動的型付き言語上での構文拡張手段の提供

岩間 雄太　市川 和央　千葉 滋

本論文では，Ruby 言語上でモジュラーな構文拡張をおこなう言語機構を提案する．プログラマーはユーザ定義演算子を定義することで Ruby 言語の構文を拡張することができる．定義できる演算子の文法は，演算子の名前部分と引数部分が任意の順番で並んでいる文法である．提案する機構の特徴は，演算子にタグを付与することができる点，及びそのタグを利用して演算子の引数部で使用できる演算子を制限できる点である．タグは演算子に複数付与することができるので，新たに非終端記号を定義して構文拡張を実現する場合と比べより細かい演算子の制限が可能となる．

1　はじめに

内部 DSL(Internal Domain Specific Language) [4] とはホスト言語 (Ruby などの汎用言語) の内部で構築された DSL のことであり，ある特定のドメインの問題を解くためのライブラリやフレームワークのことを指す．Ruby，Scala といった言語では内部 DSL を作成するためにブロックや implicit parameter などの有用な機構が提供されている．Ruby on Rails，Rspec や ScalaTest など様々なライブラリで DSL を提供しており，アプリケーション開発には欠かせないものになっている．内部 DSL はホスト言語内に DSL を構築するため他の内部 DSL やホスト言語との連携が容易である．しかし同時に，内部 DSL の文法はホスト言語の文法に依存しているため表現できる文法に制限がある．DSL 開発者が自由に構文規則を作成できるような仕組みを導入すれば，ホスト言語の文法に依存しない内部 DSL を作成することができる．しかし，こうした場合，構文解析の結果が一意に決まらなくなる曖昧性という問題が発生する．曖昧性の問題は静的型付け言語であれば解決法は知られているが，我々の知る限り，動的型付け言語では知られていない．

本論文では，Ruby 上で構文拡張を可能にする機構 Erbs を提案する．Erbs を使用すると Ruby 上に Ruby の構文に依存しない自由な構文を持つ DSL を作成できる．Erbs は DSL 開発者が新しい演算子を定義することで構文拡張をおこなう．複数の構文拡張によって起こる曖昧性を回避するために，DSL 開発者はそれぞれの演算子にタグをつけられる．また，このタグにより，Erbs は自分が行った拡張を後から第三者が更に拡張しても，拡張が壊れることがないモジュラーな構文拡張方法を提供している．

2　Ruby 上での構文拡張

Ruby は高い表現力を持ち，柔軟な構文を表現可能である．例えば，シェルスクリプトのコマンドでファイルの内容を標準出力に出力する cat のような DSL を Ruby 上で表現する例を考える．cat コマンドは Ruby 上では Listing 1 のようにメソッドを定義する．Ruby ではメソッドの引数を囲む括弧を省略できるからである．このメソッドにより Ruby 上でシェルスクリプトの cat と同等のことができるようになり cat “hello.c” とすることで hello.c の中身を標準出力に出力できる．

このように Ruby は様々な DSL を作成できるが，

Yuta Iwama, Kazuhiro Ichikawa, Shigeru Chiba, 東京大学大学院情報理工学系研究科創造情報学専攻, Dept. of Creative Informatics, Graduate School of Information Science and Technology, The University of Tokyo..

Listing 1　cat を表現する Ruby のプログラムの例

```
def cat(filepath)
  puts File.read(filepath)
end

cat "hello.c"
  # => print contents of hello.c
```

作成できない DSL も存在する．例えば cat “hello.c” “world.c” > “hello_world.c” のような構文を許す DSL は作成できない．この構文は 2 つのファイル hello.c と world.c を受け取ってその 2 つの内容を足し合わせたものをファイル hello_world.c に出力する．Ruby の構文ルール上，メソッド呼び出しの第一引数と第二引数の間はカンマで区切られている必要があるため，この構文はエラーとなる．構文解析を成功させるためには， cat “hello.c”, “world.c” > “hello_world.c” のようにカンマを加えなければならない．カンマなしの構文を許す DSL を作成できるようにするには，Ruby 言語自体がユーザによる構文拡張を許さなければならない．

ユーザに構文拡張を許す仕組みを実現するにあたって難しい点は，拡張された構文が引き起こす構文の曖昧性をどう解決するかである．例えば上の例の場合，構文拡張により cat メソッドの第一引数”hello.c” と第二引数 ”world.c” の間のカンマを省略できるようにできたとしても，> の解釈をめぐる曖昧性を解決しなければならない．Ruby では > は文字列比較メソッドでもあるので，”world.c” > “hello_world.c” 全体を第二引数とみなす解釈もありうる．しかし DSL 作成者からすれば，第二引数は “world.c” だけで、cat メソッドの呼び出し式全体が > の左辺式 (あるいは > メソッドの呼び出し先) であると解釈されてほしいところである．

このような曖昧性は、Ruby の構文拡張を必要としない簡単な DSL でも起こりうる．例えば cat メソッドの引数を 1 個だけにした cat “hello.c” > “hello_back.c” の場合でも，> の解釈は曖昧性をともなう．この > の解釈の曖昧性は，Ruby のオープンクラス機構を用いて, String クラスの > メソッド自体を Listing 2 のように書き換えることで解決可能ではある．しかし，String クラスの > のような他の場所でも頻繁に使用されるメソッドを書き換える手法は，アプリケーションの他の部分を壊す可能性があり， 非常に危険なので避けるべきである．あるいは Method Seals [7] のような既存メソッドの書き換えの有効範囲を制御できる機構を使用して，String クラスの > の書き換えをある範囲内でのみ有効にする手法と組み合わせなければならない．

Listing 2　オープンクラスを使用した cat

```
class String
  def >(others)
    [self, others]
  end
end

def cat(filepath)
  source, out = filepath
  $stdout = File.open(out, "w")
  puts File.read(source)
  $stdout = STDOUT
end

cat "hello.c" > "hello_back.c"
```

3　構文拡張システム Erbs

本章では，Ruby 上の構文拡張機構 Erbs を提案する．Erbs では DSL 開発者が新しい演算子を定義することで構文拡張をおこなう．複数の構文拡張によっておこりうる曖昧性を回避するために，DSL 開発者がそれぞれの演算子にタグを付けることができる．演算子を定義する際に，特定のタグを持つ Ruby 式だけを演算子の引数部分に受け付けるように制限することで，構文解析時に候補となる構文木を減らすことができるので，曖昧性を回避できる．また，Erbs では同一演算子に複数のタグをつけることができる．そのため，一つしかタグを付けられない場合と比べ，引数部分に書ける Ruby 式の種類を細かく制限できる．これは複数の DSL を同時に使う場合に重要な機能である．

Listing 3　キーワード defs を使用した構文拡張

```
1  defs cat file1 file2 (
2    file1: origin,
3    file2: origin
4  )
5    puts File.read(file1)
6    puts File.read(file2)
7  end
8
9  cat "hello.c" "world.c"
```

3.1　ユーザ定義演算子を採用した Ruby 上での構文拡張

Erbs はユーザが演算子を新しく定義することで構文拡張を行う．演算子は 2 項演算子だけでなく 4 項以上の演算子も定義することが可能である．DSL 開発者は新しい演算子の文法を終端記号と Ruby 式が任意の順に並んでいるものとして定義できる．Ruby 式とは既存の Ruby の構文の式と新しくユーザが定義した演算子の式のことを言う．ユーザが定義した演算子の式を実行したときは，その演算子に結びつけられたメソッドボディが実行される．演算子のオペランド部分の Ruby 式は，評価された後にメソッドボディに引数として渡される．

Erbs では演算子を定義するために defs というキーワードを使用する．Listing 3 では defs を使用し言語拡張を行い cat コマンドのような構文を実現している．Listing 3 の 9 行目の cat コマンドを実行するとファイル hello.c と world.c を読み込み，その内容を標準出力に表示することができる．defs は 構文定義部，引数定義部，メソッドボディ部を持っている．構文定義部は Listing 3 の 1 行目の cat file1 file2 の部分で終端記号 cat の後に file1，file2 という Ruby 式を並べたものを定義している．ここは新しく定義する演算子の文法を指定する部分で，終端記号と Ruby 式の並びを指定する．引数定義部は 1 から 4 行目の (file1: origin, file2: origin) の部分である．ここは定義する演算子の文法中に含まれる Ruby 式を指定する．この部分で指定された Ruby 式は評価された後，メソッドボディ部に渡される．この例では，file1 と file2 が対応する Ruby 式の評価結果を表す引数を定義している．なお，origin については次節で説明する．メソッドボディ部は 5 から 6 行目の Ruby 構文のことである．ここには定義する演算子の式を実行したときに実行される Ruby の構文を指定する．この例では Ruby でファイル名を受け取り，そのファイルの内容を読み込んで標準出力に出力するようなコードを指定した．

Listing 4　曖昧性が発生する構文拡張の例

```
1  defs checkout bname (bname: origin)
2    "checkout " + bname
3  end
4
5  defs git cmd (cmd: origin)
6    exec("git " + cmd)
7  end
8
9  def checkout(time)
10   "checkout at " + time
11 end
12
13 git checkout "master"
```

3.2　タグを使用した曖昧性の回避手段の提供

単純な方法で構文拡張をおこなうと，定義したすべての拡張構文が任意の場所で使用可能になってしまい既存の構文と拡張した構文、または拡張同士が衝突して曖昧性が発生する．曖昧性が発生すると構文が壊れて意図しない動作を引き起こす危険がある．

checkout という git サブコマンドを Ruby 上に表現する例を Listing 4 で考える．この拡張を実現することで Ruby 上で git で管理されたリポジトリのブランチを変更することができる．Erbs では Listing 4 の 1 行目の git bname と 5 行目の checkout cmd という演算子を定義すると実現できる．しかし，9 行目のように Ruby のメソッドとして checkout が定義してある場合，曖昧性が発生し 13 行目の checkout “master” を 1 行目の? checkout か 9 行目の checkout どちらを使用して構文解析すればよいか一意に決まらない．

このような曖昧性を避けるために，曖昧性の回避手段を持つことは言語拡張をする際に重要である．

そこで，Erbs ではユーザが定義した演算子にタグ

Listing 5　キーワード Operator を使用し曖昧性を回避した構文拡張

```
1  Operator(git, subcommand)
2    defs checkout bname (bname: origin)
3      "checkout " + bname
4    end
5  end
6
7  Operator(origin, git)
8    defs git cmd (cmd: subcommand)
9      exec("git " + cmd)
10   end
11 end
12
13 git checkout "master"
```

をつけることができるようにした．引数部分でタグを指定し，構文解析時に候補となる演算子を減らすことで曖昧性を回避する．タグはすべての演算子には 1 つ以上付いている．Erbs ではタグを演算子につけるために Operator キーワードを使用する．Listing 5 は Operator を使用して演算子にタグをつけることで Listing4 で発生した曖昧性を回避した例である．Operator はタグ定義部とボディ部を持っている．Listing 5 の 1 行目の (git, subcommand) の部分がタグ定義部でボディ部で定義された演算子につけるタグを指定する部分である．2 から 4 行目はボディ部で，演算子を定義する部分である．演算子の定義は 3.1 節で述べた defs を使用する．すべての演算子にタグを付与するために defs は Operator の中でのみ使用可能とした．この例では定義した演算子 checkout bname には git と subcommand の 2 つのタグをつけ，8 行目の git cmd には origin と git というタグをつけた．タグを使用することで，演算子を定義するときに defs の構文定義部に現れる Ruby 式を特定のタグをもつものに制限できる．Listing 4 は git cmd の cmd の部分に任意の Ruby 式を許していたため曖昧性が発生した．そこで Listing 5 は git cmd の cmd の部分には subcommand というタグを持つ Ruby 式のみを受け付けるようにした．こうすることで，cmd の部分の既存のメソッド呼び出しとの曖昧性を回避することができる．また，Erbs では既存の Ruby 構文に対して origin というタグを付与しているため，git cmd は既存の Ruby 構文と同等に扱える．つまり，プログラム中では既存の Ruby 構文を書けるところであれば，自由に git cmd を書ける．

Listing 6　モジュラーな拡張の例

```
1  Operator(command)
2     defs cat file1 file2 (
3       file1: origin, file2: origin
4     )
5       puts File.read(file1)
6       puts File.read(file2)
7   end
8  end
9
10 Operator(origin)
11   defs input > output (
12     input: command && !@token(>),
13     output: origin
14   )
15     $stdout = File.open(output, "w")
16     input
17     $stdout = STDOUT
18   end
19 end
20
21 cat "hello.c" "world.c" > "hello_world.c"
```

3.3　モジュラーな曖昧性の回避

Erbs では，構文拡張をモジュラーに実装できるように，defs の中でタグを複数利用できる。defs の引数定義部では 論理演算子 &&, ||, !, @token を用いて，タグ同士あるいはタグと @token 述語とを組み合わせて指定することで，タグを一つしか使えない場合と比べ，引数定義部に書ける Ruby 式を詳細に制限することができる．DSL 開発者間でタグに共通の意味付けを与えることで，ある DLS 開発者が行った拡張を後から第三者が更に拡張しても，拡張が壊れることがないモジュラーな DSL を実装できる．

Listing 6 は複数のタグと述語を使用したモジュラーな DSL の実装例である．この例は Listing 3 の出力先を標準出力からファイルに変更する (21 行目)．今までの例のように単一タグだけを用いて，この例を実現すると Ruby 構文 String#> との間の曖昧性を回避できない．一方，ここでは 1 行目から 8 行目で

command というタグを持った cat file1 file2 という演算子を定義している．defs の引数定義部では file1, file2 ともに origin タグを持った Ruby 式を許しているので，すべての組み込みの Ruby 構文が使用可能となる．10 から 19 行目では タグ origin を持った input > output という演算子を定義している．この演算子の引数部分では output は origin というタグをもった Ruby 式を許す．一方で，input に書ける式の条件は command && !@token(>) となっている．こうすることで曖昧性を回避しつつモジュラーな言語拡張を可能にしている．&& はタグ同士の AND 演算を行う．例えば A && B という指定があった場合，受け付ける Ruby 式は A かつ B のタグを持つ全ての Ruby 式である．@token(>) の ! はタグの否定演算を行い，指定されたタグを持ってないすべての Ruby 式を受け付ける．@token は引数に指定した終端記号を式中に含むすべての Ruby 式を意味する．@token は推移的であり，!@token(>) に該当する Ruby 式は，その部分式の中であっても > を含まない式である．例えば式 (a > b) == (c < d) は該当しない．したがって Listing 6 の 12 行目の input の部分に書ける Ruby 式は，command というタグを持ち，かつ > という終端記号を含まないものである．このように定義することで input の Ruby 式中では > が使用できなくなり，Ruby 組み込みの String#> メソッドとの間で曖昧性が発生することを避けられる．

例えばこの DSL に >> という上書きのリダイレクトを可能にする演算子を追加することを考える．Erbs では Listing 7 のように実装することで，既存の Listing 6 を修正することなく，そのまま再利用して >> を実装できる．Listing 6 の 2 行目からの cat から始まる演算子は command タグを持つが，この演算子式の定義では > を式中に含んでもよいことになっている．> を式中に含んではいけないことは，12 行目に分離して書かれている．この分離により，Listing 7 では cat から始まる演算子式の定義をそのまま再利用している．Listing 7 の演算子の定義で必要な，cat から始まる演算子式の式中に >> を含んではいけないという制限は，Listing 7 の中に記されている．もし単一のタグしか使えないと，cat から始まる演算子式の定義中に > や >> に関する制限を直接記すことになるので，再利用性がなくなり，モジュラーな実装でなくなる．複数タグの利用は，Listing 6 の 11 行目と Listing 7 の 2 行目のリダイレクト演算子の定義の再利用も可能にする．cat に加えて ls コマンドを実行する演算子を定義する際も，その演算子に command タグを付加すれば，そのままリダイレクト演算子と組み合わせて利用できる．

Listing 7　既存の拡張を再利用した例

```
1  Operator(origin)
2    defs input >> output (
3      input: command && !@token(>>),
4      output: origin
5    )
6      $stdout = File.open(output, "w+")
7      input
8      $stdout = STDOUT
9    end
10 end
11
12 cat "hello.c" "world.c" >> "hello_world.c"
```

4　実装

Erbs のアイデアを実現するため Ruby サブセット上でソースコード変換器を実装した．この実装がサポートしている Ruby の構文はクラス，メソッド，変数の定義，メソッドの実行，制御構文，基本的な 2 項演算子などである．@token と タグが推移的になる部分は未実装であるが複数タグ，演算子 &&, ||, ! は実装済みである．この変換器はユーザが定義した拡張を含んだ Ruby プログラムを純粋な CRuby プログラムに変換し，変換後のソースコードを CRuby 上で実行する．演算子定義部分は CRuby 上の関数定義に変換され，演算子呼び出しは CRuby の関数呼び出しに変換される．Listing 8 に Listing 3 で定義した演算子を本実装で変換した後の CRuby のプログラムを示す．

Ruby の構文解析器は Scala のパーザコンビネータライブラリを使用して実装した．既存の Ruby の構文解析器を使用しなかったのは Ruby の構文解析器はパーザジェネレータで作られており，実行時に拡張

Listing 8　変換後の Ruby のソースコード

```
module Operator::Origin
  def self.cat_File1_File2(file1, file2)
    puts File.read(file1)
    puts File.read(file2)
  end
end

Operator::Origin::cat_File1_File2(
  "hello.c", "hello2.c"
)
```

することは困難であったためである．Scala のパーザコンビネータライブラリは Packrat Parsing [3] を採用しており，再帰下降構文解析では最悪で指数時間かかる解析を線形時間で行うことを可能にしている．

5　関連研究

演算子を利用して内部 DSL を構成し構文拡張を行う言語や研究はこれまでにも多く存在する．C++は演算子オーバーロードの機構を持っており，既存の演算子のほとんどはオーバーロード可能になっている．しかし，C++の演算子オーバーロードは新たに演算子を定義することができない．Ruby はオープンクラスの機構を使用することで演算子オーバーロードを実現することができ，既存の演算子であればほぼすべてオーバーロードすることが可能である．しかし，この機構も C++と同様に新たな演算子を定義することはできない．Smalltalk [5] のメッセージ式である 2 項メッセージとキーワードメッセージを使用すると N 項演算のような形で構文を記述できる．しかし Smalltalk の 2 項メッセージには使用することのできない記号がある．また，キーワードメッセージでは終端記号の後に必ず :(コロン) が必要となる．SugarJ [2] は言語拡張をライブラリという形で提供することのできる Java 言語の拡張である．ホスト言語が構文解析できないような新しい文法をユーザが定義しその文法をシンタックスシュガーとして追加できる．定義したシンタックスシュガーをインポートすることでホスト言語の文法を変えることができる．しかし，曖昧性がある場合にはコンパイルエラーとなる．

静的型付け言語上で型を使用して曖昧性を回避している研究もある．Mixfix operators [1] は Coq や Isabelle などの言語で実装されているユーザ定義可能な演算子である．Mixfix operator は非常に強力であり様々な構文を表現することができるが名前部分と引数部分が交互にこなければいけないという制約がある．また，静的型付け言語上で実装されているため曖昧性の解決に引数部分の型を使用している．ProteaJ [6] は protean operator という演算子を使用して強力な構文拡張を可能した Java 言語の拡張である．protean operators とは演算子の名前部分と引数部分によって表現する演算子である．Mixfix operators と比べて名前部分と引数部分の並びに制限も存在しない．曖昧性の回避に返り値の型と引数の型の両方を使っているが，動的型付け言語だとこのような方法は取ることができない．

6　まとめと今後の課題

本論文では，動的型付け言語上でモジュラーな構文拡張を可能にする Erbs を提案した．ユーザが自由に新しい演算子を定義し，その演算子にタグを付与できるようにした．Erbs は複数のタグの利用を許すので，ある DSL 開発者が実装した構文拡張を，後から第三者がそれを変更せずに再利用して，モジュラーに拡張することができる．今後の課題として，述語@token の実装を完成させること，またこの@token 舞うようにすることが挙げられる．

参 考 文 献

[1] Danielsson, N. A. and Norell, U.: Parsing Mixfix Operators, *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages*, IFL'08, Berlin, Heidelberg, Springer-Verlag, 2011, pp. 80–99.

[2] Erdweg, S., Rendel, T., Kästner, C., and Ostermann, K.: SugarJ: Library-based Syntactic Language Extensibility, *Proc. of OOPSLA 2011*, ACM, 2011, pp. 391–406.

[3] Ford, B.: Packrat Parsing:: Simple, Powerful, Lazy, Linear Time, Functional Pearl, *SIGPLAN Not.*, Vol. 37, No. 9(2002), pp. 36–47.

[4] Fowler, M.: InternalDslStyle, `http://martinfowler.com/bliki/InternalDslStyle.html`, 2006.

[5] Goldberg, A. and Robson, D.: *Smalltalk-80: The*

Language and Its Implementation, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[6] Ichikawa, K. and Chiba, S.: Composable User-defined Operators That Can Express User-defined Literals, *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, New York, NY, USA, ACM, 2014, pp. 13–24.

[7] 嶺福室, 滋千葉: コールスタックに基づいてクラス拡張の有効範囲を制御するための言語機構の提案, 日本ソフトウェア科学会大会論文集, Vol. 32(2015), pp. 6p.