パーザ構築法の活用による
柔軟なデータ処理の実現に関する検討
Flexible Data Processing Based on Experiences of Parser Development

前田和昭†
Kazuaki Maeda†
† 中部大学　経営情報部
† College of Business Administration
and Information Science, Chubu University

要旨
XML で記述されたデータを処理するときには，DOM や SAX などの決まった手法を使うのが通例である．しかし，システム構築時にデータ処理の手法を一度決めると，その手法に固定されてしまい，他の場面で使いづらくなることが少なくない．そこで本稿では，コンパイラにおけるパーザ構築の経験を踏まえ，柔軟なデータ処理を実現することについて検討する．

1. はじめに

階層構造を持つデータを表現するために通常は木構造が使われる．コンパイラでは，ソースコードの構文と意味を表現するための木構造を抽象構文木（AST）[1] と呼んでいる．著者は，ソースコードを解析するために，これまで数多くの構文解析プログラム（以下，パーザと呼ぶ）を作成してきた．最近では，商用システムに組み込むために，Java 言語を対象としたパーザを作成した．Java パーザを作成するにあたり，Java 言語仕様[2]を参考にして構文を定義し，AST のために約 200 個のクラスを作成し，その内部では 1,000 個以上のフィールド・メソッドを宣言している．これらを正常に動作させるには，実装の上で数多くの問題を解決する必要があり，非常に厄介で膨大な時間が必要であった．

インターネットの発達とコンピュータの飛躍的な性能向上を背景に，XML（Extensible Markup Language）を使ってデータを表現することが多くなってきた．例えば，アプリケーション間でデータを交換したり，データベースからデータを取り出して他の製品で活用したり，またネットワークを介してデータ交換する場面など多くの応用分野が広がってきている．コンパイラの解析結果である AST を XML で表現する方法もいくつか提案されている．

XML データの入力とその処理には，DOM や SAX などの決まった手法を使う．XML データの定義からプログラムを生成する手法もよく使われる．一方，入力されたデータを木構造で保持する場合，木構造アクセスを中心とした処理を記述するために，Visitor に代表されるデザインパターンを使うことが多い．しかし，データの入力から木構造アクセスまでを効率よく支援する手法は，筆者が知る限り見当たらない．

本稿では，これまでのコンパイラ開発における経験を踏まえ，２節でデータ処理のための既存の手法を概観し，さらに３節にて柔軟なデータ処理を実現するために検討を進めている手法 REFORM（Ruby Empowering Flexible Object Representation and Manipulation）について述べる．

2. 既存手法の検討

2.1. XML データ処理

パーザの解析結果である AST を XML で表現する研究として，JavaML[3]と XMLizer[4]が提案されている．ソースコードを入力とし，XML で表現された AST を出力するツールを誰かが提供してくれれば，パーザを作成するという厄介な作業を省く事ができる．XML で表現されている AST に対して，XML を操作するライブラリを使った処理を記述すれば，パーザを作成することなくソースコードを解析するツールの出来上がりとなる．

XML で表現された AST を処理するには，DOM や SAX などの手法が有効である[5]．DOM では，XML データを読み込んだ後の木構造アクセスのための API が提供される．SAX では，タグの開始／終了などを読み込んだときにイベント発生させ，そのイベント発生時の処理を記述するための機能が提供される． Java では，DOM API を提供するパッケージ org.w3c.dom，SAX API を提供するパッケージ org.xml.sax があり，標準の開発キットを入手すれば誰でもプログラム作成が可能である．

DOM API を使えば，開発者が木構造を意識しながらデータを操作できる．しかし，DOM API は XML データを操作するために設計され，アプリケーション向けの API とは異なる．アプリケーション向けの API を提供するためには，DOM API を使って木構造を巡回しながらデータ構造を変換する必要がある．SAX を使えば，イベントに対応する処理を記述すれだけで良く，開発者にとっては DOM よりも理解しやすい．しかし，木構造を意識しながら処理を記述するための機能がないため，構造を使う処理を記述するには，状態を保持するためのスタックを自分で管理する必要があり厄介である．

開発者が XML データを意識しないでデータにアクセスできるようにするために，XML のスキーマ定義からクラス群を生成するデータバインディングという手法も広く使われている[6][7]．しかし，XML データを読み込み，木構造データを内部に作り上げる事ができても，木構造にアクセスするには，オブジェクト間の参照関係に沿って処理を進めるための単純な方法しか準備されていない．また，保守を考えると，生成されたクラス群を手作業で修正することは避けるべきなので，デザインパターンを活用した効果的な実装ができない．

2.2. デザインパターンを使った木構造アクセス

木構造を巡回するプログラムを開発したい場合，デザインパターン[8]で述べられている Visitor パターンを使うことが多い．Visitor パターンでは，木構造の各ノード上での処理を，木構造から分離するために，一連の処理を一つのクラスにまとめる．さらには，各ノードを巡回するときに，決められたメソッド（例えば，access メソッド）を呼び出し，そこから Visitor 内部の該当するメソッドを呼び出すようにダブルディスパッチを実装する．Visitor の抽象クラスを定義し， access メソッドの引数として各ノードに渡すように記述することで，ある特定の実装クラスから別の実装クラスへ簡単に切り替えることが可能になる．その結果，木構造に多彩な処理を施すことができる．

木構造の定義から各ノードでの処理を分離して一カ所にまとめるという点において，Visitorパターンは素晴らしい知恵ではあるものの，いくつかの欠点が明らかにされている．例えば，文献[9]では，以下の4つの問題点が示されている．

・混乱問題（Confusion Problem）

通常のプログラミング言語で記述する場合，Visitorパターンを適用した部分がソースコード中に点在することになり，パターン適用部と不適用部の区別がつかない．

・間接呼び出し問題（Indirection Problem）

木構造とVisitorクラスをacceptメソッドからの呼び出しを使って間接的に関連づけるので，理解しづらくなる．

・カプセル化違反問題（Encapsulation Breaching Problem）

木構造から分離したVisitorクラスから，各ノード内のフィールドにアクセスする必要があり，全てのフィールド（またはアクセッサ）を公開する必要がある．これは，クラスを使って各ノードをカプセル化する考えに違反している．

・継承関連問題（Inheritance Related Problem）

Visitor のための抽象クラスを継承する実装クラスでは，他のクラスからの機能を継承する事が難しくなる．

このような問題を解決するために，アスペクト指向プログラミング[10]を活用する手法があり得る．木構造の各ノードに accept メソッドを織り込むことで，Visitor パターンと同等の機能を記述することができ，しかも，Visitor を１つのアスペクトとしてモジュール化することができる．しかし，筆者の

経験では，XML データにアクセスする場面でデータバインディングを使うとき，生成されたクラスのソースコードを見ることはまれで，ほとんどの場合，XML のスキーマ定義を見ながら処理を記述することになる．したがって，データバインディングを使って生成されたソースコードに，アスペクト指向プログラミングを使って処理を織り込むことは最善の方法であるとはいえない．

3. 柔軟なデータ処理のための構想

前節の検討から， データの読み込みから木構造アクセスまでを上手く支援するには，
・SAXのようにイベント中心に扱うことが可能
・DOMのように木構造データを処理することが可能
・データ構造の定義からデータ処理のためのクラス群を生成することが可能
・木構造アクセスのための方法をカスタマイズ可能
であることが必要であり，これらを統合して提供すべきだと考える．このようなデータ処理を実現するための手法をREFORMと呼び，以下でその構想を述べる．

3.1. 要素列の表現

REFORMでは，データの最小単位を要素と呼び，名前と値のペアで表現する．例えば，

```
var "x"
```

は，要素の名前がvarで，要素の値が"x"であることを示す．これは単なるデータ表現ではなく，Rubyのメソッド呼び出しの形式で記述されている．要素を複数並べて要素列が出来上がる．例えば，

```
var "x"
line "3"
column "1"
```

は，var要素の次に，値が"3"のline要素と，値が"1"のcolumn要素を表現している．REFORMでは，値を表現するために4つの基本データ型（int, real, bool, string）を準備している．以下の例では，4つの要素price, tax, togo, dateがあり，price要素は整数値340，tax要素は実数値0.05，togo要素は論理値true，date要素は文字列"December 13, 2008"を持つ．

```
price 340
tax   0.05
togo  true
date  "December 13, 2008"
```

3.2. カスタマイズ可能なデータ処理

データ構造の定義から生成されたプログラム群を使って，3.1で述べた要素列を実行時に木構造に変換する．これは，コンパイラの字句解析部でトークン列が生成され，そのトークン列をパーザが受け取って木構造を作り上げていく手順と同じと考えてよい．図１にその概略を示す．

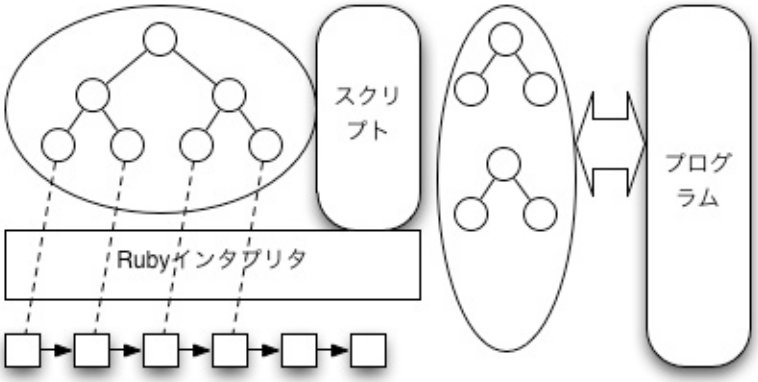

図 1　REFORM の概要

REFORM では，要素列をそのまま Ruby インタープリタで実行すると，全ての要素一つずつに対してイベントが発生するようになっている．要素列で表現されたデータは，Ruby インタープリタ上で動作するスクリプトを切り替えることで，唯一の木を引数としたイベントとして受け取ることもできる．例えば，Java プログラムの AST を考えてみる．各トークンを一つずつ受け取りたいときは，AST に相当する要素列をそのまま実行することでイベント中心の処理となる．一方，AST を木構造として受け取りたいときは，木構造に特化したスクリプトに切り替えることで，AST 一つが付随するイベントを受け取ることができる．また，クラス内で定義されているメソッド定義を順番に受け取りたいときは，メソッドを分割できない固まりと指定することで，メソッド列が付随するイベントを受け取ることができる．

4. まとめ

本稿では,SAXのようにイベント中心に扱うことができ,DOMのように木構造データを処理することができ，データ構造の定義からデータ処理のためのクラス群を生成することができ，さらには，木構造アクセスのための方法をカスタマイズ可能な手法REFORMの構想について述べた．REFORMは，コンパイラにおけるパーザ構築の経験に基づき，柔軟なデータ処理を実現するための手法とツール群である．今後，設計と実装を進めていき，製品開発での応用を通して，その評価を進めていきたい．

参考文献

[1] Alfred V. Aho 他, Compilers: Principles, Techniques, and Tools, Second Edition, Pearson Education, 2006.
[2] James Gosling 他, The Java Programming Specification, Third Edition, Addison-Wesley, 2005.
[3] Greg Badros, “JavaML: A Markup Language for Java Source Code”, The 9th International World Wide Web Conference, 2000.
[4] Gregory McArthur 他, “An Extensible Tool for Source Code Representation Using XML”, The 9th Working Conference on Reverse Engineering, 2002, pp.199-209.
[5] Tak Cheung Lam 他, “Operational and Performance Characteristics”, IEEE Computer, vol.41, no.9, 2008, pp.30-37.
[6] jaxb: JAXB Reference Implementation, https://jaxb.dev.java.net/.
[7] The Castor Project, http://www.castor.org/.
[8] Eric Gamma 他, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[9] Ouafa Hachani 他, “Using Aspect-Oriented Programming for Design Patterns Implementation”, Workshop on Reuse in Object-Oriented Information Systems Design, 2002.
[10] Gregor Kiczales 他, “Aspect-Oriented Programming”, ECOOP’97, Lecture Notes in Computer Science, vol.1241, Springer, 1997, pp.220-242.