパイラの概念的な構造を**図 6** に示す．Mono C#コンパイラは，C#ソースコードを読み込み，解析と合成の処理の後，.NET 用の中間言語 CIL（Common Intermediate Language）[27] によるオブジェクトコードを生成する．コンパイラ内部は主ドライバ，字句解析，構文解析，コード生成などのクラスからなる．構文解析を担当するパーザは，パーザ生成系 Jay を使って作られている．Mono C#コンパイラでの Jay の役割は，Yacc 互換の記法で書かれた C# の文法規則を読み込み，C#プログラムを解析する，C# で書かれたパーザを生成することである．

本稿で提案する 2 段階パージングでは，逆エンジニアリングを行いたいソースコードを構文解析するためのパーザを 2 つのステップに分割する．**図 7** に，分割された 2 つのステップとそれらの入出力を示す．

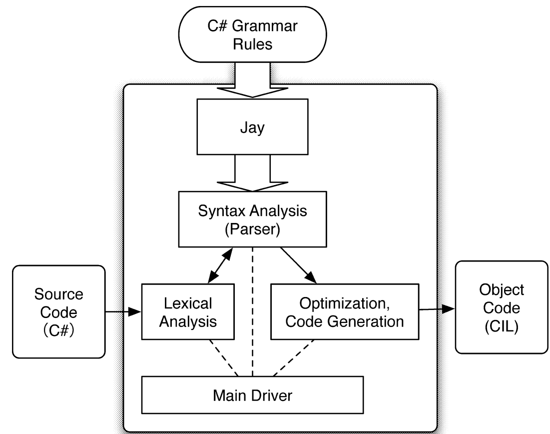

図 6 Mono C#コンパイラの概念的な構造
Fig. 6 Conceptual structure of the Mono C# compiler.

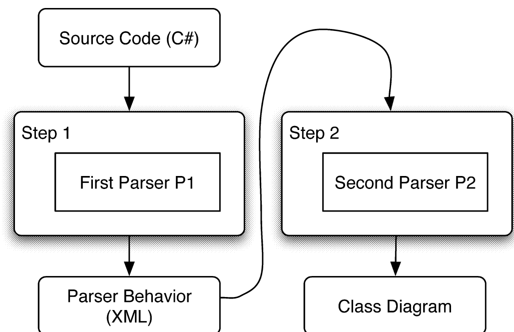

図 7 新しい方式の逆エンジニアリングのための 2 つのパーザ
Fig. 7 Two parsers for reverse engineering using the new method.

ステップ 1 の First Parser P1（以下，第 1 パーザと記す）は，新たに考案したパーザ生成系 MJay を使って開発する．第 1 パーザを含むステップ 1 では，ソースコードを読み込んで解析し，XML で記述された Parser Behavior（以下，パーザ動作と記す）を生成する．逆エンジニアリングツール開発の場合を例として考えると，ステップ 1 がパーザ動作を生成した後，ステップ 2 がパーザ動作を読み込んで解析し，構文解析を終える．その後で，図を描画するためのプログラムを動かせば，クラス図やコミュニケーション図やシーケンス図ができあがる．

ステップ 2 では，Mono C#コンパイラのソースコードから自動生成された Second Parser P2（以下，第 2 パーザと記す）を使う．図 7 に示すように，ステップ 1 で記録された XML 文書を読み込み，この中から

```
using System;
namespace Com.Xyz
{
   /// <summary>Print "Hello"</summary>
   public class Hello : World
   {
       public static void Main(string[] args){
         Console.WriteLine("Hello");
       }
   }
}
```

図 8 C#プログラムの例
Fig. 8 An example program in C#.

トークンとパーザの各動作を読み込みながら，ステップ 1 で実行した構文解析とまったく同じ動きを再現する．

3.2 ステップ 1 について

ステップ 1 では，オープンソースコンパイラのソースコードを修正して利用するが，パーザの処理が終了したところでコンパイラの処理を止めるようにする．そのためには，パーザを呼び出すメソッド yyparse() の処理が終わったところで，構文解析が正常に終了したかどうかを判断した後，コンパイラの前段階の部分を終了させる．

2.4 節にも述べたとおり，コンパイラのソースコードからパーザだけを正確に抜き出すことは非常に難しい．そこで，ステップ 1 では，ステップ 2 を動かすために必要な情報をパーザ動作として記録する機能を Mono C#コンパイラに埋め込む．これまでの経験では，ステップ 1 を実装するために，Mono C#コンパイラに施す修正作業は数時間あれば終了する．

第 1 パーザが生成するパーザ動作は，LALR パーザの基本的なアクション，すなわち，シフト，還元，トークン読み込みなどからなる．たとえば，**図 8** にある C#プログラムを考えてみる．第 1 パーザは，このC#ソースコードを読み込み，構文解析を行いながら，付録にあるような XML で書かれたパーザ動作を生成する．**表 1** に要素の名前と意味，**表 2** に属性の名前と意味，さらには，パーザ動作の DTD による定義を**図 9** に示す．なお，付録に記したパーザ動作の中で，C#プログラム中の class の前にある XML ドキュメンテーションコメントが，xdc 要素として後の方に位置しているのは，ステップ 2 を効率良く実装するための実装上の判断のためである．

ステップ 1 の概要を**図 10** に示す．この図で示されているように，ステップ 1 では，Mono C#コンパイラで使われているパーザ生成系 Jay を，新たに考案した MJay に置き換える．また，入力となるソースコード内で漢字コード（シフト JIS，UTF8，EUC）を正しく解析できるように，漢字コードの自動判定と変換のためのクラスを追加している．

表 1 パーザ動作で使われる要素
Table 1 Elements in the parser behavior.

名前	要素の意味
parseFiles	ルート要素
parse	1 つのソースファイルの情報を表現
lex	トークンの読み込み
shi	シフト
red	還元
xdc	XML ドキュメンテーションコメント
accept	すべてのパーザ動作を終了

表 2 パーザ動作で使われる属性
Table 2 Attributes in the parser behavior.

名前	属性の意味
st	状態を識別する番号
fr	シフトする前の状態の識別番号
to	シフトした後の状態の識別番号
tk	トークンの種類
va	トークンの文字列イメージ
li	ソースファイル中の行番号
co	ソースファイル中の列番号
ru	文法規則を識別する番号

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT parserFiles (parse*,acc)>

<!ELEMENT parse (lex|shi|red|xdc)*>
<!ATTLIST parse name CDATA #REQUIRED>

<!ELEMENT lex EMPTY>
<!ATTLIST lex st CDATA #REQUIRED>
<!ATTLIST lex tk CDATA #REQUIRED>
<!ATTLIST lex va CDATA #REQUIRED>
<!ATTLIST lex li CDATA #REQUIRED>
<!ATTLIST lex co CDATA #REQUIRED>

<!ELEMENT shi EMPTY>
<!ATTLIST shi fr CDATA #REQUIRED>
<!ATTLIST shi to CDATA #REQUIRED>

<!ELEMENT red EMPTY>
<!ATTLIST red st CDATA #REQUIRED>
<!ATTLIST red ru CDATA #REQUIRED>
<!ATTLIST red ln CDATA #REQUIRED>

<!ELEMENT xdc (#PCDATA)>

<!ELEMENT acc EMPTY>
```

図 9 パーザ動作の DTD による定義
Fig. 9 DTD for the parser behavior.

3.3 ステップ 2 について

MJay は，Jay が提供するパーザ生成の機能をそのまま持ち合わせたうえで，ステップ 1 の図 10 に示す

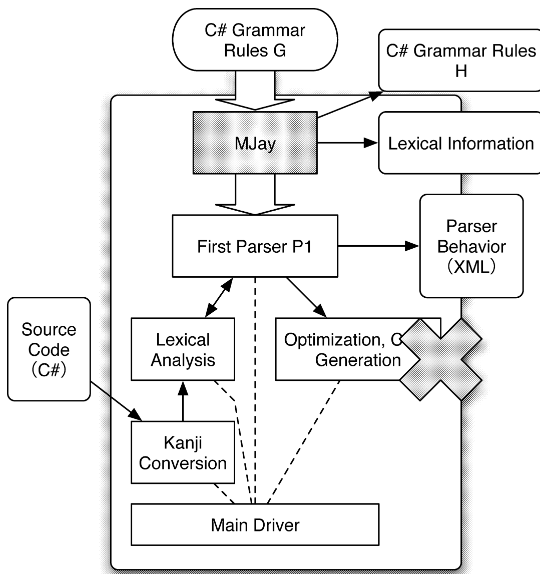

図 10 ステップ 1：Mono C#コンパイラへの機能変更と追加
Fig. 10 Step 1: Functional changes and additions to the Mono C# compiler.

```
while_statement
  : WHILE OPEN_PARENS boolean_expression
      CLOSE_PARENS embedded_statement
  {
     Location l = (Location) $1;
     $$ = new While((Expression)$3, (Statement)$5,l);
  }
  ;
```

図 11 第 1 パーザのための文法規則の一部
Fig. 11 A fragment of a grammar rule for the first parser.

```
while_statement
  : WHILE OPEN_PARENS boolean_expression
        CLOSE_PARENS embedded_statement REDUCE_656
  ;
```

図 12 MJay が生成する文法規則の一部
Fig. 12 A fragment of a generated grammar rule by MJay.

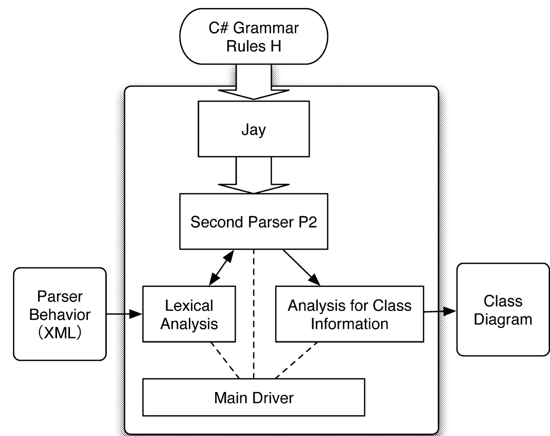

図 13 ステップ 2：第 2 パーザを含む逆エンジニアリングツール
Fig. 13 Step 2: Reverse engineering tool including the second parser.

ように，文法規則 G から文法規則に付随するアクションや余分なコードをすべて削除し，さらに特別な記号を追加して，第 2 パーザのための文法規則 H を生成する．たとえば，**図 11** に示す文法規則 while_statement を考えてみる．この例の場合，MJay は中括弧で囲まれたアクションを削除し，文法規則 while_statement を還元するための特別な記号 REDUCE_656 を追加している．

MJay は，還元–還元競合がない文法規則を生成するために，文法規則を一意に識別するための識別番号（**図 12** の例では 656）を含んだ特別な記号を文法規則に埋め込む．ステップ 1 では，解析の対象となる C#ソースコードを読み込みながら，どの文法規則をどのタイミングで還元したかをパーザ動作として文法規則の識別番号とともに記録する．ステップ 2 のために MJay が生成する文法規則には，還元のタイミングをパーザに知らせるための特別な記号（図 12 の例では REDUCE_656）が，必要とするすべての文法規則の最後に埋め込まれ，その結果，特別な記号を含むすべての文法規則の最後が異なることになる．

ステップ 2 では，パーザ動作を読み込みながら解析を再現するので，すべての還元のタイミングと還元すべき文法規則を知ったうえで解析を進めることができる．ステップ 2 の字句解析部が，XML で表現されたパーザ動作の中で還元を表す red 要素を読み込んだときに，解析対象のソースコードにはない仮想的なトークンがパーザに渡される．たとえば，ru 属性が 656 である red 要素を読み込んだ場合，トークン REDUCE_656 が字句解析部で作成されパーザに渡される．このトークンが適合すべき文法規則を一意に決めることになり，その結果，複数の文法規則を同時に還元する可能性を排除することができる．したがって，MJay が生成する文法規則には還元–還元競合がない．

ステップ 2 では，**図 13** に示すように，元来の Mono C#コンパイラで使われていたパーザ生成系 Jay が，MJay によって生成された文法規則 H を読み込み，第 2 パーザを生成する．そのため，生成された第 2 パーザの機能は構文解析のみであり，抽象構文木を生成する機能は持たない．もし，抽象構文木の生成などの操作が必要な場合は，Yacc を使った場合の開発と同じように，文法規則 H にアクションを埋め込むことに

なる．

ステップ2では，通常のYaccとLexを使った開発とほぼ同じで，yyparse()で構文解析を呼び出し，その中でトークンを取り出す字句解析のためのメソッドが使われる．違うところは，字句解析部の入力がソースコードではなく，パーザ動作を記録したXML文書であることである．2段階パージングの実装のために，ステップ2の字句解析のためのサンプルプログラムが準備されている．もし，トークンに付随する情報（たとえば，ソースファイル中の行数やカラム数など）が必要であれば，字句解析のサンプルプログラムを修正し，YaccとLexで使われるyylvalと同じ方法で情報の受け渡しができる．また，字句解析部で求めた情報を別の方法でパーザに渡したい場合は，グローバルデータを管理するクラスを作り，そのクラスのフィールドを使うことで情報の受け渡しが可能になる．

4. 評　　価

4.1 実　　験

2段階パージングの性能を調べるために，以下の要領で実験を行った．

- Paint.NET version 2.64 [28] のソースコードから142個のC#ソースファイルを選んだ．Paint.NETのソースコードの中には7,052行のファイルがあり，これは他のファイルとサイズが違いすぎるため（半分以上のファイルが200行以下），実験結果を検討しにくくなると判断し，他のソースファイルと別扱いにした．
- 2段階パージングを使って開発したC#パーザ（以下，MOCSと記す）の第2パーザでは構文解析のみを行い，抽象構文木を作らない．
- 各ソースファイルの構文解析を10回連続して実行する．その後，実行時間の平均値を求め，また，ステップ1終了後に生成されるXMLファイルの大きさを調べる．
- 比較検討のために，通常のMono C#コンパイラから最適化やコード生成などの後段階の処理を除いたプログラム（以下，GMCSと記す）を，同じように10回連続して実行して，その実行時間の平均値を求める．

C#で書かれたプログラムの実行時間を測定するには，System.DateTime構造体のメンバTicksの値を使う．具体的には，**図14**に示すように，主要な処理の実行前と実行後で，Ticksの実行結果の差を求めて実行時間とする．なお，この実験では，以下のハードウェアとソフトウェアを使った．

```
DateTime datetime = DateTime.Now;
long timeBefore = datetime.Ticks;
.....
datetime = DateTime.Now;
long timeAfter = datetime.Ticks;
long t = timeAfter - timeBefore;
```

図14 実行時間の測定方法
Fig. 14 A sample program to measure execution time.

- 本体：Apple PowerBook G4
- CPU：PowerPC G4 1.5 G Hz
- 主記憶：1 Gバイト（DDR333 PC2700）
- ハードディスク：80 Gバイト（Toshiba MK8025GAS）
- ソフトウェア：Mac OS X 10.4.7，Mono 1.1.13.2

4.2 実験結果と検討

Paint.NETのソースコード行数の分布を，**図15**のヒストグラムで示す．このヒストグラムは，200行間隔で集計したものである．この図から，ソースファイルのほとんど（具体的には63%）が200行以下であることが分かる．

ステップ1の終了後に出力されるXML文書の大きさの分布を**図16**に示す．たとえば，約2,300行のC#ソースファイルを解析すると，約2,200 KバイトのXML文書が生成されている．別扱いとした7,052行のソースファイルを解析すると約7 MバイトのXML文書が生成された．

パーザ動作として生成されるXML文書の大きさは，通常のXML文書のサイズを超えているように思える．しかし，生成されるXML文書は作業用の一時的なものであり，ステップ2の終了後に削除されることと，近年ハードディスクドライブの容量あたりの単価が年々下がっている状況を考えると重大な問題とはならないであろう．

ステップ1とステップ2が終了するまでの平均実行時間の分布を**図17**に示す．ここでは，MOCSの平均実行時間を四角で示し，また，比較検討のためにGMCSを実行させたときの平均実行時間を三角で示す．MOCSを使って最大行数（約2,300行）のソースファイルを解析するのに約17秒かかった．一方，GMCSの場合，同じ最大行数のソースファイルを解析するのに約0.7秒かかった．別扱いとした7,052行のソースファイルをMOCSで解析すると約52秒かかったのに対して，GMCSで解析すると約0.9秒かかった．

実験の結果から，MOCSの実行時間は，通常のコンパイル時間を大幅に超えているように思える．ソフトウェアを開発するとき，コンパイラは頻繁に利用さ

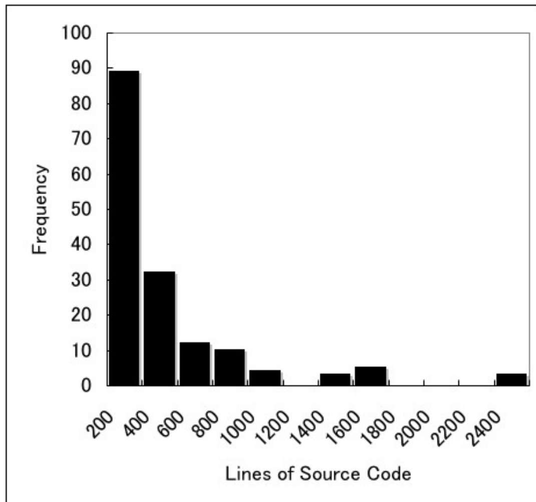

図 15　Paint.NET のソースコード行数の分布
Fig. 15　Frequency of lines of source code included in Paint.NET.

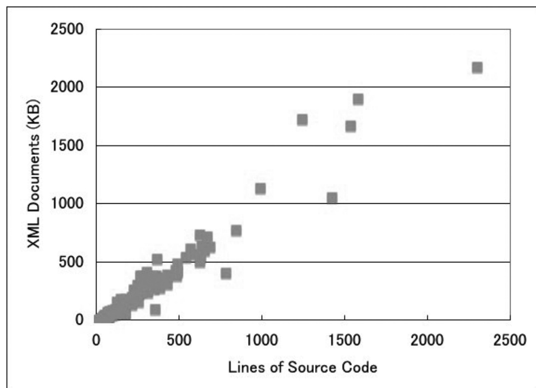

図 16　パーザ動作のために生成された XML 文書の大きさの分布
Fig. 16　Frequency of size of generated XML documents for the parser behavior.

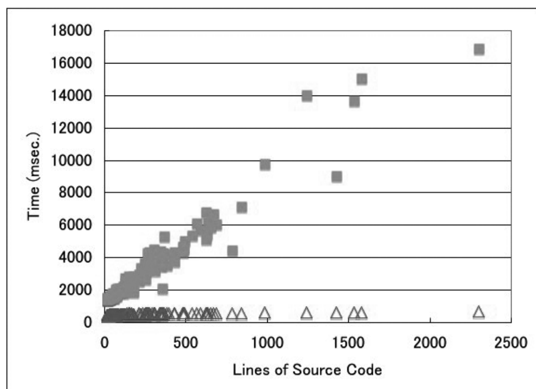

図 17　C#ソースコードの解析のための平均実行時間の分布
Fig. 17　Frequency of average execution time to analyze C# source files.

れる．しかし，逆エンジニアリングツールの場合，その目的が文書の生成であるため，1 日の間に頻繁に使われることはなく，1 週間または 1 カ月に 1 回の頻度で十分だと思われる．したがって，2 段階パージングを使ったことで，実験結果のような時間がかかったとしても，さほど問題とはならないであろう．

4.3　2 段階パージングの完成度

1 章でも述べたが，筆者が開発した 2 段階パージングを使ったパーザを含む逆エンジニアリングツールは，2006 年春から市販製品の一部として組み込まれて出荷されている．出荷後約半年の間，いくつかのバグ報告を受けた．しかし，報告を受けたバグは，2 段階パージングの実装に関するものではない．これは，2 段階パージングが安定して動作し，市販製品として耐えられることを示している．

Mono の配布物には，C#コンパイラを C# の言語仕様に従ってテストするために，約 900 のテストプログラムが含まれている．このテストプログラムを実行した結果，2 段階パージングを使ったパーザを含む逆エンジニアリングツールは，Mono C#コンパイラと変わらないテスト結果を得ることができた．

以上のことから，2 段階パージングで作成した逆エンジニアリングツールの完成度は高く，ほぼ安定して動くと考えてよい．

5. 考　　察

既存のコンパイラのソースコードを修正して利用した例として，JavaML [7] コンバータがある．このコンバータの実装では，Java の FOSS コンパイラとして有名な Jikes [8] が使われている．今回，Jikes version 1.12 をもとにした JavaML コンバータのソースコードを調べてみた．JavaML コンバータを実装するために，約 2,800 行（パッチのための差分ファイルの行数）の修正を Jikes に施している．Jikes の修正作業は，その内部に精通しなければ到底無理なことである．2 段階パージングを使った逆エンジニアリングツールのためのパーザの実装では，漢字コードの自動判定・変換のプログラムを含めて約 1,000 行（パッチのための差分ファイルの行数）の修正を Mono C#コンパイラに施した．主な修正事項は，

- パーザ生成系の MJay への置き換え
- XML 文書出力のための準備
- yyparse() の返却値を確認した後すぐに第 1 ステップを終了させること

の 3 点である．これらの修正作業は，コンパイラ内部にそれほど精通する必要はなく，コンパイラの主なク

ラスのソースコードを少しだけ読解すれば可能であろう．以上のことから，JavaML コンバータに比べて，2 段階パージングの方が実装が容易であることが分かる．

JavaML は，Java ソースコードを構文解析した後，抽象構文木を XML で出力するためのものと考えてよい．したがって，Java 言語に依存した部分がかなり存在する．もし，他のプログラミング言語に適用するならば，JavaML を見直し定義を修正する必要がある．2 段階パージングでは MJay を使うことで，FOSS コンパイラから第 2 パーザのための文法規則が自動生成される．他のプログラミング言語のための FOSS コンパイラがあり，MJay をそのまま利用できれば，どこも修正する必要なく第 2 パーザのための文法規則を取り出し，すぐにでも第 2 パーザを実行可能にできる．

JavaML コンバータにより生成された XML 文書の中から必要な情報を取り出すには，XML を読み込んだ後，DOM などの API を使って木構造を巡回する必要がある．または，XSLT を活用して，生成された XML 文書から必要な情報を取り出さなければならない．JavaML では，解析するソースコードに関してすべての情報が表現されるため，必要な情報をもれなく取り出すことは手間がかかる作業となる．これに対して 2 段階パージングでは，第 2 パーザのために生成された文法規則があるので，文法規則にアクションを埋め込めば，第 2 パーザの構文解析の途中で必要な情報を取り出すことができる．Yacc を使ったパーザ開発に慣れている開発者であれば，生成済みの文法規則にアクションを埋め込む作業は，それほど手間がかかるとは思えない．

2 段階パージングを使って開発したパーザの品質は，元となる FOSS コンパイラに依存する．本稿では Mono C#コンパイラを取り上げたが，このコンパイラは最初にリリースされてから約 5 年間の実績があり，商用のソフトウェアを開発するために使われるほど高品質である．しかも，開発者たちの不断の努力により，パーザにおける還元–還元競合はすべて除去されている．また，かなりの数のバグが除去されたことがメモとして残っていて，その結果，パーザの動作は信頼性が高い．このように信頼性が高いコンパイラから第 2 パーザを生成するため，第 2 パーザに関しては，テストを行う必要がほとんどないといってよい．

C# はいまだに仕様の変化が激しいプログラミング言語である．もし，Yacc だけを使って，現版の仕様を満たすパーザの開発に成功しても，仕様が改訂され新しい言語機能が盛り込まれると，文法規則を再度見直し修正・拡張する必要がある．2 段階パージングは，すでに使いこまれ，バグが枯れた状態の FOSS コンパイラを利用することを前提にしている．もし，どこかの開発者が最新の言語仕様を満足するように Mono C#コンパイラを修正し，その後世界中の開発者たちに使い込まれ，バグが枯れた状態になるのであれば，その成果をそのまま譲り受けることができる．もし，最新の言語仕様に対応するように，だれも書き換えてくれない場合は，自分で Mono C#コンパイラを修正する必要がある．この作業には，コンパイラの内部を知る必要があり，かなりの手間がかかる．

筆者は，Visual Basic 用の逆エンジニアリングツールの開発で，2 段階パージングの限界を経験した．Mono の配布物に含まれる Visual Basic コンパイラは，Visual Basic .NET 2003 の言語仕様にしか対応していない．ところが，逆エンジニアリングツールとして最新版の Visual Basic .NET 2005 の言語仕様によるソースコードを解析する必要があった．そこで，Mono Visual Basic コンパイラのパーザを改造するために文法規則を拡張し，コンパイラ内部を修正することで，Visual Basic .NET 2005 のプログラムの構文解析が可能なコンパイラを作った．ところが，筆者の手元においてのみテストを行ったため，テストが不十分となり，Visual Basic 用の逆エンジニアリングツールに関していくつかのバグ報告を受けることになった．C# 用の逆エンジニアリングツールに関しては，Mono C#コンパイラの品質が良かったため，いまだバグ報告を受けていない．

2 段階パージングには，もう 1 つ問題がある．第 2 パーザの文法規則は，FOSS コンパイラ内の文法規則をもとにして生成される．もし，2 段階パージングで開発したパーザがすでに存在し，その後 FOSS コンパイラの構文規則が修正された場合，MJay によって第 2 パーザのための文法規則が新しく生成され，既存の文法規則を新しいものと置き換える必要が出てくる．その結果，既存の文法規則にアクションが埋め込まれていれば，そのアクションを新しい文法規則に手作業で移す面倒な作業が必要となる．解決案としては，文法規則とアクションを分離して管理し，必要なときにマージする方法を考えればよいのだが，現状では今後の課題となっている．

逆エンジニアリングツールを開発する全体から見ると，入力となるソースコードを解析する部分は些細なことなのかもしれない．開発の手間を省くために，既存のツールを活用する方法もありうる．JavaML コンバータの場合は，FOSS コンパイラである Jikes を利用できたため，実装が簡単であった．本稿で提案する

2 段階パージングは，FOSS コンパイラのソースコードに関して，JavaML コンバータの実装とは違った利用方法を提供する．しかも，JavaML コンバータに比べて実装が容易である．このことから，2 段階パージングは，パーザを迅速に開発するための選択肢を増やすことになり，逆エンジニアリングツールの開発者に寄与できると考えている．

情報システム開発において，既存のソースコードを有効に再利用するときには逆エンジニアリングツールが不可欠である．2 段階パージングを使えば，逆エンジニアリングツールでソースコードを解析するためのパーザを迅速に開発できる．これにより，新しいプログラミング機能を活用するための逆エンジニアリングツールを素早く提供でき，その結果，情報システムの開発を促進することになる．

6. おわりに

本稿では，FOSS コンパイラのソースコードから迅速にパーザを開発するための手法「2 段階パージング」の動機とアイディア，さらには，その実装のために新たに考案した MJay について述べた．2 段階パージングは，パーザを 2 つに分割することで，従来のパーザ開発に比べて，短時間でパーザを開発できる特徴を持つ．

2 段階パージングにおける第 1 パーザは，新たに考案したパーザ生成系 MJay によって自動生成され，パーザ動作を記録する機能が埋め込まれる．MJay は，第 1 パーザとは別に第 2 パーザのための文法規則を生成する．生成された文法規則は，通常のパーザ生成系 Jay が読み込んで第 2 パーザを生成する．その第 2 パーザは，第 1 パーザで記録されたパーザ動作を読み込みながら解析処理を行う．

2 段階パージングの性能を調べるために実験を行った．その実験結果から，パーザ動作として生成される XML 文書が非常に大きくなることが分かった．また，通常のコンパイル時間を大幅に超える実行時間がかかることも分かった．しかし，最近のハードウェアの進化と逆エンジニアリングツールの利用頻度の特徴から，これらのことは重大な問題にはならないと思われる．

現在，MJay を多機能なパーザ生成系とするための再開発を進めている．オープンソースが多方面にわたって急速に広がってきている．その広がりを持つオープンソースへ日本から貢献するために，MJay を世界に向けて公開するとともに，本稿で述べた手法をさらに進化させ，他の情報システム開発に対して適用するための研究を進めていきたいと考えている．

謝辞 本研究を進めるにあたり数多くの有益なご意見をいただいた竹下亨先生と原田賢一先生に感謝の意を表する．また，貴重なコメントをいただいた査読者の方々，市販製品を出荷するまで苦労をともにした開発者たちに感謝の意を表する．本研究の一部は，文部科学省のオープンリサーチセンター整備事業の支援を受けた．

参 考 文 献

1) 日本 BEA システムズ：SOA サービス指向アーキテクチャ，翔泳社 (2005).
2) 保阪武男，早津俊秀：ビジネスは SOA で変革する。，翔泳社 (2005).
3) 竹下　亨：ソフトウェアの保守・再開発と再利用，共立出版 (1992).
4) Poseidon for UML. http://www.gentleware.com/
5) MagicDraw. http://www.magicdraw.com/
6) Visual Paradigm for UML. http://www.visual-paradigm.com/product/
7) Johnson, S.C.: JavaML: A Markup Language for Java Source Code, *9th International World Wide Web Conference* (2000).
8) Jikes' Home. http://jikes.sourceforge.net/
9) The Linux Kernel Archives. http://www.kernel.org/
10) The Apache Software Foundation. http://www.apache.org/
11) PostgreSQL. http://www.postgresql.org/
12) OpenOffice.org. http://www.openoffice.org/
13) Eclipse.org. http://www.eclipse.org/
14) 情報処理振興事業協会：オープンソースソフトウェアのセキュリティ確保に関する調査報告書 (2003).
15) 経済産業省：オープンソースソフトウエアの利用状況調査/導入検討ガイドラインの公表について (2003).
16) Asia Open Source Software Symposium. http://www.asia-oss.org/
17) Chris DiBona, D.C. and Stone, M. (Eds.): *Open Sources 2.0: The Continuing Evolution*, O'Reilly (2006).
18) Johnson, S.C.: Yacc: Yet Another Compiler Compiler, *UNIX Programmer's Manual*, Vol.2, pp.353–387 (1979).
19) A.V. エイホ，R. セシィ，J.D. ウルマン：コンパイラ—原理・技法・ツール，サイエンス社 (1990).
20) Ecma International: *C# Language Specification, Standard ECMA-334* (2006).
21) Brian, A., Malloy, J.F.P. and Waldron, J.T.: Applying Software Engineering Techniques to Parser Design: the Development of a C#

Parser, *SAICSIT '02*, pp.75–82 (2002).

22) Grammar List.
http://www.antlr.org/grammar/list

23) Foundation, F.S.: *GCC Home Page – GNU Project*. http://gcc.gnu.org/

24) Main Page - Mono.
http://www.mono-project.com/Main_Page

25) Winners of Product Excellence Awards Announced at LinuxWorld Conference and Expo in Boston. http://www.linuxworldexpo.com/live/12/media/news/

26) XML ドキュメントコメント（C#）.
http://msdn2.microsoft.com/ja-jp/library/b2s063f7.aspx

27) Ecma International: *Common Language Infrastracture (CLI), Standard ECMA-335* (2006).

28) Paint.NET. http://www.getpaint.net/

付　　録

A.1　パーザ動作

XMLで表現されたパーザ動作の一部を**図18**に示す.

```
<lex st="488" tk="PUBLIC" va="" li="5" co="9" />
<red st="488" ru="33" ln="0" />
<red st="611" ru="31" ln="0" />
<red st="695" ru="49" ln="0" />
<shi fr="16" to="55" />
<red st="55" ru="558" ln="1" />
<red st="64" ru="555" ln="1" />
<lex st="63" tk="CLASS" va="" li="5" co="15" />
<red st="63" ru="554" ln="1" />
<red st="62" ru="551" ln="0" />
<shi fr="104" to="214" />
<red st="214" ru="546" ln="0" />
<lex st="319" tk="IDENTIFIER" va="Hello" li="5" co="21" />
<shi fr="319" to="491" />
<lex st="491" tk="COLON" va="" li="5" co="23" />
<red st="491" ru="326" ln="0" />
<red st="72" ru="325" ln="2" />
<red st="512" ru="547" ln="0" />
<shi fr="627" to="711" />
<lex st="711" tk="IDENTIFIER" va="World" li="5" co="29" />
<shi fr="711" to="75" />
<lex st="75" tk="OPEN_BRACE" va="" li="6" co="4" />
<red st="75" ru="326" ln="0" />
<red st="72" ru="325" ln="2" />
<red st="33" ru="322" ln="1" />
<red st="103" ru="320" ln="0" />
<red st="213" ru="331" ln="2" />
<red st="767" ru="343" ln="1" />
<red st="768" ru="573" ln="2" />
<red st="713" ru="572" ln="1" />
<red st="712" ru="574" ln="0" />
<red st="769" ru="548" ln="0" />
<xdc> &lt;summary&gt;Print "Hello"&lt;/summary&gt;</xdc>
```

図18　XMLで表現されたパーザ動作の一部
Fig. 18　A fragment of parser behavior in XML.

(平成18年 5 月16日受付)
(平成18年12月 7 日採録)

前田　和昭（正会員）
昭和60年慶應義塾大学理工学部管理工学科卒業．平成2年同大学大学院後期博士課程修了．同年中部大学経営情報学部助手．平成15年中部大学経営情報学部助教授．現在に至る．プログラミング言語処理系，オープンソース，XML，オブジェクト指向プログラミング，データベースに興味を持つ．ACM，IEEE Computer Society，ソフトウェア科学会，電子情報通信学会，教育情報システム学会，情報システム学会各会員．