文脈によるプログラミング言語処理系による DSL 基盤の試作

森下敦司†

ContextWorkbench は，記号処理を文脈の機構で制御する点に特徴を持つプログラミング言語処理系である．文脈とは，データと関数の区別なく単語の意味を規定する概念であり機構である．本稿では，この ContextWorkbench を用いた DSL（Domain Specific Language）基盤の試作について報告する．この試作では，DSL として Smalltalk に似たスクリプト記述を可能にした．また，その記述と ContextWorkbench に固有な記述をシームレスに用いたプログラミングを可能にした．

The prototype of the mechanism for DSL based on Context-oriented Programming Environment

Atsushi Morishita†

ContextWorkbench is a programming environment for symbolic processing with the mechanism of Context. Context is a concept and a mechanism for connecting words in programs to implementations of not only variables but functions. I report the prototype development of the mechanism for DSL（Domian Specific Language）based on ContextWorkbench. The prototype enables programming with the scripting language similar to Smalltalk as DSL. And the mechanism enables seamless programming using both this scripting language and the native language of ContextWorkbench.

1. はじめに

ContextWorkbench[1]は筆者が開発した文脈によるプログラミング言語処理系である．本稿では，ContextWorkbench を用いた DSL[2]（Domain Specific Language）基盤の試作ついて報告する．

1.1 ContextWorkbench の特徴

筆者は，ContextWorkbench の次の特徴が DSL 基盤の実現に適していると考えている．

(1) 実行内容の置き換え

ContextWorkbench では，文脈機構を用いて実行時に文脈の指定や変更を行うことで，プログラムを構成する単語の意味を後付けで置き換え可能である．これにより，具体的な制御やデータ構造が不定な記述から始めて，徐々に記述を具体化することが出来る．また，単語の意味を Java の実装に置き換えることで，プログラムの実行環境を ContextWorkbench から実際のアーキテクチャに段階的に移行することが出来る．

(2) ドメインに固有な知識の整理

オブジェクト指向では，ドメインに固有な知識を整理するために，「抽象的で階層的な分類」と「組み合わせのバリエーションの整理」の両方を行う必要がある．なお，前者はクラスの継承構造のことであり，後者はクラス間の関連のことである．しかし，これらにはドメインに固有な知識と独立なスキルを必要とする．そのスキルとは，オブジェクト構造の良し悪しを判断するスキルである．そして，それはプログラミングやアーキテクチャ設計などの素養に基づくものである．なお，その事実はドメインに固有な記述に対するオブジェクト指向の適用において大きな障害になっている．一方，ContextWorkbench ではドメインに固有な記述を局所的に行い，後から文脈機構を用いて，全体的な記述と結びつけることが出来る．

1.2 文脈機構を用いたプログラミング

ContextWorkbench のプログラムでは，図 1 のように制御構造やデータ構造の記述と，それらを構成する関数や変数の実装が分離されている．そして，文脈機構を用いて，その結合を行う．

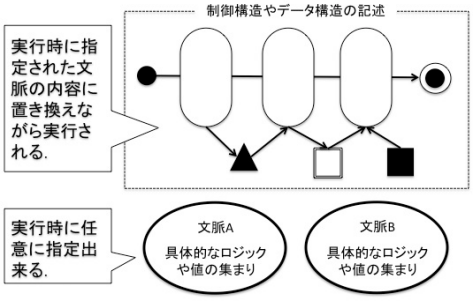

図 1 文脈機構

†株式会社 NTT データ
NTTDATA CORPORATION

次に文脈機構を用いたプログラムの例をプログラム 1 に示す.

```
00: do ( defun 'p '( in ) '[ ( return + x in ) ] )
01: do ( defctx 'A )
02: do ( ctx=A bind 'x 1 )
03: do ( defctx 'B )
04: do ( ctx=B bind 'x 100 )
05: chgctx A
06: do ( p 100 )
07: => 101
08: chgctx B
09: do ( p 100 )
10: => 200
```

プログラム 1　関数の定義と文脈の指定

プログラム 1 では, 00 行目において文脈を指定せずに“関数 p”を定義している．なお，この関数は，“入力値 in”を“x”と足し合わせて返す単純なものである．ただし，まだ“x”は未定義で良い点がポイントである. ContextWorkbench はプログラムの読み込み時に, 即座に文や単語の意味を判定するが, 文や単語に引用符を付与することで, 判定させないように出来る．そして，関数を定義する関数である“defun”は，その機構を利用することで未定な内容を記述可能にする．“関数 defun”とは，単語と関数の名前が一致した場合に，単語を関数のボディに当たる文（00 行目であれば “'[(return + x in)]”）に置き換えて解釈を行うことの指定である. そして, その解釈は, 実行時の文脈で行われる. これにより, 06 行目と 09 行目のように同じ記述であっても，実行時の文脈が違えば結果が異なってくる. 例えばこの場合，02 行目において“文脈 A”における“x”の意味を 1 に，そして 04 行目において“文脈 B”における“x”の意味を 100 に，それぞれ定義している．そして，05 行目では“文脈 A”で実行することを指定しているため，結果は 07 行目のように 1 と 100 の和になる．また，08 行目では“文脈 B”で実行することを指定しているため，結果は 10 行目のように 100 と 100 の和になる.

なお，実行時の文脈の指定には，02 行目や 04 行目のように括弧の内部で“ctx=文脈名”の形式で文脈を指定する方法と，05 行目や 08 行目のように“chgctx 文脈名”の形式で文脈を指定する方法がある. いずれの方法でも効果は同じであるが,“chgctx 文脈名”の形式を採用した方が，後に続くプログラムの記述に影響を与えないため，記述の汎用性が高まると考える.

```
00: do ( defctx 'C )
01: chgctx C
02: do ( ctx=C defun 'x '( in ) '[ ( return + 200 in) ] )
03: do ( bind 'D C )
04: chgctx D
05: do ( p 100 )
06: => 300
```

プログラム 2　間接的な文脈の指定

プログラム 2 では 00 行目で新たに“文脈 C”を定義し，03 行目で“関数 bind”を用いて，“単語 D”を“文脈 C”に束縛している．従って，04 行目における“文脈 D”の指定は，間接的に“文脈 C”を指定することになる．なお，プログラム 2 のポイントは文脈 C において“関数 x”を定義している点である．従って，05 行目の内容は，“関数 p”の中で 100 を入力にして“関数 x”を呼び出すものになるため，結果は 06 行目のように 200 と 100 の和になる．このように，ContextWorkbench では，データと関数の区別なく単語を束縛出来る.

1.3 試作の方針

試作の方針を次の(1)から(4)に示す.

(1) 構文規則の混在

DSL としての記述と ContextWorkbench に固有な記述が 1 つのファイルに混在可能であるようにする. ただし，1 つの文には単一の構文規則を適用することにする．つまり，変数宣言や関数の定義には何れか 1 つの構文規則が適用されるようにする.

(2) 記述間のシームレスな連携

DSL によるプログラムの実行結果と ContextWorkbench に固有な記述によるプログラムの実行結果をシームレスに連携する．シームレスであるとは，一方のプログラムで行われた定義や値の操作が，他方のプログラムにも反映されることを意味する.

(3) Smalltalk を模した記述

DSL の構文規則は Smalltalk[3]を模したものにする. その理由は, ContextWorkbench の対話的な実行環境が Smalltalk を模したものであり，実行環境との親和性が高いと考えるからである．また，単語に日本語を利用可能な ContextWorkbench の特徴を活かして，日本語化された Smalltalk 風のプログラミングを可能にすることも目的の 1 つとする.

(4) 開発方針

ContextWorkbench は Java のみで実装されている. 従って，DSL 基盤としての拡張機能も Java で実装する．なお，ContextWorkbench はインタプリタ方式でプログラムを実行する．DSL の実行機能は，このインタプリタにフックして DSL を実行するように

する．また，DSL としての記述を ContextWorkbench に固有な記述に自動的に変換して実行するようにする．なお，文脈木の状態を反映した構文解析を行い易くするために，既存のコンパイラコンパイラを用いず，構文解析機能等も自作する．

2. 試作した DSL の利用方法と記述例

ContextWorkbench 上に実装された DSL の利用例をプログラム 3 に示す。

```
00: do ( defctx 'obj )
01: chgctx obj
02: do ( defun 'x '( p ) '[ ( return p) ] )
03: DSL としての記述を開始する．
04: chglang smalltalk
05: a := obj x: #( 1 2 3 ).
06: @break
07: chgctx SmalltalkWorld
08: DSL で設定された値を印字する．
09: do ( printcr a )
10: => '( 1 2 3 )
```

プログラム 3　DSL の利用例

プログラム 3 のうち，04 行目が DSL として記述の開始の指定であり，06 行目が終了の指定である．従って，05 行目が DSL による記述である．プログラム 3 の詳細な説明を次の(1)から(6)に示す．

(1) **00 行目から 02 行目**

この記述は，ContextWorkbench に固有な記述である．まず，00 行目で“文脈 obj”を定義している．そして，01 行目においてプログラムを実行する文脈を“obj”に変更し，02 行目において“関数 x”を定義している．従って，“関数 x”は文脈 obj に定義される．“関数 x”は“入力値 p”をそのまま返す単純な関数である．

(2) **04 行目**

DSL としての記述の開始を“chglang 言語名”の形式で宣言している．ここでは，言語として Smalltalk を指定している．この試作では，Smalltalk を模した DSL のみを開発したが，ContextWorkbench の機構としては複数の DSL を組み込むことが出来る．従って，このように言語名を指定する必要がある．ContextWorkbench に固有な記述は，式とインタプリタに対するコマンドに分けられる．式は，“(”と“)”の間，あるいは“[”と“]”の間の記述である．ただし，左括弧の直前には，リテラルを意味する引用符が付く場合がある．また，コマンドは，式の外側に記述する．プログラム 3 の場合では，“do”，“chgctx”，“chglang”はコマンドである．コマンドは，文，単語，文脈名，言語名等の引数を取る．ContextWorkbench に固有な記述の実行では，コマンドとコマンドの引数以外の記述は読飛ばされる．従って，ファイル中に 03 行目や 08 行目のようなプログラムに影響を与えない自由な記述を行うことができる．ただし，DSL ではそのような記述を行うことは出来ない．

(3) **05 行目**

DSL としての記述である．“オブジェクト obj”に対して“コレクション#(1 2 3)”を引数に指定して“メッセージ x”を送信し，その戻り値を“変数 a”に代入するプログラムである．注目すべき点は，“オブジェクト obj”の由来である．このオブジェクトは，00 行目で定義された“文脈 obj”である．また，“メッセージ x”に対応して実行されるのは，“文脈 obj”に定義された“関数 x”である．従って，実行結果として“a”の値は“コレクション#(1 2 3)”になる．つまり，この DSL では文脈をオブジェクトとして扱う．

(4) **06 行目**

DSL としての記述の終了を意味する．なお，この記述は DSL 基盤におけるインタプリタに対するコマンドでる．先頭に“@”が付与された記述は DSL 基盤のコマンドを意味する．これにより，DSL としての記述とコマンドの区別が可能になる．また，ContextWorkbench に固有な記述におけるコマンドとの区別が容易になる．

(5) **07 行目**

プログラムを実行する文脈を“SmalltalkWorld”に変更している．この文脈は DSL 毎の初期化のタイミングで，その DSL に固有な文脈として指定される．また，この文脈は DSL の実行時に自動的に生成される．なお，05 行目の“a”のように DSL による記述の中で宣言された変数は，この文脈に定義される．

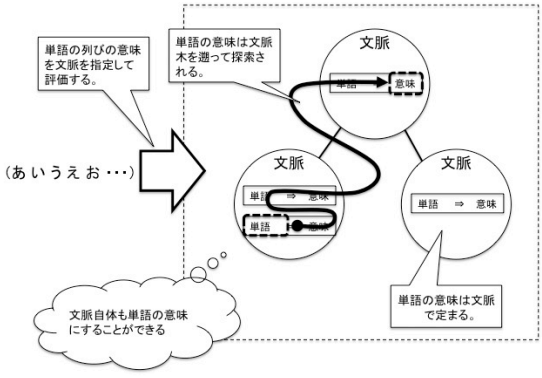

図 2　文脈木と単語の意味の探索

文脈の構造は，図 2 のように唯一の親を持つ木構造である．DSL のための文脈は，この木構造の唯一のルートの直下に定義される．このように記述間で文脈木を共有することで，記述間のシームレスな連携が可能になる．

(6) **09 行目から 10 行目**

06 行目で DSL としての記述が終了されたため，再び ContextWorkbench に固有な記述を再開する．プログラムは，07 行目で指定された DSL のための“文脈 SmalltalkWorld”において実行される．09 行目の“関数 printcr”は後続の単語の値を印字する関数である．従って，10 行目のように DSL 中で宣言された“変数 a”の値である“コレクション'(1 2 3)”が印字される．なお，この印字では，左括弧の前に引用符が付与されている．DSL は，Smalltalk におけるリテラルの指定を意味する“#”を引用符に置き換えることで自動的に評価が行われることを防ぎ，リテラルとして扱われるようにする．

3. ContextWorkbench に固有な記述の自動的な生成

3.1 基本的な変換

プログラム 4 に示すのは，プログラム 3 の 05 行目の実行時に ContextWorkbench が出力する内容である．ユーザはこの情報によって，実行された内容や環境を確認しながら対話的に作業を遂行出来る．

```
00: generated expression :
01:     ( ctx=SmalltalkWorld
02:             [ ( bind 'a ( ctx=obj x '( 1 2 3 ) ) ) ] )
03: session : default
04: context : SmalltalkWorld
05: value : true
```

プログラム 4　生成されたプログラム 1

プログラム 4 の 01 行目から 02 行目までが自動的に生成された ContextWorkbench に固有な記述である．2 章でも説明した通り，オブジェクトが文脈の指定に変換されている．また，リテラルの指定が引用符に変換されている．なお，05 行目が実行結果の値である．“関数 bind”は，単語の束縛に成功すると true を返す．

3.2 複数の文を含んだ DSL

複数の文を含んだ DSL を実行する例をプログラム 5 に示す．

```
00: chglang smalltalk
01: a := 1.
02: b := 2.
03: c := a + b.
04: @break
```

プログラム 5　複数の文を含んだ DSL

プログラム 5 には 01 行目から 03 行目まで 3 つの文が記述されている．このプログラムを実行した場合の出力内容をプログラム 6 に示す．

```
00: generated expression :
01:     ( ctx=SmalltalkWorld
02:             [ ( bind 'c ( 1 ) ) ( bind 'd ( 2 ) )
03:               ( bind 'e ( + ( c ) ( d ) ) ) ] )
04: session : default
05: context : SmalltalkWorld
06: value : ( true true true )
```

プログラム 6　生成されたプログラム 2

プログラム 6 の 01 行目から 03 行目までが自動的に生成された ContextWorkbench に固有な記述である．この記述は，1 つの文である．このように，1 回の DSL の開始から終了までに含まれる文は，1 つの文にまとめられて，一括して実行される．

3.3 二項メッセージ

プログラム 3 の 05 行目がキーワードメッセージの場合であったのに対して，プログラム 5 の 03 行目は二項メッセージの場合である．DSL の実行機能は，代入式の右辺やメッセージの引数について，2 番目の単語が二項メッセージであれば，先頭の単語を文脈の指定と見なさずに，二項メッセージを意味する単語をプログラム 6 の 03 行目のように，文の先頭へと移動する．

なお，プログラム 6 の 03 行目は，“(c)”と“(d)”の和を求めて“単語 e”を束縛するという内容であるが，括弧が自動的に外されて，“c”と“d”の和が求められる．ContextWorkbench に固有な記述の実行では，文の単語が 1 つである場合，自動的に括弧が外されて単語と見なされる．しかし，単語に対して文に対する処理を適用しようとすると， その単語は 1 つの単語の文と見なされる．例えば，文に単語を付与する“関数 append”で“1”に対して“a”を付与した結果は，“(1 a)”になる．このような

特徴は，DSL による記述から ContextWorkbench に固有な記述を生成することを容易にする．

3.4 局所変数

DSL には, Smalltalk の局所変数の形式で変数を宣言出来る．ただし，この記述は局所変数ではなく，“文脈 SmalltalkWorld”における束縛になるので注意が必要である．局所変数宣言を含んだ DSL による記述の例をプログラム 7 に示す. 01 行目が局所変数宣言である．この例では，“a”と“b”の 2 つの局所変数を宣言している．局所変数の宣言は“|”で区切られた内部に行う．

```
00: chglang smalltalk
01: | a b |
02: a := b.
03: @break
04: chgctx SmalltalkWorld
05: do ( printcr a )
06: => nil
```

プログラム 7　局所変数宣言

プログラム 7 の DSL による記述は，プログラム 8 の 01 行目から 03 行目に変換される．

```
00: generated expression :
01:     ( ctx=SmalltalkWorld
02:           [ ( bind 'a nil ) ( bind 'b nil )
03:             ( bind 'a ( b ) ) ] )
04: session : default
05: context : SmalltalkWorld
06: value : ( true true true )
```

プログラム 8　生成されたプログラム 3

プログラム 7 の 01 行目のように局所変数宣言を行うことで，02 行目に“(bind 'a nil) (bind 'b nil)”が挿入される．この記述は，変数の初期化の意味を持つ. これにより,“文脈 SmalltalkWorld”における“単語 a”や“単語 b”への束縛によって，予期しない結果が生じることを防ぐことが出来る．なお，プログラム 7 の局所変数宣言に“b”を含めなかった場合, プログラム 7 の 06 行目の“nil”は“b”になる．これは, ContextWorkbench が束縛されていない単語をリテラルとして扱うためである．

3.5 ブロック

ブロックは, Smalltalk を代表する機能であると言える．ブロックは“メッセージ value”を受信するまで実行されないため，遅延実行を可能にする．ブロックを含んだ DSL による記述の例をプログラム 9 に示す．プログラム 9 では，“a”と“d”の 2 つのブロックが定義されている．ブロックには，この例における“:b :c”や“:e”のように“|”で区切られた左側に実行時の引数を指定することが出来る．なお，引数の記述の後は局所変数の宣言も含めて，ブロックでない記述と同じである．また，引数は必須ではない．

```
00: chglang smalltalk
01: a := [ :b :c | ^b+c. ].
02: d := [ :e | | f |
03:          f := a value+e.
04:          ^f.].
05: @break
06: chgctx SmalltalkWorld
07: do ( bind 'b 1 )
08 do ( bind 'c 2 )
09: do ( bind 'e 3 )
10 do ( printcr eval d )
11: => 6
```

プログラム 9　ブロック

プログラム 9 の DSL による記述は，プログラム 10 の 01 行目から 06 行目に変換される．

```
00: generated expression :
01:     ( ctx=SmalltalkWorld
02:         [ ( bind 'a '[ ( return ( + ( b ) ( c ) ) ) ] )
03:           ( bind 'd
04:             '[ ( bind 'f nil )
05:                ( bind 'f ( + ( eval a ) ( e ) ) )
06:                ( return ( f ) ) ] ) ] )
07: session : default
08: context : SmalltalkWorld
09: value : ( true true )
```

プログラム 10　生成されたプログラム 4

プログラム 10 の 01 行目から 06 行目までが ContextWorkbench に固有な自動的に生成された記述である. ContextWorkbench に固有な記述の実行において, 引用符が付与された単語はリテラルとして扱われる．ただし，05 行目のように“関数 eval”を適用することで, 引用符を外して意味を判定することが出来る．この機構の効果は，Smalltalk におけるブロックと同じである．ただし，ContextWorkbench は，文脈の木構造を遡って単語の意味を判定するため, 引数に相当する単語を文の外側の文脈で束縛することで，引数と同様の効果を得ることが出来る．

従って，引数の指定に基づいた記述の生成は行われない．

3.6 オブジェクトの生成

Smalltalk はオブジェクト指向プログラミング言語である．従って，この DSL においてもオブジェクトの生成を行える必要がある．オブジェクトの生成に関する DSL による記述の例をプログラム 11 に示す． 02 行目は，“クラス obj”に対して“メッセージ new”を送信し，“変数 a”を生成されたオブジェクトに束縛していると読むことが出来る．

```
00: do (defctx 'obj)
01: chglang smalltalk
02: a := obj new.
03: @break
```

プログラム 11 オブジェクトの生成

プログラム 12 の 01 行目から 03 行目までが ContextWorkbench に固有な自動的に生成された記述である．ContextWorkbench の文脈機構では，文脈間の直接の親子関係をクラスとオブジェクトの関係として扱うことが出来る．何故なら，同じ文脈を親とする文脈において文を解釈する場合，同じ親の文脈に遡って単語の意味が探索されるため，同じ関数の実装が適用されることになる．また，それらは文脈であるから，実行結果の値の束縛は，親の文脈ではなく，各々の文脈で行われることになる．なお，この仕組みはクラスとオブジェクト間の通常の関係よりも柔軟である．何故なら，クラスからオブジェクトを生成しつつ，オブジェクトのレベルでもメソッド単位に実装を置き換えることが出来る．つまり，継承を行わずにオブジェクトを拡張することが可能になる．

```
00: generated expression :
01:     ( ctx=SmalltalkWorld [ ( bind 'a ( ctx=obj defctx
02:     '2d9067a76949ffba:-74096fac:133b2769d15
03:     :-7ffe ) ) ] )
04: session : default
05: context : SmalltalkWorld
06: value : true
```

プログラム 12 生成されたプログラム 5

なお，文脈は，直接の親に対して唯一な名前を持つ必要がある．そして，02 行目から 03 行目にかけて続いている文字列がそれである．この文字列の値は，java に標準で付属する VMID を用いて生成される．

4. まとめ

ContextWorkbench を拡張し，DSL による記述を実行する機構を実現することが出来た．また，その機構を利用して，Smalltalk をベースとするオブジェクト指向プログラミングに関する主要な機能を試作することが出来た．なお，この機構には，複数の DSL を組み込み可能であるから，コンパイラコンパイラの活用等によって，様々な DSL の組み込みについて簡易化を図りたいと考える．

参考文献

1) 森下敦司: “文脈によるプログラミング言語処理系”，情報処理学会夏のプログラミング・シンポジウム報告集, Vol. 2010, pp. 111-116 (2011).
2) Martin Fowler: “Domain-Specific Languages”, Addison-Wesley Professional (2010).
3) Adele E. Goldberg et al.: “Smalltalk-80: The Language and its Implementation”, Addison-Wesley (1983).