情報処理学会論文誌　プログラミング　Vol.9　No.1　1–10 (Feb. 2016)

データ記述言語DFDLとその意味論

戸澤 晶彦[1,a)]　佐藤 直人[1]　河内谷 清久仁[1]

受付日 2015年7月3日, 採録日 2015年10月27日

概要：近年，情報技術において ad-hoc データあるいは legacy データと呼ばれる，独自規格のテキストあるいはバイナリデータをどう取り扱うかということが課題になってきている．このような独自規格データを統一的に扱うための言語をデータ記述言語といい，DFDL（Data Format Description Language）もその 1 つである．ところでデータ記述言語には既存のパーサの仕組みではとらえられない部分があり，特にデータの中身に依存したパースを行う機能が要請される．このようなデータ記述言語の特徴づけとして，依存型を用いた体系を Fisher らが提案している．本論文ではこれにならい，データ記述言語 DFDL の意味論を与え，その処理について議論する．

キーワード：データ記述言語，パーサ，意味論

The Data Description Language DFDL and Its Semantics

Akihiko Tozawa[1,a)]　Naoto Sato[1]　Kiyokuni Kawachiya[1]

Received: July 3, 2015, Accepted: October 27, 2015

Abstract: Recently, in modern computer technology, there's increasing need to handle non-standard text and binary data called the ad-hoc or legacy data. The languages that handle such non-standard data are called data description languages, and DFDL (Data Format Description Language) is one example. Such data description languages cannot be captured by existing parser framework, in particular because the non-standard data is often parsed dependent on the content of the data itself. In order to characterize such feature of data description languages, Fisher et al. proposed the formal system using dependent types. In this paper, based on their work, we give semantics to the DFDL language and discuss its processor.

Keywords: data description language, parsers, semantics

1. はじめに

近年，情報技術においては XML あるいは JSON のような標準あるいはデファクトの標準データフォーマットが普及し，現在の多くのツールやプログラミング言語がこれらをサポートするようになっている．しかし一方で，ad-hoc データあるいは legacy データと呼ばれる，独自規格のテキストあるいはバイナリデータがいまだに存在している．たとえば COBOL などの古い言語の出力したデータ形式あるいは，EDIFACT [9]，HL7 [5] などの業界独自のデータ形式がその例である．

このような独自規格のデータを与えられたときに，それを最新のソフトウェアアーキテクチャやツールを使って処理したいというニーズが高まっている．このために ad-hoc データのパース，アンパースの方法を記述し，現在の標準的なデータ形式との変換を容易に可能にするための言語をデータ記述言語という．

DFDL（Data Format Description Language）[8] は OGF（Open Grid Forum）で制定された業界標準のデータ記述言語である．IBM では，IIB（IBM Integration Bus）製品および DataPower 製品に処理系が搭載されており，重要な技術となっている．

DFDL には非形式的に記述された仕様はあるが，形式的な仕様や，それに基づいた各種の整合性の検査などはまだあまり議論されていない．そこで本論文では，DFDL 仕様の要点を形式的に記述した，DFDL0 という体系を与え，その処理系の意味論の定義を通して，DFDL 仕様の解説ある

[1] 日本アイ・ビー・エム（株）東京基礎研究所
IBM Research - Tokyo, Chuo, Tokyo 103–8510, Japan
[a)] atozawa@jp.ibm.com

いはそのいくつかの課題についての議論を行う．

2. 関連研究

プログラム言語処理の分野では LR(k)，LL(k) のような文脈自由文法に基づいたパーサの枠組みが主に使われている．しかし，ad-hoc データは通常の文法の導出規則では表せないデータ表現が使われることも多く，これらの枠組みでは不十分である．

ad-hoc データでは特にデータの中身に依存したパースを行う機能が要請される．このためには，パーサのバックトラックなどの挙動を，ユーザがプログラミング的にきめ細かく制御する必要がある．たとえばパーサコンビネータ [7] のように，プログラム言語上でパーサを実装すれば，このような制御が可能になる．ただし，パーサコンビネータは入力ストリームからユーザ定義型の値を返す，普通のプログラムである．

同様の制御を，よりデータ記述言語という応用に特化しながら実現する方法として，依存型を用いた形式体系 DDC を Fisher ら [3] が定義している．また，PADS と呼ばれるこの体系に基づいたデータ記述言語が実装されている．本論文の DFDL0 は依存型の使用や，型の解釈としてパースとアンパースの処理を定義する，という点などで DDC に着想を得ている．一方，相違点として DFDL 言語の特徴であるプロパティ注釈の定式化がある．

DDC や DFDL0 の依存型はパース中のデータを，適当なホスト言語で解釈することができるため，パーサの表現力は，ホスト言語依存となる．すなわち極端なケースでは強力なホスト言語自体に入力を処理させれば，帰納的言語も受理できる．TS [2] あるいは PEG [4] はバックトラックを用いた再帰下降パーサの体系である．パーサとして解釈したとき DFDL0 や DDC の直和型 $+$ は，これらの体系で用いられる優先つき選択オペレータ / と同等のものである．よって，もし DFDL0 や DDC から依存型の機能を除いた（$\Sigma x : \tau_1.\tau_2$ の τ_2 が x を使わない）場合，受理できる言語のクラスは TS や PEG と同等のものになると考えられる．このクラスは TDPL（Top Down Parsing Language）と呼ばれ，LL(k) を含み，また $\mathtt{a}^n\,\mathtt{b}^n\,\mathtt{c}^n$ のような文脈自由でない言語も含む．

3. データ記述計算 DFDL0

データ記述言語に求められる表現能力は文脈自由文法に基づいた既存のパーサの枠組みでは表現できないことも多い．

たとえば，以下のようなテキストが与えられたとする．

```
1A2AB3ABC
```

このテキストの数値部分は続く文字列の長さを示しているとする．つまり，この文字列を (`"A"`, `"AB"`, `"ABC"`) という列としてパースしたい．

このような可変長データ表現は，多くのプログラミング言語でデータを直列化する場合に使われることが多い．PL/I 言語の可変長文字列フォーマット，あるいは ISO8583 データ形式 [6] など DFDL にとって重要なアプリケーションでも用いられている．

図 1 に上のデータをパースするための，DFDL スキーマの例を示す．ここから分かるように，DFDL スキーマはやはり標準仕様である XML スキーマ [10] の拡張となっている．XML スキーマは XML 文書の型を定義するものであるから，DFDL スキーマによるパースの結果は XML であり，与えられたテキストは以下のようにパースされる．

```
<Root>
 <Item><Length>1</Length><Value>A</Value></Item>
 <Item><Length>2</Length><Value>AB</Value></Item>
 <Item><Length>3</Length><Value>ABC</Value></Item>
</Root>
```

実際，図 1 の DFDL スキーマの XML スキーマ部分は，`Root` 要素（20–26 行目）の下に `Length` と `Value` 要素からなる，`Item` 要素（27–34 行目）の列が並ぶということを表しており，上の XML 木はこれと合致している．

DFDL スキーマは XML スキーマの記述に DFDL プロパティと呼ばれる注釈を追加したものになっている．`dfdl:format`（6–17 行目）内に表れるすべての XML 属性，あるいは，要素定義中などに直接表れる `dfdl:length` 属性（30，31 行目）などが DFDL プロパティである．これら DFDL プロパティはパーサ，アンパーサの挙動を指定するものである．すなわち，DFDL 処理系はパーサとしてこれらの注釈を解釈するとき，データを元の XML スキーマに合致するパース木に変換する．また，アンパーサとしてこれらの注釈を解釈するとき，パース木から元のデータを復元する．

たとえば，図 1 のスキーマに基づいたパースでは `length` プロパティの値が適切に与えられることで，例のようなパースが可能になる．ただし，DFDL 言語自体には 100 以上のプロパティが定義されているなど，複雑なため，本論文ですべてを説明することはできない．本論文ではこれから DFDL 言語の要点のみを抽出して，簡単化した DFDL0 という体系を定義し，これに基づいて，DFDL 言語とその処理系の挙動を説明する．

3.1 DFDL0

Fisher ら [3] はデータ型の定義を用いてパーサを定義するという提案をした．本論文もこれにならって，DFDL のモデル DFDL0 を型システムとして定義する．

基本型	b	::=	$\mathtt{unit} \mid \mathtt{bool} \mid \mathtt{int} \mid \mathtt{string} \mid \ldots$	
型	τ	::=	b	基本型
		$\mid$	$\tau^{(0)}$	0-引数型
		$\mid$	$\tau + \tau$	直和

```
 1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"xmlns:dfdl="http://www.ogf.org/dfdl/dfdl-1.0/"
 2   elementFormDefault="unqualified" attributeFormDefault="unqualified">
 3   <xs:annotation>
 4     <!-- Defaulted property values -->
 5     <xs:appinfo source="http://www.ogf.org/dfdl/">
 6       <dfdl:format representation="text" initiator="" terminator="" separator="" leadingSkip="0"
 7          trailingSkip="0" textBidi="no" floating="no" fillByte="1" escapeSchemeRef=""
 8          textNumberRounding="explicit" textNumberRoundingIncrement="1.0" decimalSigned="yes"
 9          alignmentUnits="bytes" alignment="1" encoding="utf-8" ignoreCase="no" occursCountKind="implicit"
10          byteOrder="bigEndian" lengthKind="delimited" lengthUnits="bytes" textPadKind="none" textTrimKind="none"
11          textNumberPattern="###" textNumberJustification="right" textStringJustification="left"
12          textStandardBase="10" textNumberRep="standard"
13          textStandardGroupingSeparator="," textStandardDecimalSeparator="." textStandardExponentCharacter="e"
14          textNumberCheckPolicy="lax" textStandardInfinityRep="~" textStandardNaNRep="z"
15          textNumberRoundingMode="roundHalfDown" textStandardZeroRep="" textZonedSignStyle="asciiStandard"
16          initiatedContent="no" sequenceKind="ordered" truncateSpecifiedLengthString="no"
17          />
18     </xs:appinfo>
19   </xs:annotation>
20   <xs:element name="Root">
21     <xs:complexType>
22       <xs:sequence>
23         <xs:element ref="Item"  minOccurs="0" maxOccurs="unbounded" />
24       </xs:sequence>
25     </xs:complexType>
26   </xs:element>
27   <xs:element name="Item">
28     <xs:complexType>
29       <xs:sequence dfdl:sequenceKind="ordered">
30         <xs:element name="Length" type="xs:integer" dfdl:lengthKind="explicit" dfdl:length="1"/>
31         <xs:element name="Value" type="xs:string" dfdl:lengthKind="explicit" dfdl:length="{ ../Length }"/>
32       </xs:sequence>
33     </xs:complexType>
34   </xs:element>
35 </xs:schema>
```

図 1　DFDL スキーマの例
Fig. 1　Example of DFDL schema.

$\Sigma x:\tau.\tau$	依存和
$\tau\{p:=e\}$	プロパティ注釈

可変長文字列の例を処理するための型，すなわち図 1 の DFDL スキーマに対応する，DFDL0 の型の定義は以下のように与えられる．

$$\mu\alpha^{(0)}.\Sigma x:\mathtt{int}\{\mathtt{length}:=1\}.\ \Sigma\_:\mathtt{string}\{\mathtt{length}:=x\}.\ (\alpha^{(0)}+\mathtt{unit})$$

ここで $\Sigma x:\tau_1.\tau_2$ は依存対，すなわち対 (v_1,v_2) であって $v_1:\tau_1$，$v_2:\tau_2[x/v_1]$ と型つけできるものの型を表す．また，$\mathtt{length}:=1$ の部分は，length プロパティに関する注釈である．注釈の右辺には，あるホスト言語の任意の式 e を書くことができる．

unit はパース，アンパースにおいて空文字列に対応する．DFDL0 では $\tau+\mathtt{unit}$ を XML スキーマにおけるオプショナルな型を表すのに使う．直和型は可換ではないことに注意する．パース時に左辺 τ から優先的にパースを行い，バックトラックによって右辺 unit のパースが行われるためである．

再帰については，ホスト言語の式を呼び出しのパラメタにとれるようにするため，**図 2** で定義されるような n 引数の型とそのうえでの再帰を型の構文に加える[*1]．ただし，本論文で一般に型といった場合にはデータを表現す

[*1] Fisher らは同等のことを，カインド規則によって記述している．本論文では，構文的な定義で十分と判断した．また，Fisher らは $\Pi x:*.\tau$ のかわりに $\lambda x.\tau$ を用いていたが，本論文では値を引数に取る型であるため，これを依存積として定義した．

n-引数型	$\tau^{(n)}$	::=	τ	型，ただし $n = 0$
		|	$\Pi x : *.\tau^{(n-1)}$	依存積
		|	$\tau^{(n+1)} e$	適用
		|	$\alpha^{(n)}$	型変数
		|	$\mu\alpha^{(n)}.\tau^{(n)}$	再帰型

図 2　n-引数型（$n \geq 0$）
Fig. 2　n-ary types ($n \geq 0$).

定数	c	::=	()	`unit` 型定数
		|	`true` | `false`	`bool` 型定数
		|	0 | 1 | ...	`int` 型定数
		|	`"a"` | ...	`string` 型定数
		|	...	
式	e	::=	c	定数
		|	x	変数
		|	e `of` `inl` $x \Rightarrow e$ | `inr` $x \Rightarrow e$	パターンマッチ
		|	$e < e$	不等号
		|	$e = e$	等号
		|	$e + e$	和
		|	$e - e$	差
		|	`len` e	文字列長
		|	...	

図 3　ホストプログラム言語の例
Fig. 3　Example of host programming language.

る 0 引数の型を指す．また，図 2 の定義は再帰を簡潔に項で表現するためのものであるが，読みやすさのために $A = \mu\alpha^{(1)}.\Pi x : *.\tau$ を $Ax = \tau[\alpha^{(1)}/A]$ などのようにも書けるものとする．

最後に式 e から参照できるホスト言語について，**図 3** に例をあげる．本論文ではこの言語は，最小限の単純な一階の言語となっており，一般的な意味が与えられているものとする．ホスト言語として，静的に型つけされた言語を用い，DFDL0 が表す構文木の型との整合性をあわせて検査することも考えられ，Fisher らはこの議論を行っている．しかし，本論文ではこの問題は扱わない．DFDL 自体のホスト言語は XPath 言語であり，これは動的に型つけされている．

3.2　DFDL0 の値

DFDL0 の型は，データの値自体の型を表現すると同時に，値の内部表現と，バイト列としての外部表現の間のパースまたはアンパースの手法を定義する．

値に関しては，DFDL0 の型は以下のような基本型，対および `inl`，`inr` 構築子からなる一階の値を型つけしている．

値	v	::=	c
		|	(v, v)
		|	`inl` v
		|	`inr` v

入力	みている型の部分式	環境	出力
1A2AB	Σx : `int`{`length` := 1}....	$x \mapsto 1$	(1, ...)
1A2AB	$\Sigma\_$: `string`{`length` := x}....	$x \mapsto 1$	(1, "A", ...)
1A2AB	$\alpha^{(0)} +$ `unit`	$x \mapsto 1$	(1, "A", `inl` ...)
1A2AB	Σx : `int`{`length` := 1}....	$x \mapsto 2$	(1, "A", `inl`(2, ...))
1A2AB	$\Sigma\_$: `string`{`length` := x}....	$x \mapsto 2$	(1, "A", `inl`(2, "AB", ...))
1A2AB	$\alpha^{(0)} +$ `unit`	$x \mapsto 2$	(1, "A", `inl`(2, "AB", `inr`()))

図 4　入力 `1A2AB` のパース
Fig. 4　Parsing `1A2AB`.

冒頭の XML に対応する `1A2AB3ABC` のパース結果は，

```
(1, "A",
  inl(2, "AB",
    inl(3, "ABC", inr())))
```

となる．**図 4** にこのパースの挙動を示す．型の Σx : `int`{`length` := 1}.... の部分式は，続く 1 文字を `int` 型の値として読みこみ，得られた値は出力対の左辺になると同時に，変数 x に束縛される．$\Sigma\_$: `string`{`length` := x}.... の部分は続く x 文字を `string` 型の値として読み込む．直和 $\alpha^{(0)} +$ `unit` のところでは，可能な限り左辺が処理され再帰的に入力が読まれるが，入力の終端では右辺が処理される．

アンパースについては，図 4 の入出力が入れ替わった形の処理が行われる．すなわち構文木が前順序で走査され，葉に存在するデータが左から右へ直列化され，テキストが出力される．

3.3　DFDL プロパティに基づいたパーサの制御

DFDL プロパティはパーサ，アンパーサの挙動を指定する．本節ではパーサの制御に重点を置いて説明する．

プロパティ	p	::=	`length`
		|	`lengthPattern`
		|	`initiator`
		|	`terminator`
		|	`separator`
		|	`discriminator` x

`length` プロパティは，データのバイト列表現のバイト数を指定する．`lengthPattern` プロパティは，データのバイト列表現の満たす正規表現を記述する[*2]．

[*2] 実際の DFDL においては，このほかに `lengthKind` プロパティがあり，この値を `length` プロパティを指定する場合には，`"explicit"` に，`lengthPattern` プロパティを指定する場合には，`"pattern"` に設定しなければならない．それ以外の場合は `lengthKind` は `"delimited"` が指定されているものとする．

プロパティ注釈 $\tau\{p := e\}$ の e を評価した右辺値にはプロパティに応じて期待されている型があり，`length` ならば `int` 値，`discriminator` ならば `bool` 値，それ以外のプロパティには `string` 値が期待されているものとする．処理時に型が異なる場合，あるいは，値が想定する範囲にない場合（たとえば `length` が負である場合）には処理は失敗する．

DFDL0 は LL(k) 文法を自然に表現できる．たとえば

$A \to M(\texttt{"+"}\ A \mid \texttt{""})$
$M \to P(\texttt{"*"}\ M \mid \texttt{""})$
$P \to \texttt{"("}\ A\ \texttt{")"} \mid \mathit{IntLiteral}$

という文法は

$A = \Sigma\_ : M.(A\{\texttt{initiator} := \texttt{"+"}\} + \texttt{unit})$
$M = \Sigma\_ : P.(M\{\texttt{initiator} := \texttt{"*"}\} + \texttt{unit})$
$P = A\{\texttt{initiator} := \texttt{"("}\}\{\texttt{terminator} := \texttt{")"}\}$
$\quad + \texttt{int}\{\texttt{lengthPattern} := \texttt{"[0-9]+"}\}$

のように表現できる．ここで $\tau\{\texttt{initiator} := e\}$ と $\tau\{\texttt{terminator} := e\}$ は，τ のそれぞれ前後に現れる区切り文字列を指定する[*3]．

再帰を用いて型 τ のリストは $\mu\alpha.(\Sigma\_ : \tau.\alpha) + \texttt{unit}$ という型で表現できるので，対応するリスト値について $[v_1, v_2, \ldots, v_n]$ と略記した場合は，$\texttt{inl}(v_1, \ldots \texttt{inl}(v_n, \texttt{inr}())\ldots)$ を意味するものとする．たとえば `"(1+2)*3"` のパース結果は $((\texttt{[}(\texttt{inr}\,1, \texttt{[]}), (\texttt{inr}\,2, \texttt{[]})\texttt{]}, \texttt{[}\,\texttt{inr}\,3\,\texttt{]}), \texttt{[]})$ のように書ける．

`separator` プロパティは，やはり区切り文字列を指定するが，その注釈している部分式内に出現する Σ による対の区切りを指定する．上の例と同等な文法を以下のようにも書くことができる．

$A = (\mu\alpha.(\Sigma\_ : M.\alpha) + \texttt{unit})\{\texttt{separator} := \texttt{"+"}\}$
$M = (\mu\alpha.(\Sigma\_ : P.\alpha) + \texttt{unit})\{\texttt{separator} := \texttt{"*"}\}$
$P = A\{\texttt{initiator} := \texttt{"("}\}\{\texttt{terminator} := \texttt{")"}\}$
$\quad + \texttt{int}\{\texttt{lengthPattern} := \texttt{"[0-9]+"}\}$

ただし，この文法は `"1+2"` に加えて `"1+2+"` のような入力もパースできてしまう．`unit` を処理する際には，`separator` が存在してもしなくてもいいためである．これは，実際の DFDL のオプショナルな型における挙動と対応している．

3.4 `discriminator` とバックトラックによるパース

Haskell や Python などの言語はオフサイドルールなどと呼ばれる，インデントに依存した文法を用いている．このような言語も文脈自由文法では表現できないことが知られている [1]．

[*3] DFDL0 では指定できるのは単一の文字列としたが，実際の DFDL はこれらに複数の候補を指定することもできる．

```
if a:
  if b:
    c()
    d()
  e()
```

たとえば，上のような Python プログラムを考えると，同インデントの `if b:` と `e()`，あるいは `c()` と `d()` は同じブロック内に位置する文であるが，インデントが深くなっている部分は，先行する文に従属するブロックに位置する．すなわち，`if b:` 以降の文は `if a:` に従属するブロックに位置し，`c()` と `d()` は `if b:` に従属するブロック内の文となる．

DFDL0 でこの文法を表現すると，たとえば以下のようになる．

$\mathit{Prog} = \mathit{Stmts}\ 0$
$\mathit{Stmts}\ n = \Sigma \mathit{stmt} : \mathit{Stmt}\ n.(\mathit{Stmts}\ n) + \texttt{unit}$
$\mathit{Stmt}\ n =$
$\quad \Sigma \mathit{skip} : (\texttt{string}\{\texttt{lengthPattern} := \texttt{"\textbackslash s+"}\}$
$\qquad \{\texttt{initiator} := \texttt{"\textbackslash n"}\}$
$\qquad + \texttt{unit})$
$\qquad \{\texttt{discriminator}\ \mathit{skip} := \mathit{skip}\ \texttt{of}\ \texttt{inr}\ \_ \Rightarrow \texttt{true}$
$\qquad \mid \texttt{inl}\ \mathit{indent} \Rightarrow \texttt{len}\ \mathit{indent} = n\}.$
$\quad \Sigma \mathit{stmt} : \texttt{string}\{\texttt{lengthPattern} := \texttt{"[a-zA-Z][\^{}\textbackslash n]+"}\}.$
$\quad (\mathit{Block}\ n + \texttt{unit})$
$\mathit{Block}\ n =$
$\quad \Sigma \mathit{indent} : \texttt{string}\{\texttt{lengthPattern} := \texttt{"\textbackslash s+"}\}$
$\qquad \{\texttt{initiator} := \texttt{"\textbackslash n"}\}$
$\qquad \{\texttt{discriminator}\ \mathit{indent} := \texttt{len}\ \mathit{indent} > n\}.$
$\quad \mathit{Stmts}(\texttt{len}\ \mathit{indent})$

ここで用いられている，`discriminator` プロパティは，最も最近に直和 $\tau_1 + \tau_2$ の左辺 τ_1 を選択した箇所で，その選択が満たすべきアサーションのチェックを行うものである．もし，アサーションが失敗すれば，左辺 τ_1 は諦められ，右辺 τ_2 がかわりに選択される．またこのプロパティは，一般に $\tau\{\texttt{discriminator}\ x := e\}$ の形で与えられる，すなわち変数 x にパース後の τ 型の値がバインドされ，その値に依存する `bool` 型のアサーション e が記述される．

上の *Prog* 型は文法を以下のように解釈して，定義されている．

- *Stmts* n は n 文字のインデントからなる文 *Stmt* n の列である．ただし，最初の文のインデントはすでに読み込まれているものとする．
- *Stmt* n は最初の文の場合は，空白はもう読めないので，文の内容を読み込み *stmt* にバインドする．
- *Stmt* n は最初の文でない場合は，まず改行コード `"\n"` を飛ばしてから，空白の列 `"\s+"` を読み込み *skip* にバインドする．もし，この空白の数が n に満たなかった場合は，`discriminator` がこれをチェックし，バッ

クトラックによって外側の *Stmts n* の `unit` の側を処理する．すなわち，現在のブロックを閉じる処理をする．空白の数が n ならば，やはり文の内容を読み込み *stmt* にバインドする．

- 上記のどちらのケースでも *Stmt n* は文を読みこんだ後で，次の行からこの文に従属する新たなブロックが始まるかどうかを，*Block n* でチェックする．
- *Block n* は，やはり改行コードと空白の列を読み込み，この空白の列の長さ `len` *indent* が n を超えるかどうかを調べる．もし，そうでなければ，`discriminator` によってバックトラックが起き，新しいブロックは存在しなかったものとして，外側の *Stmt n* の `unit` の側が処理される．
- もし `len` *indent* が n を超える場合は，再帰的に *Stmts*(`len` *indent*) が新しいブロックの文の列を処理する．

この *Prog* 型に基づくと，上記の Python プログラムは以下のようにパースされる．

```
(([],"if a:",
  ("  ",
    ([],"if b:",
      ("      ",
        ([],"c()",[]),
        [(inr"      ","d()",[])])),
    [(inr"  ","e()",[])])),
  [])
```

3.5　DFDL と DFDL0 の違い

DFDL0 は DFDL の簡易なモデルであるが，データ形式が直積（依存和）と直和で表現される木か，XML か，という以外にもいくつかの違いがある．

大きな違いとして，DFDL の現行の仕様（DFDL1.0）では，木の深くなる方向の再帰は取り扱えず，横方向の列が取り扱えるだけである．しかし，これは現在もコミュニティで議論されている点であり，将来的には一般の再帰が扱えるようになる可能性がある．また DFDL では依存型による値の参照のかわりに，XPath 式でパース中の XML 木内のデータを参照する仕組みを用いている．図 1 で，可変長文字列の長さを参照するのに `dfdl:length="{ ../Length }"` を用いているのがそれである．機能としては同等のことが実現できるので，DFDL0 では依存和型とそれによって束縛された変数による参照を採用した．DFDL ではホスト言語としても，XPath 仕様で定義されている演算あるいは組み込み関数が使える．

また本論文の DFDL0 では，テキストデータの処理に重点を置いたが，DFDL はほかにバイナリ表現をパースするための各種の機能をそなえている．たとえば `representation` プロパティを使って，数値型などがテキスト表現であるか，バイナリ表現であるかを指定できる．また，アラインメントなど，バイナリ表現特有のデータ境界の処理のための多くのプロパティがある．

最後に，微妙な違いとして，DFDL では XML 属性によってプロパティを指定するので，たとえば同じプロパティを同一箇所に，2 つ指定するなどということはできない．また，違うプロパティであっても，評価の順序が定まっており，`length` プロパティは `initiator` や `terminator` を除いた文字列に対して後から解釈される．DFDL0 では矛盾する `length` を同一箇所に指定すればパースは必ず失敗する．また，外側のプロパティ（$\tau\{p_2 := e_2\}\{p_1 := e_1\}$ ならば p_1）が先に解釈される．

4.　パーサ，アンパーサの意味論

4.1　パーサの意味論

DFDL のパーサは再帰下降パーサ（recursive descent parser）である．このパーサは左側から構文木を生成していく．パーサの意味論は以下の形の述語で与えられる．

$$\omega, S, T \vdash \tau \Downarrow r$$

ここで，ω は入力バイト列，S，T は後述する区切りルールで用いられる区切り文字列の集合，τ は DFDL0 の型，そして，r は出力であり，以下の形をしている．

$$r ::= \texttt{fail} \mid v, \omega$$

すなわち，パースが失敗すれば結果として `fail` が返り，パースが成功した場合，結果のパース木 v，およびいまだパースされていない入力列の剰余 ω の組が返る．

図 5 にパーサの定義を示す．順にみていく．

基本型の規則に出現する $\texttt{parse}_b$ はバイト列から，b 型の値への部分関数である．また，文字列 ω について，$|\omega|$ はその長さ，$\omega[n, m]$ は n 文字目から，$m-1$ 文字までの部分文字列であり，`@` は結合である．基本型のパースのルールの注意点としては，バイト列のどの部分をパースするかに関する区切り（delimiter）ルールがある．ルールは以下のようなものである．

- もし，直近で `length` あるいは `lengthPattern` の注釈があれば，現在位置から指定された位置までのバイト列をパースして，値を得る．
- もし，直近に `separator` あるいは `terminator` の注釈があれば，バイト列内にそのいずれかの最初の出現を見つけ，これを区切りとして現在位置から区切りまでのバイト列をパースして，値を得る[*4]．

というものである．基本型のパースが失敗する条件は，バ

[*4] 区切り文字列は，通常の構文解析の FOLLOW 集合などに似ているが異なるものである．`initiator` は決して区切り文字列とはみなされないし，また，まだ後続データを読む必要があり，`terminator` は期待されていない場所で，`terminator` で入力を区切ってしまうことも起こる．

$$\boxed{\omega, S, T \vdash \tau \Downarrow r}$$

基本型のパース規則

$$\frac{n = min(\{|\omega|\} \cup \{n \mid (\exists \zeta \in S \cup T)\omega[n, n+|\zeta|] = \zeta\}) \quad \mathtt{parse}_b(\omega[0,n]) = v \quad \text{(P1)}}{\omega, S, T \vdash b \Downarrow v, \omega[n, |\omega|]} \qquad \frac{\text{(P1) で } \omega[0,n] \text{ は } b \text{ でパースできない}}{\omega, S, T \vdash b \Downarrow \mathtt{fail}}$$

プロパティ注釈のパース規則

$$\frac{e \Downarrow \zeta : \mathtt{string} \quad \omega[0,|\zeta|] = \zeta \quad \omega[|\zeta|,|\omega|], S, T \vdash \tau \Downarrow v, \omega' \quad \text{(P2)}}{\omega, S, T \vdash \tau\{\mathtt{initiator} := e\} \Downarrow v, \omega'} \qquad \frac{\text{(P2) は満されない}}{\omega, S, T \vdash \tau\{\mathtt{initiator} := e\} \Downarrow \mathtt{fail}}$$

$$\frac{e \Downarrow \zeta : \mathtt{string} \quad \omega, S, \{\zeta\} \vdash \tau \Downarrow v', \omega' \quad \omega'[0,|\zeta|] = \zeta \quad \text{(P3)}}{\omega, S, T \vdash \tau\{\mathtt{terminator} := e\} \Downarrow v', \omega'[|\zeta|, |\omega'|]} \qquad \frac{\text{(P3) は満たされない}}{\omega, S, T \vdash \tau\{\mathtt{terminator} := e\} \Downarrow \mathtt{fail}}$$

$$\frac{e \Downarrow \zeta : \mathtt{string} \quad \omega, \{\zeta\}, T \vdash \tau \Downarrow r}{\omega, S, T \vdash \tau\{\mathtt{separator} := e\} \Downarrow r}$$

$$\frac{e \Downarrow n : \mathtt{int} \quad \omega[0,n], \{\}, \{\} \vdash \tau \Downarrow v, \mathtt{""} \quad \text{(P4)}}{\omega, S, T \vdash \tau\{\mathtt{length} := e\} \Downarrow v, \omega[n, |\omega|]} \qquad \frac{\text{(P4) は満たされない}}{\omega, S, T \vdash \tau\{\mathtt{length} := e\} \Downarrow \mathtt{fail}}$$

$$\frac{e \Downarrow \rho : \mathtt{string} \quad \rho \text{ の } \omega \text{ への最長マッチが } \omega[0,n] \text{ である} \quad \omega[0,n], \{\}, \{\} \vdash \tau \Downarrow v, \mathtt{""} \quad \text{(P5)}}{\omega, S, T \vdash \tau\{\mathtt{lengthPattern} := e\} \Downarrow v, \omega[n, |\omega|]} \qquad \frac{\text{(P5) は満たされない}}{\omega, S, T \vdash \tau\{\mathtt{lengthPattern} := e\} \Downarrow \mathtt{fail}}$$

$$\frac{\omega, S, T \vdash \tau \Downarrow v, \omega' \quad e[x/v] \Downarrow \mathtt{true} : \mathtt{bool} \quad \text{(P6)}}{\omega, S, T \vdash \tau\{\mathtt{discriminator}\, x := e\} \Downarrow v, \omega'} \qquad \frac{\text{(P6) は満たされない}}{\omega, S, T \vdash \tau\{\mathtt{discriminator}\, x := e\} \Downarrow \mathtt{fail}}$$

その他の型構築子のパース規則

$$\frac{\omega, S, T \vdash \tau^{(n)}[\alpha^{(n)}/\mu\alpha^{(n)}.\tau^{(n)}]e_1 \cdots e_n \Downarrow r}{\omega, S, T \vdash (\mu\alpha^{(n)}.\tau^{(n)})e_1 \cdots e_n \Downarrow r} \qquad \frac{e_0 \Downarrow v_0 : \_ \quad \omega, S, T \vdash \tau^{(n)}[x/v_0]e_1 \cdots e_n \Downarrow r}{\omega, S, T \vdash (\Pi x : *.\tau^{(n)})e_0 \cdots e_n \Downarrow r}$$

$$\frac{\omega, \{\}, T \vdash \tau \Downarrow v, \omega' \quad \omega', \{\}, T \vdash \tau'[x/v] \Downarrow w, \omega'' \quad \text{(P7.1)}}{\omega, \{\}, T \vdash \Sigma x : \tau.\tau' \Downarrow (v, w), \omega''} \qquad \frac{\omega, \{\zeta\}, T \vdash \tau \Downarrow v, \omega' \quad \omega'[0,|\zeta|] = \zeta \quad \omega'[|\zeta|,|\omega'|], \{\zeta\}, T \vdash \tau'[x/v] \Downarrow w, \omega'' \quad \text{(P7.2)}}{\omega, \{\zeta\}, T \vdash \Sigma x : \tau.\tau' \Downarrow (v, w), \omega''}$$

$$\frac{\omega, \{\zeta\}, T \vdash \tau \Downarrow v, \omega' \quad \omega'[0,|\zeta|] \neq \zeta \quad \omega', \{\zeta\}, T \vdash \tau'[x/v] \Downarrow (), \omega'' \quad \text{(P7.3)}}{\omega, \{\zeta\}, T \vdash \Sigma x : \tau.\tau' \Downarrow (v, ()), \omega''} \qquad \frac{\text{(P7.1),(P7.2) および (P7.3) のいずれも満たさない}}{\omega, S, T \vdash \Sigma x : \tau.\tau' \Downarrow \mathtt{fail}}$$

$$\frac{\omega, S, T \vdash \tau \Downarrow v, \omega'}{\omega, S, T \vdash \tau + \tau' \Downarrow \mathtt{inl}\, v, \omega'} \qquad \frac{\omega, S, T \vdash \tau \Downarrow \mathtt{fail} \quad \omega, S, T \vdash \tau' \Downarrow v, \omega'}{\omega, S, T \vdash \tau + \tau' \Downarrow \mathtt{inr}\, v, \omega'} \qquad \frac{\omega, S, T \vdash \tau \Downarrow \mathtt{fail} \quad \omega, S, T \vdash \tau' \Downarrow \mathtt{fail}}{\omega, S, T \vdash \tau + \tau' \Downarrow \mathtt{fail}}$$

図 5　パースの意味論
Fig. 5　Parsing semantics.

イト列がパースできないことである．

次にプロパティ注釈に関する規則をみる．定義中，$e \Downarrow v : b$ は，ホスト言語の式 e が b 型の値 v に評価されることを表す．ルール中 separator 以外のパースが失敗する条件は，自明なものである．separator は，既述したように $\Sigma x : \tau.\tau$ 型で対を作るときに調べられる．よって，注釈が与えられた時点では S に区切り文字列を記憶しておくだけである．length あるいは lengthPattern のルールでは，これらの注釈があれば区切りルールはこちらが優先になるので，外側で定義された，S，T の内容は捨てられる．

最後に，それ以外の型構築子，特に $\Sigma x : \tau.\tau$ の規則であるが，(P7.1) (P7.2) はそれぞれ separator が指定されていない場合，いる場合のルールであり自然だが，(P7.3) は若干，特殊なルールである．DFDL ではオプショナルな型に対する値が空である場合に，separator が見つかれば separator のバイト列が消費されるが，見つからなくてもエラーにはならず，バイト列も消費されない．(P7.3) はこの挙動に対応している．

定義 1（パーサの意味論）　入力バイト列 ω の型 τ に基づくパースは

$$\omega, \{\}, \{\} \vdash \tau \Downarrow v, \mathtt{""}$$

のとき成功し，パース木は値 v で与えられる．

4.2　アンパーサの意味論

アンパーサの意味論はバックトラックを含むパーサの意味論と比べれば簡単なものになっている．また，いくつかのプロパティは無視される．

以下の述語を定義する．

$$S \vdash v : \tau \Downarrow r$$

直近の separator の定義 S が与えられたとき，述語は値 v が型 τ に基づいてアンパースされた場合，出力は r であることを示す．ただし，r は

$$r ::= \mathtt{fail} \mid \omega$$

すなわちアンパースが失敗する，または出力バイト列 ω を出して正常終了したことを示す．

図 6 にアンパーサの意味論を示す．

基本型のルールにおいて，$\mathtt{unparse}_b(v)$ は b 型の値 v をアンパースした結果の文字列を示す．また，プロパティ注釈のルールにおける，initiator，terminator，および $\Sigma x : \tau.\tau$ の規則における separator は適切な位置に区切り文字列を出力する．

$$\boxed{S \vdash v : \tau \Downarrow \omega}$$

基本型のアンパース規則

$$\frac{}{S \vdash v : b \Downarrow \mathtt{unparse}_b(v)}$$

プロパティ注釈のアンパース規則

$$\frac{e \Downarrow \zeta : \mathtt{string} \quad S \vdash v : \tau \Downarrow \omega \quad \text{(U1)}}{S \vdash v : \tau\{\mathtt{initiator} := e\} \Downarrow \zeta \mathbin{@} \omega} \qquad \frac{\text{(U1) は満たされない}}{S \vdash v : \tau\{\mathtt{initiator} := e\} \Downarrow \mathtt{fail}}$$

$$\frac{e \Downarrow \zeta : \mathtt{string} \quad S \vdash v : \tau \Downarrow \omega \quad \text{(U2)}}{S \vdash v : \tau\{\mathtt{terminator} := e\} \Downarrow \omega \mathbin{@} \zeta} \qquad \frac{\text{(U2) は満たされない}}{S \vdash v : \tau\{\mathtt{initiator} := e\} \Downarrow \mathtt{fail}}$$

$$\frac{e \Downarrow \zeta : \mathtt{string} \quad \{\zeta\} \vdash v : \tau \Downarrow r}{S \vdash v : \tau\{\mathtt{separator} := e\} \Downarrow r}$$

$$\frac{e \Downarrow n : \mathtt{int} \quad \{\} \vdash v : \tau \Downarrow \omega \quad |\omega| = n \quad \text{(U3)}}{S \vdash v : \tau\{\mathtt{length} := e\} \Downarrow \omega} \qquad \frac{\text{(U3) は満たされない}}{S \vdash v : \tau\{\mathtt{length} := e\} \Downarrow \mathtt{fail}}$$

$$\frac{S \vdash v : \tau \Downarrow r}{S \vdash v : \tau\{\mathtt{lengthPattern} := e\} \Downarrow r} \qquad \frac{S \vdash v : \tau \Downarrow r}{S \vdash v : \tau\{\mathtt{discriminator}\, x := e\} \Downarrow r}$$

その他の型構築子のアンパース規則

$$\frac{S \vdash v : \tau^{(n)}[\alpha^{(n)}/\mu\alpha^{(n)}.\tau^{(n)}]e_1 \cdots e_n \Downarrow \omega}{S \vdash v : (\mu\alpha^{(n)}.\tau^{(n)})e_1 \cdots e_n \Downarrow \omega} \qquad \frac{e_0 \Downarrow v_0 : \_ \quad S \vdash v : \tau^{(n)}[x/v_0]e_1 \cdots e_n \Downarrow \omega}{S \vdash v : (\Pi x : *.\tau^{(n)})e_0 \cdots e_n \Downarrow \omega}$$

$$\frac{\{\} \vdash v : \tau \Downarrow \omega \quad \{\} \vdash w : \tau'[x/v] \Downarrow \omega' \quad \text{(U4.1)}}{\{\} \vdash (v, w) : \Sigma x : \tau.\tau' \Downarrow \omega \mathbin{@} \omega'} \qquad \frac{S \vdash v : \tau \Downarrow \omega \quad \text{(U4.2)}}{S \vdash (v, ()) : \Sigma x : \tau.\tau' \Downarrow \omega}$$

$$\frac{\{\zeta\} \vdash v : \tau \Downarrow \omega \quad w \neq () \quad \{\zeta\} \vdash w : \tau'[x/v] \Downarrow \omega' \quad \text{(U4.3)}}{\{\zeta\} \vdash (v, w) : \Sigma x : \tau.\tau' \Downarrow \omega \mathbin{@} \zeta \mathbin{@} \omega'} \qquad \frac{\text{(U4.1),(U4.2) および (U4.3) のいずれも満たさない}}{S \vdash (v, w) : \Sigma x : \tau.\tau' \Downarrow \mathtt{fail}}$$

$$\frac{S \vdash v : \tau \Downarrow w}{S \vdash \mathtt{inl}\, v : \tau + \tau' \Downarrow w} \qquad \frac{S \vdash v : \tau' \Downarrow w}{S \vdash \mathtt{inr}\, v : \tau + \tau' \Downarrow w}$$

図 6　アンパースの意味論
Fig. 6　Unparsing semantics.

アンパース時に lengthPattern および discirminator のチェックは行われない．length 制約については，長さが正しく指定した長さであることがチェックされる．この length に対する挙動は，実際の DFDL ではパッド文字（textStringPadCharacter, textNumberPadCharacter など）を指定することで切り替えることができる．本論文ではこれらのプロパティは省略した．

定義 2（アンパーサの意味論）　値 v の型 τ に基づくパースは

$$\{\} \vdash v : \tau \Downarrow \omega$$

のとき成功し，結果のバイト列は ω で与えられる．

5. 議論

5.1 停止性

DFDL0 では LL(k) では表現できない左再帰的な定義，たとえば

$$(\mu\alpha.(\Sigma\_ : \alpha.\,\mathtt{int}) + \mathtt{int})\{\mathtt{separator} := \texttt{"-"}\}$$

を用いたパースは一般に停止しない．

いま，nilable 集合 N を以下を満たす最小の集合として定義する．

- $\mathtt{unit} \in N$ あるいは $\mathtt{string} \in N$.
- $\tau_1 \in N$ または $\tau_2 \in N$ ならば $\tau_1 + \tau_2 \in N$.
- $\tau_1 \in N$ かつ $\tau_2 \in N$ ならば $\Sigma x : \tau_1.\tau_2 \in N$.
- $\tau \in N$ ならば $\tau e \in N$.
- $\tau \in N$ ならば $\Pi x : *.\tau \in N$.
- $\tau[\alpha/\mu\alpha.\tau] \in N$ ならば $\mu\alpha.\tau \in N$.
- $\tau \in N$ ならば，$\tau\{p := e\} \in N$．ただし，$p :\!= e$ が静的に空文字列を受理しないと（たとえば $\mathtt{length} :\!= n$ $(n \geq 1)$）判定できる場合は除く．

同様に Fischer-Ladner 閉包 $C(\tau)$ と ϵ 閉包 $C_\epsilon(\tau)$ を

- $\Pi x : *.\tau_1 e \in C(\tau) \cup \{\tau\}$ ならば $\tau_1 \in C(\tau)$．$C_\epsilon(\tau)$ についても同様．
- $\tau_1 e \in C(\tau) \cup \{\tau\}$ ならば $\tau_1 \in C(\tau)$．$C_\epsilon(\tau)$ についても同様．
- $\mu\alpha.\tau_1 \in C(\tau) \cup \{\tau\}$ ならば $\tau_1[\alpha/\mu\alpha.\tau_1] \in C(\tau)$．$C_\epsilon(\tau)$ についても同様．
- $\tau_1 + \tau_2 \in C(\tau) \cup \{\tau\}$ ならば $\tau_1, \tau_2 \in C(\tau)$．$C_\epsilon(\tau)$ についても同様．
- $\Sigma x : \tau_1.\tau_2 \in C(\tau) \cup \{\tau\}$ ならば $\tau_1, \tau_2 \in C(\tau)$.
- $\Sigma x : \tau_1.\tau_2 \in C_\epsilon(\tau) \cup \{\tau\}$ ならば $\tau_1 \in C_\epsilon(\tau)$．かつ，$\tau_1 \in N$ ならば $\tau_2 \in C_\epsilon(\tau)$.
- $\tau_1\{p := e\} \in C(\tau) \cup \{\tau\}$ ならば $\tau_1 \in C(\tau)$.
- $\tau_1\{p := e\} \in C_\epsilon(\tau) \cup \{\tau\}$ かつ $p :\!= e$ が $\mathtt{initiator} :\!= \zeta$

（$|\zeta| \geq 1$）でないならば $\tau_1 \in C_\epsilon(\tau)$.

を満たす，それぞれ最小の集合として定義する．それぞれ直感的には N は空文字列をパースしうる型の集合，$C(\tau)$ は τ によるパースの際に図 5 の述語に出現しうる型（または適用である場合はその型の左辺の型）の集合，$C_\epsilon(\tau)$ は，τ でパースしたときに入力を消費せずに到達する型の集合（$C(\tau)$ の部分集合）である．

命題 1　任意の $\tau' \in C(\tau)$ について，$\tau' \notin C_\epsilon(\tau')$ ならば τ に基づくパースは必ず停止する．

証明は，命題の条件が満たされれば，各 $\tau' \in C(\tau)$ に関して，それがパースの規則に再帰的に出現するときに，必ず，入力バイト列中の現在位置が前に進んでいることを示せばよく，容易である．

すでに述べたように，DFDL で再帰を取り扱えるようにするかという議論が行われているが，再帰を取り扱う場合には，停止性の検査は重要になってくる．

5.2　今後の課題

言語に形式的な意味論を与えることは，言語仕様の理解に役に立つと同時に，各種の整合性のチェック，あるいはコンパイラ，最適化の設計などに役にたつ．本論文の DFDL0 はまだ完全なものにはほど遠いが，DFDL 技術への貢献に向けていくつか方向性がある．

- 木正規型への拡張．本論文の DFDL0 のベースは直和，直積からなる，抽象データ型であるが．DFDL で扱っている XML スキーマのモデルとしては，直和のかわりに，集合和を持ちいた木正規型を使うこともできる．
- round-trip 性．アンパースとパースの合成が恒等関数になるという性質である．DFDL の用途から考えて，この性質があることは望ましい．ただし，この検査のためには，たとえば `inr` 値をアンパースした文字列に対して，必ず直和型の左辺によるパースが `fail` することなどを示さなければならず，容易ではない．
- DFDL0 は簡易なモデルであるため，DFDL 処理系における中間言語としての利用が考えられる．応用として，バックトラック，パターンマッチなどの最適化を通じた処理系の高速化が考えられる．

6.　まとめ

近年，重要になってきているデータ記述言語 DFDL の依存型に基づいた形式的なモデル DFDL0 を定義し，そのパーサ，アンパーサとしての意味論を与えた．再帰データ型を記述する一般的な体系のうえに DFDL0 を構築したことで，DFDL の要点を取り出した簡易なモデルを作ることができた．また，このモデルのうえで，DFDL0 パーサの停止性の静的な検査の議論も行った．

参考文献

[1] Adams, M.D.: Principled parsing for indentation-sensitive languages: Revisiting landin's offside rule, *40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13*, Rome, Italy - January 23-25, pp.511–522 (2013).

[2] Birman, A.: The Tmg Recognition Schema, PhD Thesis, Princeton University, Princeton, NJ, USA (1970).

[3] Fisher, K., Mandelbaum, Y. and Walker, D.: The next 700 data description languages, *Proc. 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006*, Charleston, South Carolina, USA, January 11-13, pp.2–15 (2006).

[4] 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, pp.111–122, ACM (online), DOI: 10.1145/964001.964011 (2004).

[5] Health Level Seven International: HL7, Health Level Seven International, available from ⟨http://www.hl7.org/⟩ (accessed 2015).

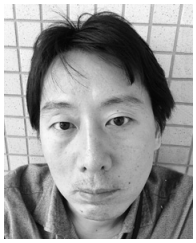
[6] ISO/TC 68/SC 7 Core banking: ISO 8583-1:2003, Financial transaction card originated messages – Interchange message specifications – Part 1: Messages, data elements and code values, Internatioal Orgnization for Standardization, available from ⟨http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=31628⟩ (accessed 2015).

[7] Leijen, D. and Meijer, E.: Parsec: Direct Style Monadic Parser Combinators For The Real World, Technical Report (2001).

[8] OGF DFDL Working Group: Data Format Description Language 1.0, Open Grid Forum, available from ⟨http://www.ogf.org/dfdl⟩ (accessed 2015).

[9] UN/CEFACT (United Nations Centre for Trade Facilitation and Electronic Business): EDIFACT, United Nations, available from ⟨http://www.unece.org/cefact/⟩ (accessed 2015).

[10] W3C XML Schema Working Group: XML Schema 1.0, World Wide Web Consortium, available from ⟨http://www.w3c.org/XMLSchema/2001⟩ (accessed 2015).



戸澤 晶彦

2000 年東京大学理学系研究科，修士課程修了．同年日本アイ・ビー・エム（株）入社．以来，東京基礎研究所研究員として XML 処理，プログラミング言語の研究に従事．2014 年より東京大学情報理工学系研究科，社会人博士課程．日本ソフトウェア科学会会員．

佐藤 直人

2001 年日本アイ・ビー・エム（株），東京基礎研究所研究員．ACM 会員．1997 年東京大学理学系研究科博士課程修了．博士（理学）．

河内谷 清久仁　（正会員）

1963 年生．1987 年東京大学大学院理学系研究科情報科学専門課程修士課程修了．同年日本アイ・ビー・エム（株）入社．以来，同社東京基礎研究所にて，オペレーティングシステムやマルチメディア処理システム，携帯情報システム，Java 処理系ランタイム，並列・分散プログラミング言語 X10 等の研究に従事．現在，同研究所ディープ・コンピューティング&アナリティクス担当部長．1994 年大会奨励賞，2005 年論文賞，2008 年日本ソフトウェア科学会高橋奨励賞受賞．博士（政策・メディア）．ACM シニアメンバー，日本ソフトウェア科学会会員．