情報処理学会論文誌 Vol.58 No.3 644–653 (Mar. 2017)

オープンデータ時代のスキーマ・フュージョン技術

山口 真弥[1,a)] 倉光 君郎[1,b)]

受付日 2016年5月31日, 採録日 2016年12月1日

概要：スキーマ検証はデータ交換に信頼性をもたらす重要な技術であり，古くから研究されてきた．しかし，近年利用がさかんになっているオープンデータはデータフォーマットが多様であるため，スキーマ検証が利用できないという問題がある．我々はパーサジェネレータ技術の基盤となっている解析表現文法（PEG）に注目し，これを多様なフォーマットに対するスキーマ検証に活用する．本論文では，スキーマに従って PEG を書き換えるスキーマ・フュージョン技術を示す．また，スキーマ・フュージョンによって得られるスキーマ検証器の実行時間を既存の検証器と比較して評価を行う．

キーワード：スキーマ検証，オープンデータ，スキーマ言語，解析表現文法，パーサジェネレータ，構文解析

A Schema Fusion Technique for Open Data Era

Shin'ya Yamaguchi[1,a)] Kimio Kuramitsu[1,b)]

Received: May 31, 2016, Accepted: December 1, 2016

Abstract: Schema validation is an important technique that achieve a reliability of data exchange on the Web. Historically, schema validation has been developed intensively in XML. However, open data, which become popular recently, are transmitted and received in various formats. Since we can't apply XML schema validators to another format data, schema validation is not available in most of open data. In this paper, we propose *Schema Fusion* algorithm, which provides a schema constrained PEG by creating a fusion of a schema definition and a PEG representing syntax of data format. By applying our algorithm, we can obtain generated parsers performed as schema validators that validate open data in various formats. We evaluate the performances of generated schema validators and confirm its practicality by comparing an existing schema validator.

Keywords: schema validation, open data, schema language, parsing expression grammar, parser generator, parsing

1. Introduction

スキーマ検証は, 古くて新しい問題である．インターネットがデータ交換に利用される以前から，EDI（Electronic Data Interchange）をはじめとするデータの送信側と受信側のデータの一貫性を高める技術が開発されてきた．ただし，それらは閉鎖的なネットワークを前提としていたため，送受信側の相互運用性はそれほど複雑でなかった．

Web 上でデータ交換が行われるようになると，データの送信側と受信側の関係が複雑化し，よりオープンなものとなった．その中で，標準的なデータ形式として XML が提案され，2000 年代前半には XML 上でスキーマ技術がさかんに研究された．XML/DTD，RelaxNG，XML Schema が標準化されるに至り，やがて研究として下火になった．

さらに現在ではオープンデータの利用が活発になり，誰でも自由に利用できるデータが Web 上に溢れるようになった．オープンデータには社会基盤に関わるデータも含まれており，新たな社会インフラとしての重要性が高まっている．

さて，2000 年代前半の Web 普及期と現在では大きな違いがある．それは，Web 上で取り扱われるフォーマットの多様性である．XML が主流であった時代と比較して，

[1] 横浜国立大学
Yokohama National University, Yokohama, Kanagawa 240–8501, Japan
[a)] yamaguchi-shinya-tm@ynu.jp
[b)] kimio@ynu.ac.jp

© 2017 Information Processing Society of Japan

JSON や CSV などの，より軽量なフォーマットやアドホックなログ形式など様々なフォーマットが用いられるようになった．利用者は用途に合わせてフォーマットを選択するようになったのである．

ここで問題となるのは，スキーマ検証である．スキーマ検証の技術は XML 形式に集中しており，それらの技術を他のフォーマットで用いることはできない．つまり，現在 Web 上で交換されるデータのほとんどに対してスキーマ検証を行うことができない．スキーマ検証を行うことができなければ，交換されるデータの一貫性は保証されない．

今後社会インフラとしてより多くのオープンデータが用いられるためには，データ交換の一貫性を保証する必要がある．

我々はパーサジェネレータ技術の基盤である解析表現文法（PEG）に注目し，これをオープンデータ時代のフォーマットが多様なデータに対してスキーマ検証を実現するアプローチとして活用する．本論文では，スキーマ・フュージョン技術を提案する．スキーマ・フュージョンとは，スキーマ定義に従ってデータフォーマットの構文を表す文法定義の一部を書き換えることでスキーマ定義によって入力が制限されたパーサを自動生成する技術である．このパーサで入力を処理することでスキーマ検証を行うことができる．

本論文の構成は次のとおりである．2 章では現在の Web 上のデータ交換についての現状と課題を分析し，3 章ではオープンデータ時代のスキーマ検証器への要求と方針を述べる．4 章ではスキーマ・フュージョン技術の詳細を説明し，5 章では PEG によるスキーマ検証器の評価を行う．

2. 現状と課題

スキーマ検証は古くて新しい問題である．Web 上でのデータ交換が始まると同時に，多くのスキーマ検証技術に関する研究が行われた．いくつかのスキーマ言語は標準化され，実用的な技術として確立された．しかし，現在のデータ交換は，スキーマ検証に関して新たな問題をかかえている．それは，XML が主流であった時代と比較して，多様なフォーマットが扱われるようになったことに起因する．

本章では，現在の Web 上でのデータ交換について，特に近年さかんに利用されているオープンデータを中心に現状と課題を分析する．我々の分析の結果，以下の 3 点が明らかになった．

- オープンデータの重要性が高まっている．
- スキーマは標準化に向かっている．
- フォーマットは多様になっている．

2.1 データの利用

まず，データの利用シーンから，過去と現在におけるデータ交換の違いを分析する．

初期の Web では EDI（Electronic Data Interchange）と呼ばれる方式でデータ交換が行われていた．それらは，送受信者双方が特定の規約に基づいてデータ交換をしていた．つまり，扱われるデータの構造や内容が制限されており，受信者やデータのドメインが限定されていた．この方式は固定の規約でデータをやりとりするため，商取引におけるあらゆる手続きを自動化できるというメリットがあるが，規約の変更によってシステム全体を作り変えなければならないという問題があった．

一方で，現在のオープンデータは，標準化されたオープンなフォーマットで提供されている．言い換えれば，データの内容が制限されず，自由に誰もが扱える形式で交換される．EDI 方式と比較して，不特定多数の受信者を想定しているため，データのドメインが受信者側に委ねられている．オープンデータを有効に活用している例として，米国の The Climate Corporation が提供する農業経営者向けリアルタイム情報配信サービス Climate Field View がある [7]．このように，受信者側がデータを自由に扱うことで新しい価値やビジネスを創出されており，今後オープンデータが普及するに従って，より応用領域が拡大していくものと考えられる．

また，オープンデータは社会基盤システムを構築する社会インフラとしての重要度が高まっている [25]．オープンデータとして公開されているデータには道路交通，上下水道，電力，その他公共設備に関する情報など，社会基盤に関するデータが含まれており，それらを公共インフラの維持管理に役立てる取り組みやアプリケーションを通して市民に様々な情報を提供するサービスが注目されている．

2.2 オープンデータの課題

オープンデータの社会インフラとしての重要度が高まっている一方で，我々はオープンデータが不特定多数の受信者から利用されているという事実に注意しなければならない．特に，データのフォーマットやスキーマの変更は，システムやアプリケーションの動作に大きく影響する．送信者側の変更によって実際に問題が発生した事例として，2011 年 7 月に東京電力が提供していた電力使用状況データの変更がある．提供していた CSV 形式のスキーマが突然変更されたことによって，電力モニタリングアプリなどのアプリケーションにおいていっせいに誤動作が発生した [15]．このようなトラブルを防ぐためには，送信者と受信者がデータのスキーマを共有し，交換されるデータに対してスキーマ検証を行う必要がある．

2.3 スキーマ語彙標準

スキーマ検証は Web 上のデータ交換の信頼性を高める手段として古くから用いられてきた．データの利用者はスキーマ定義を共有することによって，送られてくるデータ

© 2017 Information Processing Society of Japan

表 1 オープンデータカタログにおけるフォーマット別データ件数
Table 1 Format-specific number of data in the open data catalogs.

フォーマット	DATA.GOV (US)	DATA.GOV.UK	DATA.GO.JP
HTML	74,825 (22.0%)	11,433 (37.3%)	6,286 (29.7%)
XML	42,632 (12.6%)	389 (2.7%)	151 (0.7%)
PDF	35,082 (10.3%)	1,074 (3.5%)	8,733 (41.3%)
CSV	12,472 (3.7%)	5,376 (17.5%)	720 (3.4%)
JSON	10,859 (3.2%)	196 (0.6%)	0 (0%)

が必要な情報を持っているかどうかを検証することが可能になる．特に XML 形式ではスキーマに関する研究がさかんに行われており，XML Schema や XML/DTD などの標準化されたスキーマ言語が存在する．

スキーマ定義はスキーマ言語によってその定義が記述される．利用者はスキーマ定義に用いる語彙を自由に選択できるためデータの関係が曖昧になる場合がある．たとえば，`Name` という語彙を用いた場合，データの作成者以外は，それが何の名前を指し示しているかは正確に知ることができない．それが人物の名前であるのか，別の何かの名前であるのかはデータを読み取って文脈から判断するしかないため曖昧である．

スキーマ定義の曖昧性をなくすための取り組みとして，スキーマ定義に用いる語彙の標準化が行われている [3], [8], [18]．語彙を標準化することで，スキーマ定義を明確に定めることができ，データの意味を機械的に判断することが可能になる．本論文では，このような語彙の標準をスキーマ語彙標準と呼称する．

2.4 フォーマット多様性

現在 Web 上でのデータ交換に用いられるフォーマットは多様である．従来から利用されてきたテキストデータフォーマットである XML や CSV に加えて RDFa や microdata などのメタデータを含んだ新しいフォーマットも存在する．一方で，JPEG や PDF など，画像や特定のソフトウェアに依存したフォーマットで交換されることも依然として少なくない．オープンデータにおいてもフォーマットの多様さは同様であり，米国のオープンデータカタログである DATA.GOV [23] では 49 種類ものフォーマットを取り扱っている．

表 1 は 2016 年 2 月時点の米国（DATA.GOV），英国（DATA.GOV.UK [22]），日本（DATA.GO.JP [9]）におけるフォーマット別のデータ件数をそれぞれまとめたものである．掲載しているフォーマットは，DATA.GOV における利用数の上位 5 種類のフォーマットである．括弧内の数値は全データに占める割合を表している．3 つのカタログサイトに共通して，大きな割合を占めているフォーマットは HTML であり，スキーマ技術が早くに確立された XML 形式は必ずしも主流ではない．むしろ，XML 形式はあまり好まれず，CSV や JSON 形式が利用されることも多い．すなわち，オープンデータにおいてスキーマを利用するためには，フォーマットの多様性に対応することが求められる．

3. オープンデータ・スキーマ技術

2 章における現状と課題分析に基づき，オープンデータに求められるスキーマ検証技術について述べる．

3.1 要求

オープンデータ時代のスキーマへの要求は以下にまとめられる．

(1) スキーマ語彙標準への準拠
(2) フォーマット非依存

“スキーマ語彙標準への準拠” とは，スキーマ定義を構成する語彙のすべてがスキーマ語彙標準で定義されていることを意味する．つまりスキーマ語彙標準に準拠したスキーマ定義には曖昧性がなく，それに従ったデータは機械的に処理ができる．

フォーマット非依存とは，スキーマ定義やスキーマ検証が特定のフォーマットに依存しないという性質である．オープンデータが多様なフォーマットで提供されることは 2.4 節で示したとおりである．今や，Web 上ではあらゆるフォーマットが氾濫しており，すべてのデータを 1 つのフォーマットに統一することは発行者・利用者の双方にとって現実的ではない．フォーマット非依存なスキーマが実現すれば，1 つのスキーマ定義に対して複数のフォーマットのデータを検証することができ，データ交換の信頼性を向上させることができる．

スキーマ検証器には実用的な性能が求められる．スキーマ検証器の実装面での要求は以下の 2 点である．

(3) 線形時間での動作
(4) 複数のプログラミング言語環境で動作

過去の XML におけるスキーマ検証の研究において，ストリーミング処理は重要な研究対象として扱われてきた [19]．ストリーミング処理は，入力長が不明であるデータを扱う．そのため，これらの研究において入力長に対するスキーマ検証時間の線形時間性は重要な目標であった．本研究においても，将来的にストリーミング処理への対応を目指しており，線形時間での動作を目標とする．

2.1 節では，オープンデータの応用領域が不特定多数の利用者に委ねられていることを示した．このことから，利用者は様々なプログラミング言語環境でデータを扱うことが容易に想像できる．すなわち，オープンデータを扱うスキーマ検証器は特定の言語だけでなく，あらゆる言語で動作することが求められる．

3.2 方針

3.1 節であげた要求を満たすスキーマ検証器を実現する

© 2017 Information Processing Society of Japan

表 2 PEG によるデータフォーマットの文法定義
Table 2 Syntax definition of data formats described in PEGs.

フォーマット	構文規則数
HTML	28
XML	16
CSV	5
JSON	13
RDFa	21
microdata	19

```
File      = _ Value _ !.
Value     = String / Number / Object / Array
          / Null / True / False
Object    =  '{' Member (',' Member )* '}'
Member    =  String _ ':' _ Value
Array     = '['  Value (',' Value )*   ']'
String    = '"' (!'"' .)*  '"' _
True      = 'true' _
False     =  'false  _
Null      = 'null'  _
Number    = '-'? INT (FRAC EXP?  / '' ) _
INT       = '0' / [1-9] [0-9]*
FRAC      = '.' [0-9]+
EXP       = [Ee] ( '-' / '+')? [0-9]+
_         = [ \t\r\n]*
```

図 1 PEG で定義した構文の例（JSON）
Fig. 1 A JSON syntax definition in PEGs.

ため，我々はパーサジェネレータ技術に注目する．パーサジェネレータ技術は構文解析器（パーサ）を自動生成するための技術である．パーサの自動生成は，形式文法で定義された構文規則に従って行われる．パーサジェネレータに用いられる代表的な形式文法としては，文脈自由文法（*context free grammar*, *CFG*）や解析表現文法（*parsing expression grammar*, *PEG* [5]）がある．我々は，特に PEG に着目する．理由は，PEG によるパーサジェネレータが前節であげた要求 (2)，(3)，(4) を満たしやすいためである．PEG はオープンデータで用いられるほとんどのテキストデータフォーマットの構文を記述する表現力を持っている．**表 2** は PEG によって定義された DATA.GOV の利用数が上位にあるテキストデータフォーマットの文法と，その構文規則数をまとめたものである．また，パックラット構文解析 [4] による線形時間での解析手法が知られている．さらに，多くの言語環境での実装が存在するため，PEG で構文を記述すれば PEG パーサジェネレータが対応している言語上でパーサを動作させることができる．

例として JSON の構文を PEG で記述した文法を**図 1** に示す．PEG の構文規則は，非終端記号 A と解析表現 e を用いて，$A = e$ という形式で表される．この例では，13 の構文規則で JSON の構文を定義することができ，この文法から JSON データを受理するパーサが生成される．

我々は，オープンデータのスキーマ検証に求められる要求に対応するため，PEG とスキーマ語彙標準から定義されたスキーマ定義を融合する．本論文ではこの方式をスキーマ・フュージョンと名付ける．具体的な方針としては，フュージョン・アルゴリズムによってスキーマ定義による制約を付加した文法を生成し，その文法から自動生成されるパーサをスキーマ検証器として機能させる．本方式の利点として，スキーマ定義を PEG に融合することができれば，パーサジェネレータをそのまま利用することができ，個別のフォーマットごとにスキーマ検証器を実装しなくてもよい点がある．つまり，文法定義を用意すれば，1 つのスキーマ定義から複数のフォーマットに対するスキーマ検証付きパーサを生成することが可能になり，フォーマット多様性に対応することができる．また，パースとスキーマ検証を同時に行えることも利点の 1 つとしてあげられる．既存のスキーマ検証器の多くはテキストデータをパースした結果を用いて検証を行うため，この点で実行時間の短縮が期待される．

```
<p itemscope itemprop="Person"
            itemtype="http://schema.org/Person">
  <span itemprop="name">Rose Tyler</span> was sponsored by
  <span itemscope itemprop="sponsor"
                 itemtype="http://schema.org/Person">
    <span itemprop="name">Sarah Jane Smith</span>
  </span> in the membership process.
</p>
```

図 2 Schema.org の利用例（microdata 形式）
Fig. 2 An application of Schema.org (microdata format).

また，スキーマ語彙標準として Schema.org を利用する．Schema.org [18] は Google や Microsoft，Yahoo などによって 2011 年に設立された，代表的なスキーマ語彙標準の 1 つである．実際に，Google や Bing などの検索エンジンで採用されており，主に Web ページ上で標準化された語彙を用いてマークアップを記述することで，効率的な検索やリッチスニペットの表示などに利用されている．**図 2** は microdata 形式で Schema.org を利用する例である．Schema.org で定義されているスキーマは，プロパティの集合で構成される木構造で表される．現在，既存のあらゆるスキーマ語彙標準が Schema.org に統合されており，様々な分野の語彙を広くカバーしている．

4. スキーマ・フュージョン

本章では，本論文の中心的アイデアであるスキーマ・フュージョンについて説明する．

スキーマ・フュージョンとはデータフォーマットの文法を，スキーマ定義に対応する文法に書き換えることである（**図 3** 参照）．具体的には，形式文法 Nez で記述された JSON や XML などのデータフォーマットの文法に対してスキーマ言語 Celery で記述されたスキーマ定義を文法変

© 2017 Information Processing Society of Japan

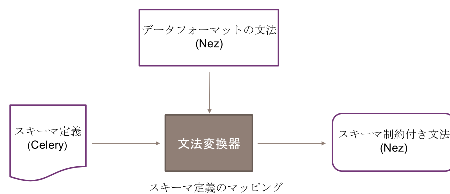

図 3　スキーマ・フュージョン
Fig. 3　Schema fusion.

C_S	:=	**type** TypeName = ($propertyName:Type)+	**type** 宣言
TypeName	:=	[A-Z] [a-zA-Z]+	型名
propertyName	:=	[a-z] [a-zA-Z]+	プロパティ名
Type	:=	TypeName	スキーマ型
	|	#Primitive	プリミティブ型
	|	Type / Type	選択型
Primitive	:=	Boolean	ブール型
	|	Number	数値
	|	Text	文字列
	|	Date	ISO 8601 フォーマット
	|	Time	時刻フォーマット
	|	DateTime	日付と時刻

図 4　Celery 構文定義
Fig. 4　The syntax definition of Celery.

換器を介して融合する（スキーマ定義のマッピング）．この，スキーマ制約を融合したデータフォーマットの文法を**スキーマ制約付き文法**と呼ぶ．そして，スキーマ制約付き文法から生成されるパーサを用いて入力をパースすることで，スキーマ検証を行う．すなわち，パースが成功すればデータはスキーマ定義に対して有効であり，失敗すれば無効なデータと判断する．本章では，スキーマ言語 Celery と，スキーマ定義を対応付けるための PEG ベースの形式文法 Nez，そしてそれらを融合する文法変換アルゴリズムの詳細を順に述べる．

4.1　スキーマ言語：Celery

スキーマ·フュージョンで扱うスキーマ定義は Schema.org に準拠する．Schema.org のスキーマ定義は型（types）の集合として表される．型はそれぞれプロパティ（properties）の集合からなる．プロパティは別の型か，Boolean や Number などのプリミティブなデータ型を持っている．プロパティには複数の型からなる集合型を定義でき，集合のいずれかの型に属する値がマッチする．Schema.org ではこれらの型どうしの関係が階層的に定義されている．スキーマ・フュージョンでは以上の Schema.org の性質を表現するためのスキーマ言語が必要となる．

本研究では，スキーマ定義のための言語として Celery を用いる．Celery はシンタックスに依存しないスキーマ言語である．Celery においてスキーマ定義は**型**と呼ばれる．Celery の構文 C_S は図 4 に定義される．

Celery の型は複数の **type** 宣言によって構成される．この **type** 宣言は，Schema.org の型を指定し，そのプロパティを定義する．このうち，TypeName と propertyName はそれぞれ Schema.org の型とプロパティの名前に対応している．プロパティに対する型定義は Type によって与えられる．スキーマ型は **type** 宣言で定義された型を呼び出す型であり，プリミティブ型は Primitive で定義される Schema.org 上の基本データ型を表す．選択型は 2 つの型のうちどちらかの型にマッチすることを表す型である．これらの構文によって Celery は Schema.org と同等の表現力を持つ．

type 宣言で定義された型は，指定したプロパティのみが必須要素となる．また，プロパティの型定義が Schema.org での定義と異なる場合は，**type** 宣言内で指定した型が優先される．

以下は Person 型（schema.org/Person）を Celery で定義した例である．

```
type Person {
  $name:#Text
  $birthDate:#Date
  $adderess:PostalAdress / #Text
 }
```

4.2　形式文法：Nez

我々は，スキーマ定義を対応付ける PEG として純宣言的 PEG 拡張言語 Nez [12] を用いる．Nez を用いる理由として，Nez は抽象構文木（AST）の構築を宣言的に記述できることがあげられる．Nez の AST 構築と Schema.org の木構造を対応付けることで，Nez 文法にスキーマ定義をマッピングする．さらに，書き換えられた文法から生成されるパーサからはスキーマ検証の結果（成功/失敗）だけでなく，スキーマ定義によって型付けされた AST を得ることができる．また，Nez は各言語環境に対するパーサジェネレータや，Nez 文法から他の PEG 文法への変換器，文法や解析アルゴリズムの最適化 [11] などを提供しており，3.1 節で定義した要求に合致する形式文法であるといえる．本節では Nez の文法，特に AST 構築演算子について説明する．

表 3，**表 4** に Nez の演算子の一部をまとめた．このうち，表 3 の PEG 演算子は，参考文献 [5] での定義と同じである．

4.2.1　Nez 拡張文法

Nez の解析表現文法への拡張の 1 つが，AST を柔軟に構築する演算子である（表 4）．Nez が構築する AST の各ノードには入力から切り取られた文字列が格納される．Nez では演算子 { e } を用いてノードを構築する文法上の部分を特定する．この演算子が囲む範囲で受理された文字列がノードに格納される．以下は構文規則 Name が CHAR

© 2017 Information Processing Society of Japan

表 3　PEG 演算子
Table 3　PEG operators.

PEG	説明
'abc'	文字列
[abc]	文字クラス
.	任意の 1 文字
A	非終端記号
(p)	グルーピング
$p?$	オプション
$p*$	0 文字以上の繰返し
$p+$	1 文字以上の繰返し
$\&p$	肯定先読み
$!p$	否定演算子
$p_1 p_2$	連接
p_1/p_2	優先度付き選択

表 4　Nez 拡張演算子
Table 4　Nez operators.

AST	説明
{ e }	ノードの構築
$lable($e$)	子ノードの指定
{$ e }	左結合
#t	タグ付け
' '	文字列置換

で定義される文字列を取り出す例である．

```
CHAR = [a-zA-Z0-9]
Name = { CHAR+ #Name}
```

Name は CHAR が一回以上マッチに成功した場合に，マッチした文字列をノードに格納する．また，演算子 #t はノードの型を区別するためのタグを付ける演算子である．タグによって，構築したノードに対して名前を付けることができる．

演算子 $lable($e$) は 2 つの AST ノードを親子関係で接続し，木構造を作る演算子である．また，$label$ に任意の名前を付けることによって，AST 走査の際に子ノードを指定して取り出すことが可能になる．たとえば，上記の例で名前のリストを構築したい場合，以下のように記述することができる．

```
CHAR = [a-zA-Z0-9]
PersonName = { $given(Name) _ $family(Name) #PersonName}
Name = { CHAR+ #Name}
```

以上の AST 構築演算子で定義される構文規則から生成される構文解析器が，実際に入力を解析して出力する AST の例は以下のように表現される．

```
#PersonName[
  $given#Name['John']
  $family#Name['Doe']
]
```

4.2.2　スキーマ定義のマッピング

スキーマ・フュージョンでは，Nez の AST 構築演算子を利用して，データフォーマットの構文規則に対して注釈をつける．この注釈をもとに，文法に対して Celery で定義された Schema.org のスキーマ定義をマッピングする．ここで注意しなければならないことは，Schema.org は木構造で定義されているのに対して，データフォーマットは必ずしも木構造のデータモデルを持っていないということである．たとえば，CSV は関係モデルでデータを表現しており，値と区切り文字のみでデータが構成されている．一方で Schema.org のプロパティはキーと値を持っているため，CSV の文法定義に対してスキーマ定義を直接マッピングすることができない．そこで我々は以下の 2 種類の注釈を文法定義に付加する．

- $object：キーと値を持つ構文
- $list：値のみで構成される構文

これらの注釈は実際の文法中では子ノードの指定演算子のラベルと対応する．つまり，$object(A) と記述された非終端記号 A はキーと値を持つ構文であると判別される．さらに，$object ラベルが注釈された構文規則中にはプロパティを表す $prop ラベルを指定された構文規則が定義される．また，プロパティを表す構文規則にはキー（$key）と値（$value）に対応する構文規則が定義される．一方で，$list としてラベル付けされた構文規則の中には値（$value）に対応する構文規則が定義される．このように，構文規則にラベル付けすることで注釈を与える．この方法は元来存在する構文規則に対してラベル付けを施すため，文法定義そのものの構成を変更するものではない．また，これら 2 種類の注釈を付加できれば，オープンデータとして利用されるデータフォーマットに対して Celery のスキーマ定義をマッピングできる．

表 5　PEG による注釈付き文法定義
Table 5　Annotated grammar definitions by PEGs.

フォーマット	構文規則数	$object の数	$list の数
HTML	28	2	1
XML	16	2	1
CSV	5	0	1
JSON	13	1	1
RDFa	21	1	0
microdata	19	1	0

表 5 は表 2 の各文法に対して $object と $list の注釈を付与した結果を表す．いずれも構文規則数に変化はなく，$object と $label のみでデータモデルに対応するフォーマットの特定が行える．これらの結果より，オープンデータとして利用される頻度の高いデータフォーマットに関して $object と $label の注釈のみで対応できることを確認できる．

4.2.3　スキーマ定義と AST の対応

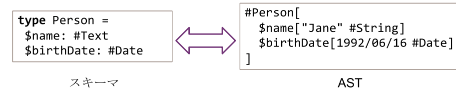

図 5　スキーマ定義と AST の対応
Fig. 5　Correspondence of schema definitions and ASTs.

図 5 のようにスキーマ定義と Nez 文法上の AST 構築を対応付けることで，パースの結果としてスキーマ定義によって型付けされた AST を得る．つまり，任意のデータフォーマットの文法に対して，スキーマ定義が同じであれば同一の AST を取得することができる．

最後に，図 1 を AST 構築演算子を用いて再定義した文法を**図 6** に示す．PEG の演算子のみで定義した文法に対

© 2017 Information Processing Society of Japan

```
File    = _ { Value #JSON} _ !.
Value   = $(String) / $(Number) / $object(Object) / $list(Array)
        / $(Null) / $(True) / $(False)
Object  = { '{' $prop(Member) (',' $prop(Member) )* '}' #Object}
Member  = { $key(String) _ ':' _ $value(Value) #Member}
Array   = '[' { $value(Value) (',' $value(Value) )*  #Array} ']'
String  = '"' { (!'"' .)*  #String }'"' _
True    = { 'true' #True} _
False   = { 'false #False } _
Null    = { 'null' #Null } _
Number  = { '-'? INT (FRAC EXP? #Float / '' #Integer) } _
INT     = '0' / [1-9] [0-9]*
FRAC    = '.' [0-9]+
EXP     = [Ee] ( '-' / '+')? [0-9]+
_       = [ \t\r\n]*
```

図 6 Nez で定義した構文の例（JSON）
Fig. 6 A JSON syntax definition in Nez.

してわずかな変更を加えるだけで AST の構築を行えることが分かる．また，`$object`，`$list` をはじめとした注釈が付加されており，データフォーマットの特定を行える．

4.3 文法変換アルゴリズム

文法変換アルゴリズムの基本的な方針として，変換対象となるデータフォーマットの文法定義をトップダウンで解析し，スキーマ定義を対応付ける文法上の位置を特定して書き換えを実行する．変換アルゴリズムは以下の 3 つの手順で実行される．

(1) *Format Specifying*：スキーマ定義を対応付ける構文規則を特定
(2) *Property Deploying*：特定した構文規則に対してスキーマ定義のプロパティを展開
(3) *Schema Merging*：展開したプロパティの構文規則を合成した新しい構文規則を生成

以下では，図 6 の JSON 文法に対して，次のスキーマ定義をフュージョンする例によって各手順を説明する．

```
type Person =  $name:#Text $birthDate:#Date
```

4.3.1 Format Specifying

まずはじめに，文法定義に付加された注釈から，データフォーマットの特定を行う．具体的には，`$obejct` か `$list` の注釈を持った構文規則を特定する．たとえば，図 6 の JSON 文法では構文規則 `Object` が構文規則 `Value` 中で注釈 `$object` を付加されているため，以下の構文規則が特定される．

```
Object = {'{' $prop(Member) (',' $prop(Member)*)'}' #Object}
```

4.3.2 Property Deploying

Format Specifying によって構文規則が検出されると，*Property Deploying* では，その構文規則に存在する `$prop` で注釈された解析表現に対してスキーマ定義内に定義されたプロパティの展開を行う．この例では構文規則 `Object` 中の非終端記号 `Member` が表す構文規則に対してプロパティを展開し，スキーマ定義による制約を付加する．

```
Member = {  $key(String) ":" $value(Value) #Member}
```

Algorithm 1 *deployObjectProperty* function
Input: Property p, Expression[] *expList*
Output: Expression[] *expList'*
  Expression e'
  for all Expression e : *expList* **do**
    if *e.label is key* **then**
      $e' \Leftarrow embedKey(p.name, e)$
      $expList'.add(e')$
    else if *e.label is value* **then**
      Type $t \Leftarrow p.type$
      $e' \Leftarrow fusion(t, e)$
      $expList'.add(e')$
    else
      $expList'.add(e)$
    end if
  end for
  return *expList'*

プロパティの展開は各フォーマット型の *deploy* 関数によって実行される．**Algorithm 1**（*deployObjectProperty*）は *Object* 型の *deploy* 関数を擬似コードで表したものである．*Object* 型の場合，プロパティはキーと値によって構成されるため，それぞれ `$key`，`$value` としてラベル付けられた位置の表現に対して書き換えを実行する．`$key` では，プロパティの名前を直接埋め込み（*embedKey*），`$value` ではプロパティの型を *fusion* 関数によってスキーマ・フュージョンアルゴリズムを再帰的に適用し，展開する．*deployObjectProperty* 関数によって，上記の `Member` とスキーマ定義 `Person` のプロパティを基に，以下の構文規則が追加される．

```
Member#Person$name =  '"name"' ":" $name(Text)
Member#Person$age =  '"birthDate"' ":" $birthDate(Date)
```

ここで，Celery のプリミティブ型である `Text` や `Date` の構文規則は本来の JSON 文法に存在しない．それゆえ，必要なプリミティブ型の構文規則はこの段階で文法定義に追加される．

また，注釈が `$list` だった場合，キーが存在しないため，`$value` のみを展開する．

4.3.3 Schema Merging

最後に，*Schema Merging* では，*Property Deploying* によって追加されたスキーマ制約付きの構文規則を，*Format Specifying* で特定された構文規則に対して合成する．つまり，スキーマ定義によって制約された入力のみを受理する構文規則を生成する．

```
Member#Person = Member#Person$name
              / Member#Person$birthDate
```

`$object` で表されるデータフォーマットはキーを持っ

© 2017 Information Processing Society of Japan

ているため，プロパティの出現は順不同である．実際，XML のタグや JSON オブジェクトのプロパティは，特にスキーマ定義による制約がない限り順序に依存しない．それゆえ，プロパティに対応する構文規則 `Member` を `Member#Person$name`，`Member#Person$birthDate` の**選択**（`'/'`）に置き換える．この置き換えによって，生成されたパーサは入力データのプロパティがどの順序で出現しても受理することができる．なお，注釈が `$list` である場合，プロパティは順序制約がなければ判別できないため，これらの構文規則は**選択**ではなく**連接**によって表される．

最後に，フォーマット全体を表す構文規則 `Object` は `Member#Person` を合成することで以下のように書き換えられ，アルゴリズムは終了する．

```
Object#Person =
 { '{' Member#Person (',' (Member#Person)* '}' #Person}
```

5. 実験

本章では，スキーマ・フュージョンによって得られるスキーマ検証器を評価する．我々は，4.3 節で示したアルゴリズムを基にプロトタイプを実装し，実験を行った．

5.1 データセット

本実験ではベンチマーク用のデータセットとして JSON 形式の OSNI Opendata Benchmark [20] を利用した．このデータセットに対してスキーマを定義し，スキーマ・フュージョンによってスキーマ定義を対応付けた Nez 文法を生成した．本章ではこの文法を通常の文法と区別するためにフュージョン文法と呼ぶ．また，フォーマットの違いによる影響を確認するため，データセットを JSON 形式から XML，microdata フォーマットに変換し，それぞれの文法に対して評価を行う．

5.2 性能評価

各フォーマットのフュージョン文法から生成されたパーサの性能評価を行う．実験は以下の環境で実行される．

- OS：Ubuntu 14.04
- CPU：Core i7-4770
- RAM：8 GB
- Java 1.8.0_51
- Nez 1.0-1069
- XercesJ 2.11.0 [24]

また，各入力ファイルに対して 10 回計測を行い，それらの平均値を結果として記録する．

表 6 は各フォーマットにおける，通常文法とフュージョン文法の Nez パーサの実行結果である．各フォーマットで，通常文法よりもフュージョン文法の方が実行時間が短い．この理由は，フュージョン文法はスキーマ定義で指定されたデータ構造のみを受理するように書き換えられており．文法中に固定の値や型が埋め込まれていることによって文字列マッチや優先度付き選択が通常の文法に比べて削減されているためと考えられる．

表 6 通常文法とフュージョン文法の比較
Table 6 A comparison of normal grammars and fusion grammars.

	XML	JSON	microdata
ファイルサイズ [Byte]	3,947,367	2,050,087	5,194,562
実行時間 [ms]			
通常文法	69.51	44.25	87.78
フュージョン文法	34.19	32.73	76.17

表 7 XML/DTD とフュージョン文法の比較
Table 7 A comparison of XML/DTD validator and a fusion grammar.

	フュージョン文法	DTD
実行時間 [ms]	34.19	31.04

さらに実行時間の比較のために XML/DTD スキーマ検証を行った（**表 7**）．Celery によるスキーマ定義と等価な DTD を記述し，Java で記述した実験用プログラムにおいて実行時間を測定した．XML 文法での結果に注目すると，DTD での実行時間と比較して，同程度の実行時間である．つまり，一般的なスキーマ検証器と同等の性能を有しているといえる．

6. 関連研究

DTD [1] や XML Schema [21]，RelaxNG [2] などの XML スキーマ言語は，XML タグや XML 要素に対応するデータ型を定義する．また，これらのスキーマ言語は正規木文法（*regular tree grammar*）によって形式化されており，その意味論は形式的に定義されている [14]．一方で，Schema.org は XML スキーマ言語と異なり，フォーマットに依存する形式ではなく，キーと値の集合で構成されるオブジェクトとしてデータ型を定義する．これによって，プログラミング言語に対してマッピングしやすいという利点を持つ．しかし，我々が知る限りにおいては，Schema.org の意味論はいまだ定義されていない．Schema.org に対応したスキーマ検証器 [6] は存在するが，実装の正しさは明らかでない．

本研究で用いた Nez 以外にも，ANTLR [16], [17] や Yacc [10] など，様々なパーサジェネレータが存在する．それらはプログラミング言語の構文解析を目的として開発されているため，本研究で紹介したスキーマ制約付きの文法から生成されるパーサをスキーマ検証器として利用する例はない．Nez は，他のパーサジェネレータと異なり，文法中に AST 生成演算子を記述することができ，文脈依存な言語を解析するための拡張を備えているため，スキーマ検証に適している．しかし，現時点では，XMLSchema などで実装されている uniqueness 制約や，データ要素の n 回の繰返しなど，意味的なスキーマ制約を表現することがで

© 2017 Information Processing Society of Japan

きない．この点は，Nez の拡張文法を適用することで今後解決していくべき課題である．

また，Parsec [13] などのパーサコンビネータとプログラミング言語の型定義を組み合わせることで，スキーマ・フュージョンと同等のスキーマ検証が実現できると考えられるが，複数のデータフォーマットと多言語環境に対応するには大きな実装コストがかかる．この観点で，Nez と Celery を用いる我々のスキーマ・フュージョンは 1 つのスキーマ定義から複数のデータフォーマット，複数の言語環境に対してスキーマ検証器を自動生成できるため優れている．

7. 結論

本論文では，オープンデータのスキーマ検証技術として，スキーマ・フュージョンを紹介した．本方式は，スキーマ定義とデータフォーマットの構文を表す PEG を組み合わせることによってフォーマット多様性に対応した．また，スキーマ語彙標準に準拠することで，明確なスキーマ定義が可能となった．今後の課題として，uniqueness 制約など，データ構造以外のスキーマ制約を表すために必要な PEG の拡張を示すことがあげられる．

参考文献

[1] Bray, T. et al.: Extensible Markup Language (XML) 1.0 (5th Edition) (2008), available from ⟨https://www.w3.org/TR/REC-xml/⟩.

[2] Clark, J. and makoto, M.: RelaxNG Specification (2001), available from ⟨https://www.oasis-open.org/committees/relax-ng/spec-20011203.html⟩.

[3] Dan Brickley, L.M.: FOAF Vocabulary Specification 0.99 (2014), available from ⟨http://xmlns.com/foaf/spec/⟩.

[4] Ford, B.: Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl, *Proc. 7th ACM SIGPLAN International Conference on Functional Programming, ICFP '02*, New York, NY, USA, ACM, pp.36–47 (online), DOI: 10.1145/581478.581483 (2002).

[5] Ford, B.: Parsing Expression Grammars: A Recognition-based Syntactic Foundation, *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '04*, New York, NY, USA, ACM, pp.111–122 (online), DOI: 10.1145/964001.964011 (2004).

[6] Google: Structured Data Testing Tool (2016), available from ⟨https://search.google.com/structured-data/testing-tool/u/0/⟩.

[7] Government, U.: Climate Field View - DATA.GOV (2010), available from ⟨https://www.data.gov/applications/climate/⟩.

[8] IPTC: rNews Embedding metadata in online news (2013), available from ⟨https://iptc.org/standards/rnews/⟩.

[9] Japan: DATA GO JP (2014), available from ⟨www.data.go.jp/⟩.

[10] Johnson, S.C.: *Yacc: Yet another compiler-compiler*, Vol.32, ell Laboratories Murray Hill, NJ (1975).

[11] Kuramitsu, K.: Packrat Parsing with Elastic Sliding Window, *Journal of Infomration Processing*, Vol.23, No.4, pp.505–512 (online), DOI: http://doi.org/10.2197/ipsjjip.23.505 (2015).

[12] Kuramitsu, K.: Nez: Practical Open Grammar Language, *Proc. 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, New York, NY, USA, ACM, pp.29–42 (online), DOI: 10.1145/2986012.2986019 (2016).

[13] Leijen, D. and Meijer, E.: Parsec: A practical parser library (2001).

[14] Murata, M., Lee, D., Mani, M. and Kawaguchi, K.: Taxonomy of XML Schema Languages Using Formal Language Theory, *ACM Trans. Internet Technol.*, Vol.5, No.4, pp.660–704 (online), DOI: 10.1145/1111627.1111631 (2005).

[15] Okumura, H.：東電の CSV 形式が突然変更された—Okumura's Blog (2011)，入手先 ⟨https://oku.edu.mie-u.ac.jp/~okumura/blog/node/2580⟩.

[16] Parr, T. and Fisher, K.: LL(*): The Foundation of the ANTLR Parser Generator, *Proc. 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, New York, NY, USA, ACM, pp.425–436 (online), DOI: 10.1145/1993498.1993548 (2011).

[17] Parr, T., Harwell, S. and Fisher, K.: Adaptive LL(*) Parsing: The Power of Dynamic Analysis, *Proc. 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '14*, New York, NY, USA, ACM, pp.579–598 (online), DOI: 10.1145/2660193.2660202 (2014).

[18] Schema.org Community Group: Schema.org (2012), available from ⟨https://schema.org⟩.

[19] Segoufin, L. and Vianu, V.: Validating Streaming XML Documents, *Proc. 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '02*, New York, NY, USA, ACM, pp.53–64 (online), DOI: 10.1145/543613.543622 (2002).

[20] SpatialNI: OSNI Open Data – Benchmark data, available from ⟨http://osni.spatial-ni.opendata.arcgis.com/datasets/⟩.

[21] Thompson, H.S., Beech, D., Maloney, M. and Mendelsohn, N.: XML Schema Part 1: Structures Second Edition (2004), available from ⟨http://www.w3.org/TR/xmlschema-1/⟩.

[22] UnitedKingdom: Data.gov.uk (2010), available from ⟨https://data.gov.uk/⟩.

[23] U.S. Government: Data.gov (2009), available from ⟨https://www.data.gov/⟩.

[24] Xerces: Apache Xerces Project (2015), available from ⟨http://xerces.apache.org/⟩.

[25] 新井イスマイル：ソフトウェア技術者から見たオープンデータの魅力，コンピュータソフトウェア，Vol.32, No.3, pp.3_10–3_22（オンライン），DOI: 10.11309/jssst.32.3_10 (2015).

© 2017 Information Processing Society of Japan

山口 真弥

1992 年生．2015 年横浜国立大学理工学部数物電子情報工学科卒業．同年同大学大学院修士課程入学．現在在学中．

倉光 君郎 （正会員）

1972 年生．愛知県出身．2000 年東京大学大学院理学系研究科情報科学専攻博士課程中途退学．同年東京大学大学院情報学環助手．2007 年より横浜国立大学工学部准教授．博士（理学），ACM 会員．

© 2017 Information Processing Society of Japan