東京大学
情報理工学系研究科 創造情報学専攻
修士論文

他の拡張構文との衝突の回避に留意した構文拡張機構 Erbs の開発

Erbs: Syntax extension mechanism avoiding conflicts with other extended syntax

岩間 雄太
Yuta Iwama

指導教員　千葉 滋 教授

2017 年 1 月

概要

異なる開発者による拡張構文との衝突の回避に留意した構文拡張を可能にする言語機構 Erbs を提案する．Erbs を使用すると，Ruby には存在しない構文を持つ DSL を作成できる．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．定義できる演算子は，名前部分と引数部分が任意の順番で並んでいる文法を持つものである．ある拡張構文を定義した後に，第三者が新たな拡張構文を定義した場合，定義した構文同士が衝突する可能性がある．これにより，構文解析の結果が一意に決まらなくなったり，成功してはいけない構文解析が成功したりする，といった問題が発生する．Erbs では，この問題を回避するために，定義した拡張構文にタグを付与できるようにし，タグ同士の論理演算によって，ある拡張構文中では他の拡張構文の使用を制限できるようにした．これらの機構により，構文解析時に候補となる構文の数を減らし，この問題を回避しやすくした．

Abstract

We propose Erbs, which is syntax extension mechanism avoiding conflicts with other extended syntaxes by different developers. Erbs allows users to create a DSL that doesn't exist in Ruby. DSL developers extend a syntax by defining new operators in Erbs. The grammar of Erbs' operators is that the name part and the argument part are arranged in an arbitrary order. If DSL developer defined new operators after defining an operator by another DSL developer, the problem which the operators conflict with operators defined by another developer would occur. This problem is called an ambiguity. In order to avoid this problem, Erbs allows tags to be added to the defined operators. And Erbs restricts the use of operators in other operators by a logical operation between tags. These mechanisms reduce the number of candidate syntaxes at the time of parsing, making it easy to avoid this problem. These mechanisms reduce the number of candidate syntaxes at the parsing time and make it easy to avoid this problem.

目次

第 1 章

はじめに

現代のソフトウェア開発では，ますます開発のスピードが上がっており，製品を素早く動く形にしユーザまで届けることが重要になっている．このような流れに合わせて，ただ動くだけのプログラムではなく可読性が高さ，再利用性，安全性やメンテナンス性などが重要になっている．開発スピードをあげて，品質が高いソフトウェアを開発するためには，OSS(オープンソースソフトウェア) のような，他者によって開発およびメンテナンスされているライブラリやフレームワークを用いて開発すること多くなっている．ソフトウェアを開発する際には一からすべて作るのではなく，すでに公開されているソフトウェアを組み合わせて使用することが当たり前となっている．OSS として公開されているフレームワークやライブラリは可読性の高い構文をユーザに提供するために，ホスト言語の機能を使用して自然言語に近い内部 DSL を構成している．内部 DSL[1] とはホスト言語 (Ruby などの汎用言語) の内部で構築された小さな言語のようなもので，ある特定のドメインの問題を解くための設計，構築される．内部 DSL は Ruby on Rails [2] や ScalaTest などで積極的に使用されている．これは Ruby[3] や Scala [4] などのモダンな言語は内部 DSL を構築しやすいような機能を豊富に提供していからである．さまざまなライブラリが提供している内部 DSL を組み合わせて開発する言語指向プログラミング [5] をおこなうことができる．汎用言語内で特定のドメインを解く際に，内部 DSL はホスト言語との相性がよいのでドメインの問題を可読性，再利用性を保ったまま記述できる．内部 DSL はホスト言語と連携がしやすく，プログラム可読性を高めるのに非常に有用であるが，ホスト言語上で構成されるという性質からホスト言語の文法によって表現できる文法が制限されている．ホスト言語の文法に制限されずに新しい構文を作成する方法はいくつかある．例えば，外部 DSL を使用するという方法がある．外部 DSL とはホスト言語とは全く別の構文を持った DSL のことである．字句解析や構文解析器などを自分で実装することで，ホスト言語に依存しない構文を作成することができる．外部 DSL は正規表現，SQL や YACC (Yet Another CompilerCompiler)[6] などがあり，様々な場所で使用されている．外部 DSL はホスト言語の文法に制限されることはないが，ホスト言語と連携して使用することはほぼ不可能である．また，字句解析器などを自分で実装しなければならず作成のコストが高いなどの欠点がある．他には，字句マクロや構文マクロなどの方法がある．字句マクロとは構文解析前に一致する文字列をすべてほかの文字列に置換し，そのあと構文解析をするマクロ

である．構文マクロは文字列ではなく，抽象構文木を使用して置換をおこなう．このマクロは構文解析後に行われ，指定した抽象構文木と一致したとき，その部分を他の構文木に変換する．字句マクロは C や C++ などの言語で使用されており，構文マクロは Lisp 族の言語，MetaML[7]，Template Haskell[8] などの言語で使用されている．この二つの方法は，ホスト言語との連携をすることが可能であるが，予期しない部分を置換してしまいホスト言語とマクロ間で曖昧性が発生しエラーを引き起こす危険性があり，十分注意して使用しなければならない．曖昧性とは構文解析の結果，構文同士が衝突し結果が一意に定まらないことである．

他の方法として，ホスト言語の構文を拡張可能にすることでホスト言語の文法に制限を受けない構文を定義するという方法がある．この方法を使用すると字句解析などをユーザが作成する必要がなく，ホスト言語の文法に制限を受けない構文を作成することができる．この手法は SugarJ[9] や ProteaJ[10] などの言語で実現されている．これらの言語は言語拡張をしたときに発生する曖昧性を型を使用して解決している．構文解析時ではなく，実行時に型が決定する動的型付け言語では，型を使用しているこの方法は使用できないという問題がある．また，ホスト言語と拡張構文の間に発生する曖昧性を解決するのは既存の手法で考慮されてきたが，拡張構文同士の間に発生する曖昧性を考慮するのは困難である．なぜなら，自分が拡張構文を定義したあとに更に第三者が拡張するので，拡張構文を定義した時点では今後どんな拡張が定義されるか予期できないからである．

そこで我々は，異なる開発者による拡張構文との衝突の回避に留意した構文拡張を可能にする言語機構 Erbs を提案する．Erbs は Ruby の構文を拡張することで Ruby に存在しない構文を作成することができる．ホスト言語の文法の制約がなくなるため，ホスト言語では表現できないような自然言語に近い可読性の高い内部 DSL を構築できる．Erbs は演算子を定義することで，Ruby 言語を拡張している．演算子を定義するときに，特定のタグを持つ Ruby 式 (ユーザが定義した演算子と組み込みの式) 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，曖昧性の発生やユーザの意図しない動作を回避できる．また，Erbs では一つの演算子に複数のタグをつけ，それらの論理演算をとることができる．そのため，タグを一つしか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．

本論文では，我々の提案する Erbs のアイデアと背景，および制約，そして利用例について述べる．特に，タグをどのように使用すると曖昧性を回避できるかについて述べる．本論文では Ruby 上に Erbs のアイデアを実装した．この処理系の実装と利用例についても紹介する．二章では構文拡張をする際に発生する曖昧性について述べる．発生する曖昧性を，ホスト構文と拡張構文間の曖昧性と，拡張構文間同士の曖昧性に分けそれぞれについて詳細に説明する．また，内部 DSL と外部 DSL の違いとそれぞれの構築例についても述べ，最後に関連研究について述べる．三章では我々が提案する Erbs の概要と解決する問題について例を使用して説明する．Erbs が使用する構文拡張，どのように曖昧性を回避しているか，そしてどのような機能を持っているか，またそれらの機能はどのような制限を持っているかについて述べている．四章では我々が実装した Erbs の実装について紹介する．Erbs はソースコード変換器であり，

変換した Ruby コードを CRuby の処理系に渡してソースコードを実行している．変換の際，Erbs はどのように演算子を CRuby のコードに変換しているかについて述べる．また，Erbs がサポートしている Ruby のサブセットについても述べる．さらに，Erbs がどのような実装をされているか，定義された演算子は実行されるまでにどのように内部的に持たれているかについて述べる．五章では我々が実装した Erbs が表現できる様々な構文について紹介する．六章では本論文のまとめと今後の課題について述べる．

第 2 章

ドメイン固有言語と構文拡張

内部 DSL は様々な分野のアプリケーションで使用されており，アプリケーション開発に欠かせないものとなっている．内部 DSL はホスト言語の上に実装されるため，文法に大きな制限がある．そのため，ホスト言語を拡張する様々な手法が提案されている．ホスト言語の文法が制限されないように，ユーザに構文拡張の手段を提供すると拡張構文同士，あるいは拡張構文とホスト言語の既存の構文とが衝突し構文解析エラーの発生や，意図しない挙動になるという問題がある．本章では，DSL の詳細な説明し，既存の言語拡張機構を紹介する．さらに，Ruby 言語上で構文拡張を行った際に発生する具体的な問題について触れる．最後に言語拡張についての関連研究を述べる．

2.1 ドメイン固有言語

ドメイン固有言語 (DSL) は，ある特定のドメインの問題を解くように作られたプログラミング言語，またはライブラリのことである [11]．ドメインについて知識を持っているユーザが DSL を使用することで，問題の理解や修正を容易に行うことができる．また，DSL は汎用言語とは違い，小さな言語であるため，簡潔に問題を表すことができ，メンテナンス性が向上する [12]，可読性が向上する [13] などの利点がある．DSL のよく知られている例として，YACC (Yet Another Compiler Compiler) [6][14]，SQL などがある．YACC とはプログラミング言語の構文を定義するための DSL を提供しているパーザージェネレーターのことである．ソースコード 2.1 は YACC を使用した単純な四則演算の定義例の一部である．YACC の DSL を使用して文法を定義することで，四則演算の構文を文解析する構文解析器のコードを生成することができる．SQL は DBMS(Data Base Management System) を操作するための DSL である．ソースコード 2.2 は users テーブルから年齢 (age) が 20 以上のユーザをすべて探す SQL 文である．この構文を DBMS に渡すことで，SQL の結果を得ることができる．以上のようにドメイン専用の言語を使用することで，汎用言語上でプログラムを作成するよりも宣言的にそのドメインについてのプログラムを記述することができ，可読性やメンテナンス性が向上する．

DSL は内部 DSL と外部 DSL に分類できる．内部 DSL はホスト言語内に構築する DSL の

ソースコード 2.1. YACC による単純な四則演算の定義の一部

```
1  expr: NUM
2      | expr '+' expr { $$ = $1 + $3; }
3      | expr '−' expr { $$ = $1 − $3; }
4      | expr '∗' expr { $$ = $1 ∗ $3; }
5      | expr '/' expr { $$ = $1 / $3; }
6      ...
```

ソースコード 2.2. users テーブルから 20 歳のユーザを表示する SQL 文

```
1  SELECT ∗ FROM users WHERE users.age > 20;
```

ソースコード 2.3. Ruby 上での cat の定義

```
1  def cat(filepath)
2    print File.read(filepath)
3  end
4
5  cat "hello.c"
```

ことで，外部 DSL はホスト言語とは全く異なった独自の構文を持つ DSL のことである．以下の節では内部 DSL と外部 DSL の詳細な説明，およびその利点と欠点について述べる．

2.1.1 内部 DSL

内部 DSL[1] 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 のことである．ホスト言語の書き方を工夫することで自然言語に近い構文を表現している．内部 DSL 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．しかしホスト言語の構文上で構築するため，ホスト言語と組み合わせて使用することができる．

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 ソースコード 2.3 のようにメソッドを定義することで実現できる．Ruby ではメソッド呼び出しを実行する際に，method_name(args1, args2) というようにメソッド名の後に括弧が続き，その後引数を書き最後に括弧で閉じるが，引数を囲む括弧を省略して method_name args1, args2 というように書くこともできる．このような特別なルールが Ruby にはあるのでソースコード 2.3 のような構文を実現できる．このメソッドにより Ruby 上でシェルスクリプトの cat と同等のことが実現できるようになり 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．このようにホスト言語が本来備えている機能を工夫して使用することで、内部 DSL は構築される．

様々な場所で内部 DSL を使用している例が存在する．Ruby on Rails[2] や Rspec，scalaTest といったライブラリで積極的に内部 DSL が使用されており，アプリケーション開発において欠かせないものとなっている．

ソースコード 2.4. Ruby のクラス継承構造を表した Graphviz の外部 DSL

```
1 digraph ruby_class {
2     "Enumerable" -> "Object" -> "Kernel" -> "BasicObject";
3     "Data" -> "Object"
4     "Thread" -> "Object"
5     "Array"  -> "Enumerable"
6     "Hash"  -> "Enumerable"
7 }
```

2.1.2　外部 DSL

外部 DSL とはホスト言語とは全く別の構文を持った DSL のことである．外部 DSL を作成するにはホスト言語で，字句解析器，構文解析器や評価器などを実装する必要があるため作成するコストが内部 DSL にくらべて大きい．また，ホスト言語との連携がほぼ不可能である．しかし，外部 DSL の文法はホスト言語に依存しないため，ホスト言語に制限されず様々な構文を定義，使用することが可能である．

ソースコード 2.4 はグラフ可視化ツールの一つである Graphviz[15] というソフトウェアの外部 DSL である．Graphviz 自体は C++ で実装されている．ソースコード 2.4 のような DSL を記述することで，図 2.1 の有向グラフを作成することができる．図 2.1 は Ruby のクラス継承構造を dot 言語で表した図である．

2.1.3　既存の DSL の実装方法

DSL の実装方法は様々である [11]．以下では DSL の既存の実装方法について述べ，さらにそれぞれの方法の利点，欠点を述べる．

コンパイラやインタープリタの実装

新しくコンパイラやインタープリタを実装することで DSL を構築する方法がある．この手法は，YACC や Graphviz などのツールで使用されている．自由に DSL の文法を設計することができるため，表現力の高い DSL を作成することができる．また，言語の最適化，エラー処理や静的解析なども他の方法にくらべ容易である．しかし，コンパイラを実装するコストを無視することができない．DSL 用に字句解析器，構文解析器，評価器を実装しなければならないからである．また，ホスト言語との連携もできないため再利用性に欠ける．

関数や演算子を使用した実装

既存の関数定義やユーザ定義の演算子を使用することで DSL を構築する．Scala[4] や Ruby[3] などの言語は内部 DSL を構築するために様々な機構を提供している．例えば，Scala の場合 Implicit Conversion や Implicit Parameter．Ruby の場合はブロックやオープンクラ

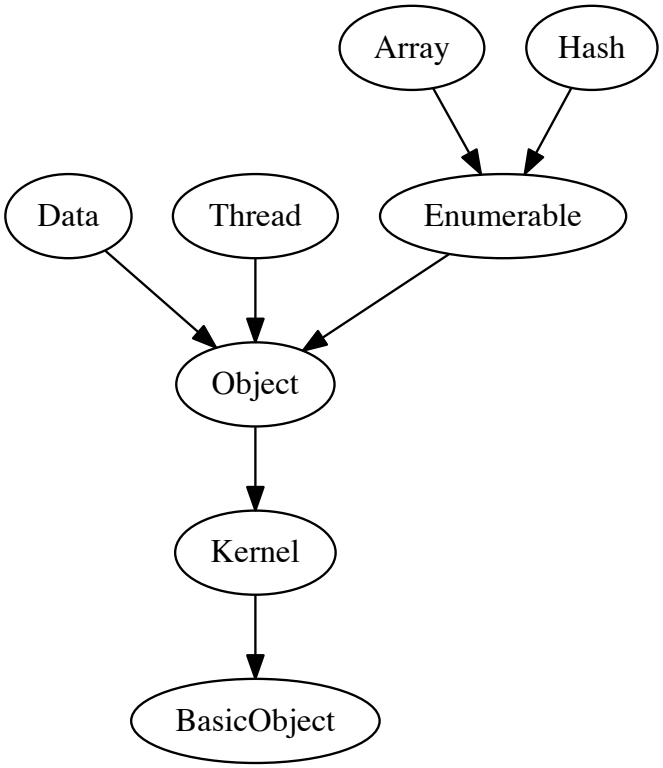

図 2.1. Ruby のクラス継承構造を Graphviz で作成した図

スなどである．この手法はホスト言語の構文をそのまま使えるので，DSL を作成するコストが低い．また，同じ理由でホスト言語との連携が容易である．さらに，定義した DSL を再利用しやすいことなどの利点がある．しかし，ホスト言語の文法のみを使用するため構築できる文法に非常に大きな制約がある．

マクロ使用した実装

字句マクロや構文マクロなどのマクロ機構を使用して DSL を構築する方法がある．この方法の利点は関数や演算子を使用した実装と同様に DSL を作成するコストが低いこと，ホスト言語上で実現しているのでホスト言語との連携が容易で，定義した DSL を再利用しやすいという点である．欠点は，ユーザが構文木を意識しなければならない点，マクロとそれ以外の構文が衝突し意図しない挙動になることがあるなどがある．字句マクロと構文マクロについての説明を以下に述べる．

字句マクロとは，ソースコード中に現れる特定のパターンの文字列を置き換えるマクロのことである．C 言語や C++ 言語では #define macro_name と書くことにより字句マクロを定

ソースコード 2.5. C++ で作成したシェル風の DSL

```
1  #define cat (new Cat)
2  #define hello_world ->run("hello.c")
3  #define world "world";
4
5  class Cat {
6  public:
7    Cat run(string s) { return *this; }
8    void operator > (string s){
9      // 入力をリダイレクトする
10   }
11 };
12
13 cat hello_world > world
```

義できる．定義されている字句マクロは字句解析処理と構文解析処理の間で展開される．ソースコード 2.5 は C++ 上で，シェルスクリプトの cat コマンドの出力を他のファイルにリダイレクトする疑似コード例である．二つの字句マクロ cat，hello_world と world を定義することで本来の C++ では表現できないような DSL を実現できるようになる．しかし，ソースコード中の特定のパターンに合致したすべて文字列を変更してしまうので，cat という関数や変数を定義して使用することができなくなる．意図せず cat という変数などを定義すると変換されて意図しない動きをしバグの原因なる．

構文マクロとは，プログラムを構文解析して得た抽象構文木を別の抽象構文木に変換するマクロのことである．Lisp 族の言語，MetaML[7] や Template Haskell[8] などの言語が提供している．構文マクロは，構文解析が終了し，得られた抽象構文木が評価される前に実行される．字句マクロとは違い，文字列で一致して置き換えるのではなく構文を見るため，字句マクロよりも名前の衝突に強く安全である．Scheme の構文マクロは hygienic [16] であることが保証されている．これは，構文マクロの定義中で使われている変数名がそのマクロを利用している場所の環境に存在する変数名と衝突していた場合にも，マクロ構文が変数を上書きしないことを保証しているということである．ソースコード 2.6 は lisp 上で，シェルスクリプトの cat コマンドの出力を他のファイルにリダイレクトする例である．この lisp 処理系では，defmacro というキーワードを使用することで構文マクロを定義している．構文全体を括弧で囲んでいるものの，構文マクロを使用することでシェルスクリプトの文法とほぼ同じ構文を表現できる．しかし，他のマクロの中でマクロを展開しようとすると意図しない挙動になることがあり完全に曖昧性を回避しているとはいえない．

拡張可能なコンパイラ

拡張可能なコンパイラとは，ユーザがホスト言語の文法を拡張し，文法の制約を緩和することでホスト言語に依存しない構文を構築できるようにしている言語ことである．ホスト言語の文法を拡張することで，ホスト言語との連携を取りながら，ホスト言語の文法に制約を受けな

ソースコード 2.6. lisp 言語で作成したシェル風の DSL

```
1 (defmacro cat (input-file d output-file)
2   (if (equal d '>)
3 ␣␣␣␣␣(let␣(c␣(file-open␣input-file))
4 ␣␣␣␣␣␣␣␣␣(file-write␣c␣output-file))))
5
6 (cat␣"hello␣world"␣>␣"world")
```

ソースコード 2.7. Ruby の構文では表現できない DSL

```
1 cat "hello.c"  "world.c" > "hello_world.c"
```

い構文を定義できる．この手法の実現方法としては，ユーザ定義の演算子を使用する方法やユーザに BNF などの構文を定義できるようにしてコンパイラを拡張できるようにする方法などある．ユーザがプログラミング言語の構文を自由に拡張可能であると，標準化委員会や言語の開発者から承認を受けずに言語を拡張できる．拡張可能なコンパイラを持っているプログラミング言語として Honu [17] などがある．言語拡張を行うと，ホスト言語の文法と拡張構文，または，拡張構文の文法が衝突する．文法が衝突すると，構文解析の結果が一意にならない曖昧性という問題が発生したり，意図しない構文が適用されて意図しない計算結果を返したりするようになる．

2.2 Ruby 上での構文拡張によって発生する問題

Ruby はそれ自体でも高い表現力を持ち，柔軟な構文を表現可能である．例えば，ソースコード 2.3 で述べたように Ruby シェルスクリプトの cat を表す DSL を実現することができる．Ruby は様々な DSL を作成できるが，作成できない DSL も存在する．例えば ソースコード 2.7 のような構文を許す DSL は作成できない．この構文は二つのファイル hello.c と world.c を受けとり，その二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，method_name args1 args2 ではなく，method_name args1, args2 というように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．つまり，構文解析を成功させるためには cat “hello.c”, “world.c” > “hello_world.c” のように第一引数と第二引数の間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 の表現力では限界があるため，Ruby 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．曖昧性とは，構文解析の結果，構文同士が衝突し結果が一意に定まらないことである．曖昧性が発生する場所は大きく二つ分けることができる．一つ目は，既存の構文と拡張構文の間と．拡張構文同士である．

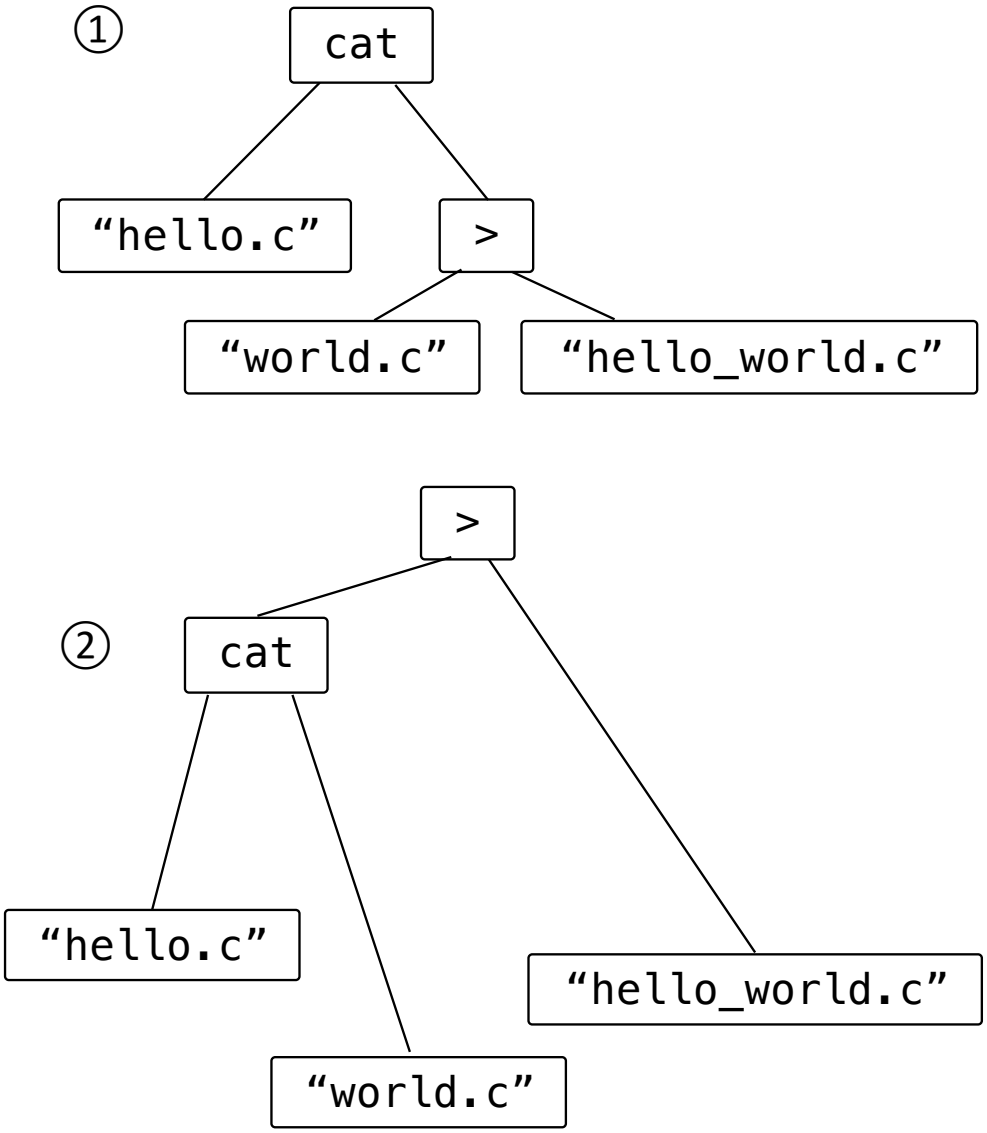

図 2.2. 曖昧性のある二つの AST

2.2.1　既存の構文と拡張構文同士に発生する曖昧性

構文拡張の作者が既存の構文，ここでは組み込みの Ruby 構文，に十分に気をつけて構文拡張をしなければ簡単に既存の構文と拡張構文同士の曖昧性が発生する．例えばソースコード 2.7 の場合は構文拡張により 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，Ruby の組み込みの > メソッドと作成した拡張構文 > の間で発生す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，図 2.2 の 1 のように"world.c" > “hello_world.c” 全体を第二引数とみなす解釈ができてしまう．しかし DSL 作成者からすれば，図 2.2 の 2 のように第二引数は “world.c”

ソースコード 2.8. オープンクラスによる組み込みクラスの書き換え

```
1  p [1, 2, 3].to_s                    # "[1, 2, 3]"
2
3  class Array
4    def to_s
5      self.join
6    end
7  end
8
9  p [1, 2, 3].to_s                    # "123"
```

ソースコード 2.9. オープンクラスを使用して実現した cat

```
1  class String
2    def >(other)
3      [self, other]
4    end
5  end
6
7  def cat(filepath)
8    source, out = filepath
9    $stdout = File.open(out, "w")
10   puts File.read(source)
11   $stdout = STDOUT
12 end
13
14 cat "hello.c" > "hello_back.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.

さらに曖昧性は, 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 メソッドの引数を一個だけにした cat “hello.c” > “hello_back.c” の場合でも, > の解釈についての曖昧性が発生する. この > の解釈についての曖昧性は, Ruby のオープンクラス機構を用いると解決できる. オープンクラスとは既存のクラスの挙動を後から再定義, または新しい挙動を追加しもとのクラスの挙動を変更することができる機能である. ソースコード 2.8 は組み込みクラスである配列クラスにたいしてオープンクラスを用いて文字列に変換するメソッドの挙動を再定義した例である. このプログラムは Array というクラスの to_s というメソッドを定義している. この機能を使用して String クラスの > メソッドをソースコード 2.9 のように書き換えれば DSL 作成者の意図どおりの挙動となる. 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. 他の方法として Method Seals[18] 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実現できない.

ソースコード 2.10. help オプションを表す DSL

```
1  git --help
2  diff --help
```

2.2.2　拡張構文同士で発生する曖昧性

拡張構文同士で発生する曖昧性は更に解決することが難しい．自分が拡張構文を定義したあと，第三者が更に拡張するので拡張構文を定義した時点では今後どんな拡張が定義されるか予期できないからである．例えば，Ruby 上で git コマンドと diff コマンドの ヘルプオプションを実現するソースコード 2.10 のような拡張を実現する例を考える．このプログラムを実行するとそれぞれのコマンドのマニュアルが標準出力に表示される．git コマンドの –help という拡張構文により実現したときに，それを知らない第三者が diff コマンドの –help という拡張を定義したとする．この二つの拡張が同時に使われた場合，二つの拡張構文 –help が衝突し曖昧性が構文解析エラーが発生したり，どちらかが上書きされ意図しない挙動をしたりするようになる．

2.3　関連研究

これまで様々な言語で構文拡張やそれによって発生する曖昧性の回避の研究が行われている．それらの研究について詳細に述べる．

2.3.1　演算子オーバーロード

演算子オーバーロードとは，既存の演算子をユーザが定義したデータ型に定義できる機構ことを言う．C++，Ruby[3] や Ada[19] などの言語はこの機能をサポートしている．

Ruby の標準ライブラリの一つである Matrix では，演算子オーバーロードを使用して様々な演算子を実現している．Matrix は Ruby で行列を扱うために使うクラスである．例えばソースコード 2.11 は以下の行列の 2 乗を計算をするプログラムである．この機能を実現するために Matrix クラスに ソースコード 2.12 のような ** というメソッドを定義している．Ruby や C++ の演算子オーバーロードには制限があり，任意の演算子を定義できない．

$$A = \begin{bmatrix} 7 & 6 \\ 3 & 9 \end{bmatrix}$$

2.3.2　ユーザ定義の中置二項演算子

Haskell，ML，Scala，Prolog などの言語を使用することで，ユーザ定義の中置の 2 項演算子を定義することができる．中置の 2 項演算子とは，_ op _という並びで表される演算子であ

ソースコード 2.11. 行列のべき乗を計算する Ruby プログラム

```
1 Matrix[[7, 6], [3, 9]] ** 2
2
3   => 67 96
4      48 99
```

ソースコード 2.12. Matrix クラスの実装の引用

```
1 class Matrix
2   ...
3
4   def ** (m)
5     ...
6   end
7 end
```

ソースコード 2.13. ScalaTest のテスト例

```
1 result should be (3)
2 string should startWith regex "hel*o"
3 result should not be an [Oranguatan]
```

る．_は演算子の引数部分を，op を演算子の名前部分ことを表す．この機能は，演算子オーバーロードを更に拡張したものである．演算子オーバーロードは既存の演算子に対してオーバーロードするだけだったが，ユーザ定義演算子は既存の演算子以外にも様々な演算子を定義することができる．ユーザ定義演算子を使用することで，より自然言語に近い構文を構築することができため簡潔に対象の問題を表現できる．

中置の二項演算子を使用した例として，ScalaTest という Scala 製のライブラリがある．ソースコード 2.13 の一行目は result の値が 3 であることを確認するテスト．二行目は result のクラスまたはトレイト名が Oranguatan と一致するかのテスト．三行目は string が"hello"から始まる文字列かを確認するためのテストである．一行目は should が中置二項演算子で，result と be(3) がこの演算子の引数である．二行目の構文は should と regex が中置の 2 項演算子である．regex の引数は startWith と "hel*o" である．また，startWith は定数である．should の引数は string と二項演算子 regex が作る式である．三行目の構文は should と be と an が中置の 2 項演算子である．これらの構文解析のルールは非常に複雑である．例えば二行目の"hel*o には括弧は省略できるため付いてないが，一行目の括弧は省略できない．また，どれが定数で演算子なのかということが非常にわかりづらい．

ソースコード 2.14. メッセージ式の例

```
1  9 sqrt.
2  1 + 2.
3  100 between: 101 and: 99.
```

ソースコード 2.15. メッセージ式で表した ScalaTest

```
1  result should be 3.
2  string should: startWith regex: "hel*o".
3  result should: not be: an Oranguatan.
```

2.3.3　メッセージ式

Smalltalk [20] のメッセージ式を使用して内部 DSL を構築することができる．ソースコード 2.14 の一行目のように，オブジェクト (ソースコード 2.14 の一行目では 9) に対してメッセージ (ソースコード 2.14 の一行目では sqrt) を呼ぶ式のことをメッセージ式という．Smalltalk のメッセージ式は，単行メッセージ，二項メッセージ，キーワードメッセージの三種類がある．単項メッセージはソースコード 2.14 の一行目のような引数を取らないメッセージ式のことをいい，二項メッセージはソースコード 2.14 の二行目のような一つの引数を取るメッセージ式のこという．キーワードメッセージはソースコード 2.14 の三行目のような，引数を取るメッセージ式のなかで二項メッセージ式以外の式のことを言う．それぞれの意味は，一行目は 9 の平方根を求めるプログラム，二行目は 1 と 2 を足すプログラム，三行目は 100 が 101 から 99 の範囲の中に存在するかを確認するプログラムとなっている．単項メッセージ式とキーワードメッセージ式を使用するとソースコード 2.13 のテストはソースコード 2.15 のように記述できる．メッセージ式を使用することで，括弧やブラケットの省略は可能であるが，代わりに :(コロン) が必要となり冗長さが残る．キーワードメッセージ式の終端記号の最後にはコロンが必要だからである．また，二項メッセージ式には使用することのできない記号があるなどの制約もある．

2.3.4　mixfix 演算子

mixfix 演算子 [21] は Coq[22] や Isabelle[23] 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 演算子は複数の名前部分と引数部分をもち，名前部分には任意文字列を使用可能であり，infix，prefix，postfix および closed という形式の演算子を作ることができる．infix とは中置のことで $\_\ op_0\ \_\ op_1\ \ldots\ \_\ op_n\_$ のように，名前部分 $op_i(0 \leq i \leq n)$ の前後に引数部分が存在する形式のことである．prefix とは前置のことで $\_\ op_0\ \_\ op_1\ \ldots\ \_\ op_n$ のように，名前部分 $op_i(0 \leq i \leq n)$ の前に常に引数部分が存在する形式のことである．postfix は 後置のことで，$op_0\ \_\ op_1\ \ldots\ op_n\ \_$ のように，名前部分 $op_i(0 \leq i \leq n)$ の後に常に引数部

ソースコード 2.16. mixfix 演算子の表現できる形式

```
1 _ ? _ : _
2 if _ then _ else _
3 _[ _ ]
4 [_]
```

ソースコード 2.17. mixfix 演算子でより簡潔になったテストの例

```
1 result should be 3
2 string should startWith regex "hel*o"
3 result should not be an Oranguatan
```

ソースコード 2.18. シェルスクリプトで cat を表した例

```
1 cat "hello.c" "world.c"
2
3 cat "hello.c", "world.c"
```

分が存在する形式のことである．closed とは infix とは逆に，op_0 _ op_1 ... _ op_n というように，引数部分の左右に常に名前部分 $op_i (0 \leq i \leq n)$ が存在する形式のことである．ソースコード 2.16 は mixfix 演算子の例である．一行目は infix で，二行目は prefix，三行目は postfix，四行目は closed の例である．mixfix 演算子は 2.3.2 項のユーザ定義の中置二項演算子よりも表現力の高い構文を定義可能である．mixfix 演算子を使用することで，ソースコード 2.13 はソースコード 2.17 のように簡潔に記述可能である．一行目や三行目のように括弧やブラケットを省略できるのは，be や an を prefix 演算子の形式で定義できるからである．

mixfix 演算子は非常に強力であり中置の二項演算子では表現できないような様々な構文を表現することが可能であり，プログラムをより自然言語に近い形で定義できる．しかし，この演算子は名前部分と引数部分が交互に並ばなければならないという制約があり簡潔さを損なう場面がある．たとえば，ソースコード 2.18 のように使用される演算子 cat _ _は引数部分が連続しているため定義できない．この演算子を定義するためには cat _, _のようにカンマを第一引数と第二引数の間においた postfix 形式の mixfix 演算子を定義しなければならない．しかしカンマを演算子に付与すると，構文はソースコード 2.18 の三行目のように冗長になってしまう．また，曖昧性の解決に型を使用しているので型が実行時につく動的型付け言語では mixfix 演算子が利用している手法を使用することができない．

2.3.5 ユーザが拡張可能なコンパイラ

SugarJ

SugarJ [9] 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．この拡張を使用することで，Java 上で XML リテラルなどの構文を表現することができる．構文拡張による曖昧性は型による解決をしており，曖昧性がある場合にはコンパイルエラーとなる．

ProteaJ

ProteaJ [10] 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という利点を持つ．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．

Wyvern

Wyvern[24] [25] は型を使用することで，任意の構文ユーザ定義リテラルを定義できるプログラミング言語である．しかし，構文ユーザ定義リテラルには制約があり， general literal form という特殊な構文の中でしか使うことができない．general literal form を使用しなければならず文法に大きな制約があるため，構築する DSL 構文の文法の自由度が下がる．Typed Syntax Macros[26] と呼ばれる型付きマクロのようなもの使用して Wyvern では定義できなかった部分を補完した機能もあるが，この機能も general literal form を使用しなければならないため定義できる構文の文法度に大きな制約がある．

2.3.6　プログラミング言語拡張についての研究

プログラミング言語を拡張する研究，その際に発生した曖昧性を回避しようとする研究は盛んに行われている．しかし，拡張構文の簡潔さ，曖昧性の回避手段を考慮していること，様々な言語で使用できるという項目をすべて満たした言語は我々の知る限りない．簡潔さとは構文を使用する際に特殊な記号や文字列を要求しないことで，より自然言語に近い構文を作ることができるということである．曖昧性の回避手段を考慮しているとは構文拡張時に発生する曖昧性を回避する手段を持っているということである．様々な言語で使用できるとうことは静的型付け言語と動的型付け言語の両方でその手法を使用できることである．表 2.3.6 はこれまで紹介した既存の言語機構や言語の特徴を示した表である．構文マクロ，SugarJ，ProteaJ 以外の言語機構は制約のある構文拡張しか許していなかったり，さまざまざ構文拡張を許していたとしてもその開始位置などに特殊な構文が必要だったりするため構文の簡潔さにかけるという問題がある．mixfix 演算子が △ である理由は，特殊な文字列などは必要ないが，演算子の名前部分と引数部分の並びの順番に制約があるからである．構文拡張時に発生する曖昧性を考慮しているものは ProteaJ と構文マクロのみでこれら以外の言語では曖昧性を発生しないものと考えていたり，発生するほどの構文拡張を許していなかったりする場合が多い．構文マクロが △ の理由は Schema の構文マクロは hygine であるため，ある程度の曖昧性は解消できるが構

文マクロ中で他の構文マクロを実行しようとすると意図しない動作になることがあり，完全に曖昧性回避を考慮しているとは言えないからである．また，演算子オーバーロードは曖昧性が発生するほど強力な拡張を行うことができないことからこの評価となった．表現力の高い簡潔な構文を定義できる言語は曖昧性の回避に型を使用している場合が多い．たとえば mixfix 演算子や ProteaJ は型を使用して曖昧性を回避している．そのため，これらの言語で使用されている手法は動的型付け言語では使用できない．他の動的型付け言語で使用できる手法は構文が簡潔ではなかったり，曖昧性の回避について考えていないことが多い．

	構文の簡潔さ	曖昧性回避を考慮	様々な言語で言語で使用可能
演算子オーバーロード	×	-	○
ユーザ定義の二項演算子	×	×	○
メッセージ式	×	○	○
mixfix 演算子	△	○	×
字句マクロ	×	×	○
構文マクロ	○	△	○
SugarJ	○	×	×
ProteaJ	○	○	×
Wyvern	×	×	×

表 2.1. 既存研究の特徴

第 3 章

Erbs

本研究では，構文拡張を行う際に発生する拡張構文同士やホスト言語と拡張構文同士の文法の衝突に留意して，Ruby 上で構文拡張が行える構文拡張機構 Erbs を提案する．Erbs は 2 章の最後で述べた構文の簡潔さ，曖昧性の回避手段を考慮していること，様々な言語で使用できるという構文拡張の実現を目指している．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．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，曖昧性の発生やユーザの意図しない動作を回避できる．また，Erbs では同一演算子に複数のタグをつけ，それらの論理演算をとることができる．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．

Erbs を用いると拡張した構文をもった DSL をライブラリまたは，フレームワークとして実現できる．Erbs を使用するユーザは，ライブラリやフレームワークを作成するユーザと，作成されたライブラリを使用してアプリケーションコードを書くエンドユーザである．つまり，ライブラリ作成者が Erbs の構文拡張機能を使用して演算子を定義し，エンドユーザがそれを使用する．エンドユーザは Erbs を直接使うことはないので，構文定義の詳細な理解をしなくとも表現力の高い DSL を使用できる．

本章では Erbs のアイデアとその例について述べ，その制限事項についても議論する．

3.1 Erbs 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おこなう．演算子は二項や三項演算子だけでなくそれ以上の項を持つ演算子も定義することが可能である．また，演算子の名前部分にはスペース以外のすべての文字列を使用することができる．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いう．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．演算子の引数部分の 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．

ソースコード 3.1. キーワード defs を使用した構文拡張

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

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．ソースコード 3.1 では defs を使用し言語拡張を行い シェルスクリプトの cat のような構文を実現している．ソースコード 3.1 の 六行目の 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，およびメソッドボディ部を持っている．構文定義部は ソースコード 3.1 の一行目の cat file1 file2 の部分で終端記号 cat の後に file1，file2 という二つの 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．引数定義部は一行目の (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 Ruby 式を指定することで，定義した演算子は演算子の名前部分が cat で，引数部分が file1 と file2 という解釈がされることになる．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二から三行目の 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．

3.2 タグを利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間で曖昧性が発生する．曖昧性が発生すると構文が壊れて，構文解析が失敗するべき構文が成功したり，逆に成功しなければならない構文が失敗するという危険がある．

3.2.1 ホスト言語との衝突回避手段

既存の構文と拡張構文との間に発生する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．Erbs ではユーザが定義した演算子にタグをつけることをできるようにした．演算子の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できるようする．

ソースコード 3.2 では checkout という git サブコマンドを Ruby 上に表現する例を考える．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

ソースコード 3.2. 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"
```

ソースコード 3.3. 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"
```

ができる．Erbs を使用しソースコード 3.2 の一行目の git bname と五行目の checkout cmd という演算子を定義することでこの機能を実現できる．一行目は演算子の名前部分が checkout で引数部分が bname である演算子である．この演算子は受け取ったブランチ名 (bname) を文字列 “checkout ” と足し合わせ，その結果を返す．五行目は演算子の名前部分が git で引数部分が cmd である演算子である．五行目で定義された演算子を実行すると受け取ったコマンド (cmd) を文字列 “git ” と足し合わせて，その結果をシェルに渡すことでブランチのチェックアウトを実現している．九行目は Ruby のメソッドとして定義した checkout メソットである．このメソッドは受け取った時間の文字列 (time) に対して文字列 “checkout at ” を足し合わせたものを返す Ruby のメソッドである．このように，Ruby のメソッド checkout と 演算子 checkout が定義してある場合，曖昧性が発生する．十三行目の checkout “master” を構文解析する際に，一行目の演算子である checkout か九行目の Ruby メソッドの checkout どちらを使用しすればよいか一意に決まらない．

このような既存の構文と拡張構文との間に発生する曖昧性を避けるために，Erbs では演

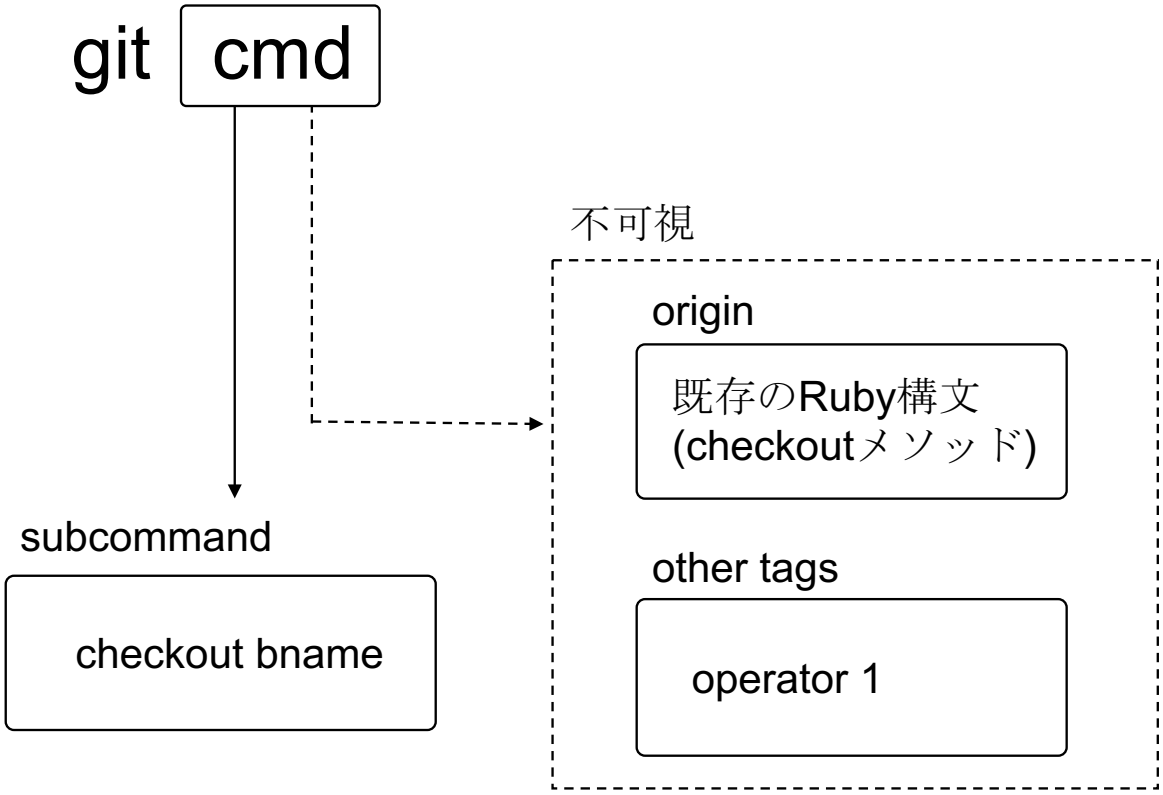

図 3.1. Erbs の引数部分の制限による可視部分と不可視部分

算子にタグをつけるようにした．タグはすべての演算子には一つ以上付いていて，アルファベットと数字を使用することができる．タグを演算子につけ，そのタグを使用して演算子を制限することで，定義した演算子の使用場所を制限することでき，曖昧性を回避につながる．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．ソースコード 3.3 は Operator を使用して演算子にタグをつけることでソースコード 3.2 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．ソースコード 3.3 の 一行目の (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．二から四行目はボディ部で，演算子を定義する部分である．演算子の定義は 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 の二つのタグをつけ，八行目の git cmd には origin と git というタグをつけた．Erbs では origin は特別なタグである．origin は既存の Ruby 構文についているタグである．ユーザが定義した演算子に対して origin というタグを付けるということは既存の Ruby 構文と同様に扱えるようにすることを意味している．

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．ソースコード 3.2 は git cmd の引数部分である cmd 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 ソースコード 3.3 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 (図 3.1)．git cmd には origin タグがついているため既存の Ruby 構文と同等に扱える．つまり，プログラム中で既存の Ruby 構文を書けるところであれば自由に git cmd 演算子を使用できるということである．逆に checkout bname には git と subcommand というタグしかついていないので，origin がついている演算子とは違い指定された特定の部分でのみ使用可能となる．

3.2.2　拡張同士の衝突の回避

Erbs を使用することで，構文拡張同士の回避できるように定義することも可能となる．拡張構文同士に発生する曖昧性を考える上で難しい点は，ある演算子を定義した時点で，第三者が更に拡張するので，拡張構文を定義した時点では今後どんな拡張が定義されるか予想できない点である．Erbs では defs の引数定義部で論理演算子 &&, ||, ! を用いて，タグ同士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ようにした．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 DSL を実装できる．

ソースコード 3.4 は構文拡張同士に発生する曖昧性の回避例である．一行目から五行目でタグ diff と option を持った演算子–help を定義している．この演算子は引数部分を持たないので 0 項の演算子で，文字列 “–help” を返す．七行目から十一行目ではタグ diff と origin を持った演算子 diff cmd を定義している．この演算子の引数は cmd で，この cmd に書くことができる Ruby 式の条件は diff && (subcommand || option) である．Erbs では && 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．また，|| はタグ同士の OR 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 または B のタグを持つ全ての Ruby 式である．つまり，この条件の意味は，タグ diff を持っていてかつ subcommand または option というタグを持っているすべての Ruby 式を受け付けるということになる．十三行目から十七行目 git と option を持った演算子–help を定義している．この演算子は引数部分を持たないので 0 項の演算子で，文字列 “–help” を返す．十九行目から二十三行目ではタグ diff と origin を持った演算子 git cmd を定義している．この演算子の引数は cmd で，この cmd に書くことができる Ruby 式の条件は git && (subcommand || option) である．この演算子の引数 cmd に来ることができる Ruby 式の条件はタグ git を持っていてかつ subcommand または option というタグを持っているすべての Ruby 式ということになる．

もし論理演算が使用できない場合は，ソースコード 3.5 のように，diff cmd の引数部分の cmd にはタグ diff 一つしか指定することができない．このような場合に，演算子 –help と 演算子 diff cmd を定義した DSL 開発者以外の開発者が，既存の演算子定義の存在を知らずに演算子–help と 演算子 git cmd を定義した場合，曖昧性が発生し正しく構文解析できない．しかし，Erbs ではタグ同士の論理演算を使用することができるため，同じ–help という演算子の使

ソースコード 3.4. 拡張同士の衝突

```
1  Operator(diff, option)
2    defs --help ()
3      "--help"
4    end
5  end
6
7  Operator(diff, origin)
8    defs diff cmd (cmd: diff && (subcommand || option))
9      "diff " + opt
10   end
11 end
12
13 Operator(git, option)
14   defs --help ()
15     "--help"
16   end
17 end
18
19 Operator(git, origin)
20   defs git cmd (cmd: git && (subcommand || option))
21     exec("git " + cmd)
22   end
23 end
24
25 diff --help
26 git --help
```

用するコンテキストを詳細に制限できる．そのため，曖昧性の発生を防ぐことができる．(図 3.2)

3.3 再利用可能性の高い拡張

Erbs では複数のタグ，タグ同士の論理演算，@token という述語を使用することで，再利用可能性の高い構文拡張をすることができる．ソースコード 3.6 は @token，複数のタグおよび述語を使用した DSL の実装例である．この例はソースコード 3.1 の出力先を標準出力からファイルに変更するものである．ここでは一行目から六行目で 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．八から一六行目では 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することができる．

@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

ソースコード 3.5. 拡張同士の衝突

```
1  Operator(diff, option)
2    defs --help ()
3      "--help"
4    end
5  end
6
7  Operator(diff, origin)
8    defs diff cmd (cmd: diff)
9      "diff " + opt
10   end
11 end
12
13 diff --help
```

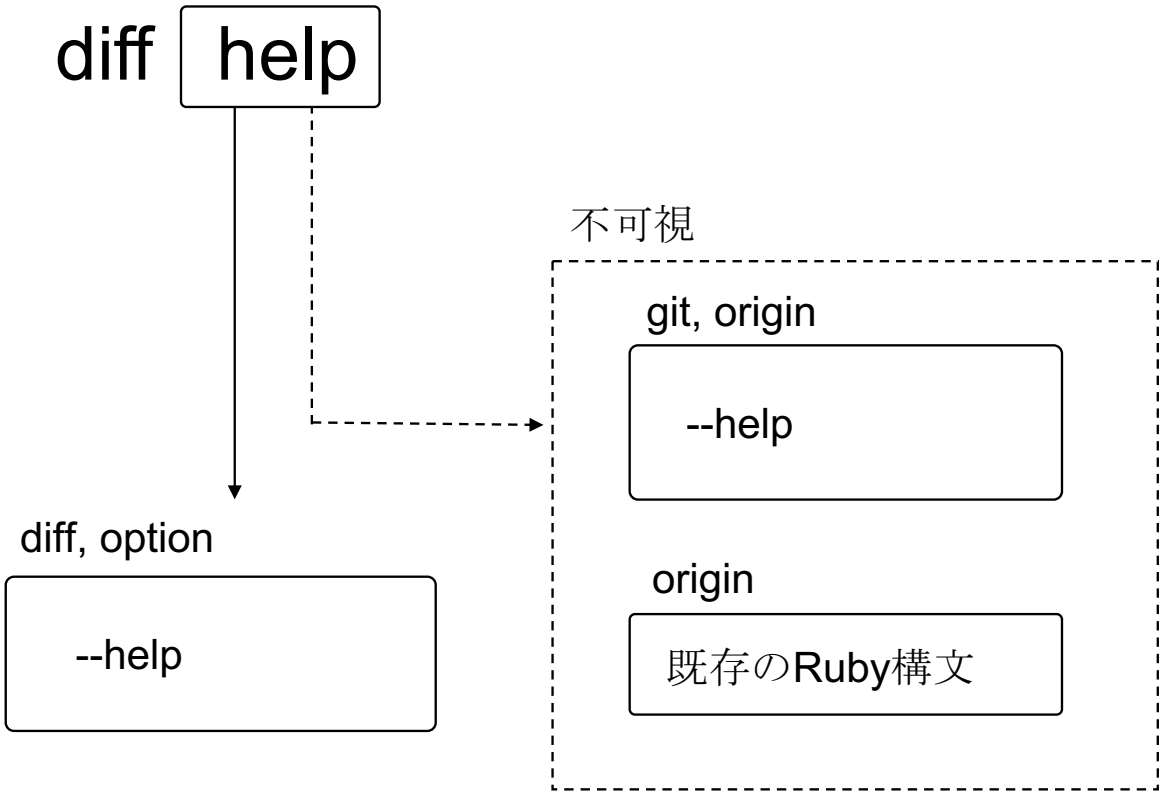

図 3.2. 拡張間同士の曖昧性を回避するための引数部分の制限

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 ソースコード 3.6 の十二行目の 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

ソースコード 3.6. モジュラーな拡張の例

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

> 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 ソースコード 3.7 のように実装することで，既存の ソースコード 3.6 を修正することなく，そのまま再利用して >> を実装できる．ソースコード 3.6 の二行目からの 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，一二行目に分離して書かれている．この分離により，ソースコード 3.6 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．ソースコード 3.6 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，ソースコード 3.6 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，ソースコード 3.6 の十一行目とソースコード 3.6 の二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．

3.4 Erbs の制限

Erbs は構文的な制限事項を持っている．以下ではそれらについて議論する．

ソースコード 3.7. 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"
```

ソースコード 3.8. proteaj が定義できる演算子 [10] より引用

```
1  Regex reg = hel+o; // helo, hello, helllo, ...
```

ソースコード 3.9. Erbs で構築した正規表現 DSL

```
1  reg = hel + o
```

名前部分と引数部分のセパレータの制限

定義する演算子の名前部分と引数部分の間にスペースが必要という制限がある．例えば Erbs でソースコード 3.8 のような he という文字列の後に l が一回以上続き，その後 o という文字列が続くという正規表現を表現する DSL を作成することはできない．これは，Erbs ではこのような演算子を表そうとすると hel と + の間 と + と o の間にスペースが必要になり，ソースコード 3.9 のようになってしまう．これは文法上大きな制約になになり，正規表現を表していることがわかりづらくなってしまう．ProteaJ の場合はこの制限がないため，ソースコード 3.8 というような DSL を図 3.3 というような演算子を定義することで作成することができる．この図は (A) で引数部が二つ並んでいる Regex 型の演算子を定義している．(A) の引数部分には Regex 型の演算子を許している．そして (C) で引数部のあとに + が来る Regex 型の演算子を定義している．(C) の引数部分は Regex 型 を許している．(D) では引数部分のみの Regex 型の演算子を定義している．(D) の引数部分は Letter 型の演算子を許す．そして，文字列 a から z をそれぞれを引数なしの Letter 型の演算子として定義している．この定義によって，hel というのは演算子 (A) と演算子 h，e と l を使用することで表現できる事がわかる．

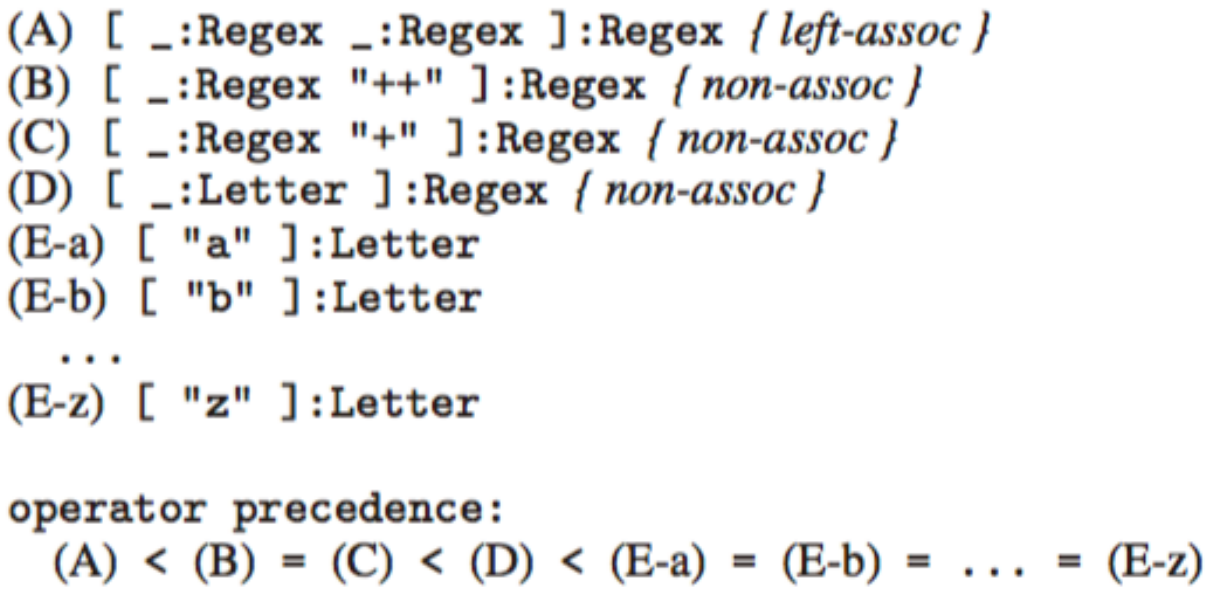

図 3.3. ProteaJ における正規表現リテラルの定義 ([10] の図 4 より引用)

曖昧性回避の制限

Erbs は曖昧性の回避手段を提供しただけで，Erbs を使用することで完全に曖昧性を回避することは不可能である．さきで述べた複数タグや論理演算などの機能を使用して，DSL 開発者が正しく設計をした場合に初めて拡張構文同士の曖昧性を回避した再利用可能性の高い拡張構文を作成することができる．Erbs はタグを演算子につけて，引数部分に来ることを許す演算子を減らすことで曖昧性を回避していので，ユーザがすべての演算子に origin タグをつけたり，すべての演算子に同じタグをつけた場合は曖昧性の発生を回避するのは困難となる．たとえば，ソースコード 3.4 を変更して，ソースコード 3.10 のようにすべての演算子に origin タグを一つだけつけた場合，これらの演算子はすべて既存の Ruby 構文として扱われる．そのため，構文解析の時すべての演算子が可視な状態となってしまい曖昧性が発生し意図しない挙動を引き起こす．この問題は Erbs だけでなく，曖昧性の回避に型を使用した方法を取っている言語拡張でも起こる問題である．たとえば，型を使用してる場合にはすべて Any 型や Object 型などのすべての型を許すような型をしようする場合にはこの問題と同じ問題が発生する．このような曖昧性の回避手段を持っている言語を使用する際は，DSL 開発者の能力の大きく依存する．

@token の制限

Erbs は @token を使用することで引数に指定した終端記号を式中に含むすべての Ruby 式がくるという指定をすることができるが，指定できる終端記号には制限がある．指定できる終端記号は基本的な演算子と，ユーザが新たに定義した演算子のみがである．@token の指定を

ソースコード 3.10. 曖昧性の発生する例

```
1  Operator(origin)
2    defs --help ()
3      "--help"
4    end
5  end
6
7  Operator(origin)
8    defs diff cmd (cmd: origin)
9      "diff " + opt
10   end
11 end
12
13 Operator(origin)
14   defs --help ()
15     "--help"
16   end
17 end
18
19 Operator(origin)
20   defs git cmd (cmd: origin)
21     exec("git " + cmd)
22   end
23 end
24
25 diff --help
26 git --help
```

サポートしている演算子は，+，−，∗，/，&&，||，>=，>，<=，< である．この制限は，Erbs 上で，ホストの Ruby 式と，ユーザ定義の演算子の扱いを分けているため発生する．そのため，すべての組み込み Ruby 式を Erbs の演算子定義機能を使用して作成すればこのような制限はなくなる．

第 4 章

実装

本研究では，3 章で提案した Erbs のアイデアを実現するためソースコード変換器を実装した．この変換器は Ruby のサブセットの構文とユーザ定義演算子を純粋な Ruby へと変換する．その後変換された純粋な Ruby のソースコードを Ruby の処理系に渡して，ソースコードを実行している．実行する際に使用した Ruby は C 言語で実装された処理系である CRuby(MRI) を使用した．本章では Erbs の実装方法の詳細について述べる．また，どのようにユーザが書いたソースコードが Ruby のコードに変換されるかについて述べる．

4.1 実装したソースコード変換器

Erbs は図 4.1 のように，ユーザが書いた Ruby のサブセットと演算子を純粋な Ruby のコードに変換し，そのソースコードを Ruby の処理系で実行する．以降変換後の純粋な Ruby のことを CRuby と呼ぶことにする．

引数が “pen” かどうかを判定する演算子をソースコード 4.1 に示す．このプログラムを Erbs を使用して CRuby のコードに変換するとソースコードと 4.2 となる．演算子定義部分は Ruby のメソッド定義に変換される．ソースコード 4.1 では，二行目の ls this a x ? という演算子をソースコード 4.2 の三行目の op_ls_this_a_X_63 というメソッドに変換している．ソースコード 4.2 の三行目の self とは CRuby 特殊な構文である．このキーワードをメソッド名の前につけてメソッドを定義すると，クラスが一つだけ持っている特異メソッドにメソッドを定義するというになる．このように定義されたメソッドは明示的なレシーバを必要とせず OriginPen::op_ls_this_a_X_63 というようにメソッドを実行できる．変換後の CRuby のコードもある程度可読性が必要なのでこのような構文を使用している．メソッド名は最初に op_ というプレフィックスのあとに演算子の名前部分と引数部分をアンダースコアで繋いだものをメソッド名としている．どの部分が引数かわかるように引数部分の文字列は大文字に変換している．また，Ruby のメソッド名は末尾に!，? を使えることを除いて数字と文字列しか使えため，数字と a から z および，大文字の A から Z までの文字列以外はアスキーコードに変換してメソッド名として使用している．定義したメソッドはそれぞれのタグのキャメルケースでつないだクラスの中に定義している．ソースコード 4.2 はもと演算子に origin と pen

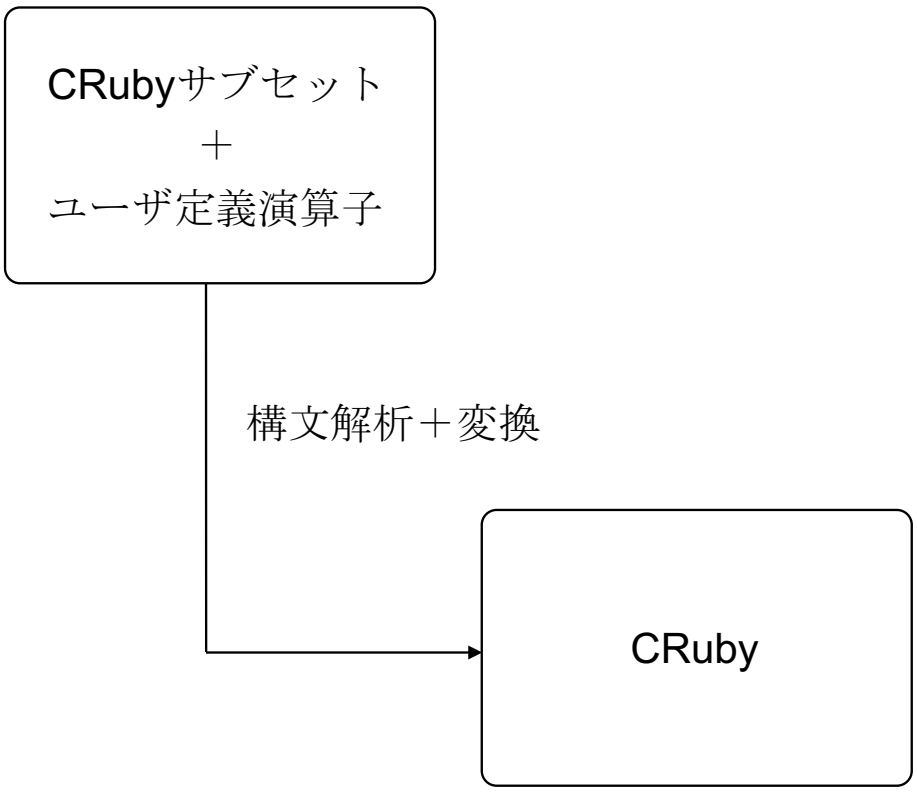

図 4.1. のソースコード変換器

というタグが付いているので CRuby コードに変換する際には，OriginPen というクラスを作成し，そのクラスの中にメソッドを定義している．変換後のクラスが CRuby でもともと定義されている他のクラスやメソッドとの衝突を避けるために，Operator というモジュールの中に変換されたメソッドは定義されている．CRuby はモジュールを定義するときには module というキーワードを使用する．ソースコード 4.2 の三行目から五行目の演算子のボディ部である．もしこの部分に演算子が存在する場合にはその演算子を更に展開し，それ以外はそのままメソッドボディとして使用する．演算子呼び出しは CRuby のメソッド呼び出しに変換される．ソースコード 4.1 の九行目の Is this a “pen” はソースコード 4.2 の十一行目の Operator::OriginPen::op_Is_this_a_X_63(“pen”) に変換される．Ruby の処理系には JRuby や mrubys 等様々な処理系が存在するが，本研究では公式処理系である MRI(Matz ’ Ruby Implementation) を使用している．使用したバージョン Ruby2.3.1(r54768) である．

Erbs がサポートしている Ruby のサブセットの構文はクラス，モジュール，メソッド定義および実行，インスタンス変数，グローバル変数，ローカル変数の定義，if，unless，while や until などの制御構文，基本的な二項演算子などである．まだ実装が完了していない構文として case 式，後置の while や for 式等がある．実際に構文解析に成功するソースコード例を付録のソースコード A.1 に示した．

本変換器は Scala を使用して記述し，構文解析器をパーザーコンビネータ [27] を使用して作成した．Scala のパーザコンビネータライブラリは Packrat Parsing[28] を採用している．

ソースコード 4.1. 変換前のソースコード

```
1  Operator(origin, pen)
2    defs Is this a x ? (x: origin)
3      if x == "pen"
4        puts "yes"
5      else
6        puts "no"
7      end
8    end
9  end
10
11 Is this a "pen" ?
```

ソースコード 4.2. 変換後のソースコード

```
1  module Operator
2    class OriginPen
3      def self.op_Is_this_a_X_63(x)
4        if x == "pen"
5          puts "yes"
6        else
7          puts "no"
8        end
9      end
10   end
11 end
12
13 Operator::OriginPen::op_Is_this_a_X_63("pen") # "yes"
```

パーザコンビネータとはパーザを引数にプログラムの文字列を受けとり，構文解析を返す関数とみて，小さなパーザを組み合わせてパーザを組み立て手法のことをいう．Scala の実装だけであんく Haskell の Parsec などのライブラリもある．このライブラリは抽象構文木の最上位から再開に向かって進みながら抽象構文木を作成しながら構文解析を勧めていくトップダウン構文解析を行っている．逆に，最下位から最上位に向かって構文解析を勧めてい行く方法をボトムアップ構文解析という．Packrat Parsing とはトップダウン構文解析にメモ化を組み合わせた構文解析に手法で，メモ化したことでバックトラックをした際に同じ構文解析を二度行うことがなくなるので，通常指数時間かかる構文解析時間を線形時間で行うことができる．通常，Packrat Parsing は左再帰を扱うことができないが [29] の手法を使用することで左再帰を扱うことができる Packrat Parser を構築できる．このライブラリは [29] の手法を実装しているため左再帰を持っている構文の構文解析にも成功する既存の 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 bison[30] というパーザジェネレータで作られており，実行時に構文解析器を拡張しながら構文解析を勧めていくことは困難であるためである．ソースコードのコード行数は 2667 行で，Ruby の構文解析器部分は 163 行である．

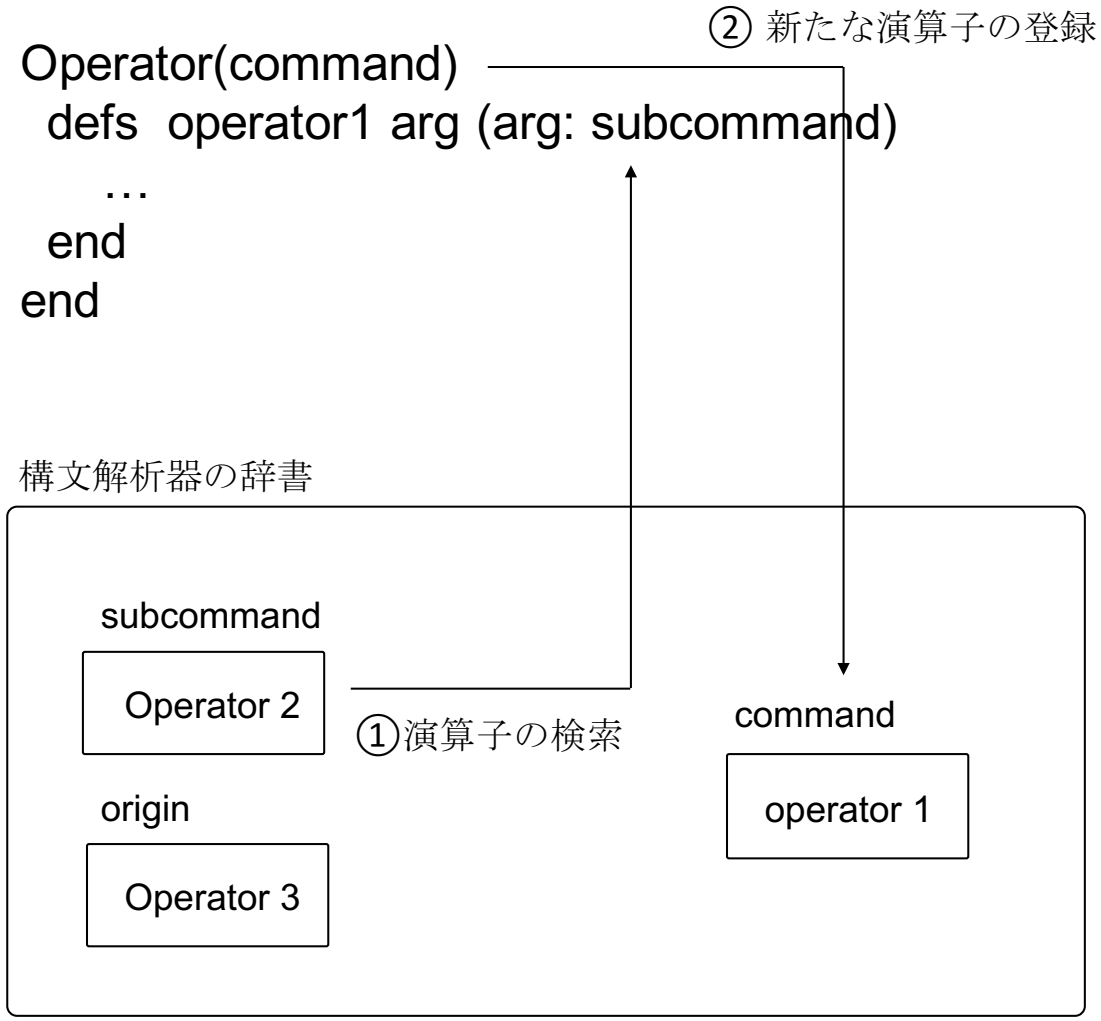

図 4.2. 構文解析の流れ

4.2　処理の流れ

ユーザが記述した拡張構文と Ruby サブセットを含んだソースコードがどのように処理されていくかの実装について述べる．演算子を定義してから実装される流れは 4 章で述べたとおりである．実際に Erbs がどのようなアルゴリズムで構文拡張をし，構文解析をすすめていくかについて以下で述べる．

おおまかな処理の流れは図 4.2 のように，ユーザが新たに演算子を定義すると，ユーザが定義した構文解析器の辞書から演算子を表す構文解析器を作成し，既存の構文ルールに新たな構文ルールを追加する．そして，使用部分では定義した構文ルールの中から適切な構文を探し出しその構文を使用して構文解析をすすめる．

1. defs キーワードを使用して演算子 op を定義する．

(a) 演算子 op の構文を表す，構文解析器をパーザーコンビネータを使用して組み立てる．ユーザが定義した演算子が定義されている構文解析器の辞書 dict から op の引数部に必要な演算子を探し出す．
   i. 引数部には論理演算子を使用できるので，dict から演算子を表す構文解析器を探し出す際は op の引数部で定義されたタグの論理演算子を選言標準形に変換する．
   ii. 選言標準形に変形したタグの項それぞれに対して dict から演算子を探し出し，最後に探し出した演算子つなぎ合わせ，大きな構文解析器を組み立てる．
   iii. 必要な演算子がなければエラーとする．

(b) 組み立てた構文解析器 op のタグの定義にタグ origin が含まれている場合，Ruby サブセットの構文に op を追加する．op に origin が含まれていない場合は何もしない．

(c) 定義した演算子 op は構文解析器の辞書 dict に登録される．この辞書のキーはタグをの集合であり，値は定義した op のリストである．値が op のリストであるのは，同じキー値に対して異なる演算子が定義されることがあるからである．

2. 演算子呼び出しが行われたときは，定義されている演算子を適用する．
3. すべての構文解析が成功し，構文木が完成すると，その構文木を PrettyPrinter に入力としてわたす．PrettyPrinter が構文木をもとに Ruby のコードを作成する．
4. 作成された Ruby コードを Temporary ファイルに出力し，そのファイルを Ruby 処理系で実行することでプログラムの出力を得る．

第 5 章

ケーススタディ

本章では Erbs を用いて実装した DSL を紹介する．また，実際のすでに使われている DSL を Erbs で表現できることを示す．以下で紹介するプログラムは，定義部分と使用部分のユーザが違う．定義部分はライブラリやフレームワークを作成するユーザが記述し，使用部分はエンドユーザが記述している．そのため，実際のユースケースでは，定義部分はライブラリとしてパッケージングされ，ライブラリを使用するエンドユーザに提供されることを想定している．まず，Erbs で N 項演算子の定義方法を示す．その後 Mixfix 演算子の prefix と postfix 形式の演算子の実現方法を示す．そしてシェルスクリプトのパイプを Erbs で実現する例示す．最後に terrafor というインフラ環境の構築，変更を自動化できるツールの DSL を Erbs で実現する方法について示す．

5.1 様々な形の演算子

ソースコード 5.1 には N 項の演算子，Mixfix 演算子が実現していた prefix 形式の演算子，postfix 形式の演算子を定義している．一から十六行目までで N 項演算子の定義し使用している．この演算子を使用して 1 から 10 までの数の加算を計算している．この例では１０までの引数しか渡していないが Ruby の処理系が許す限りの数の引数を書くことができる．Erbs は二行目から十行目のように演算子定義の引数部分にその演算子が持っているタグを定義する再帰的な演算子の定義を許している．このように定義することで引数が N 項続く演算子を表すことができる．十八から二十九行目は Mixfix 演算子の Prefix 形式の演算子定義とその使用部分である．この演算子を使用して Ruby には存在しない三項演算子を実現している．Prefix 形式とは $\_\ op_0\ \_\ op_1\ \ldots\ \_\ op_n$ という文法をもった演算子である．Mixfix 演算子とは違い，Erbs は名前部分と引数部分の順番に制限はないのでこれ以上の表現力を持った演算子を定義することが可能である．十八から二十九行目は Mixfix 演算子の Postfix 形式の演算子定義とその使用部分である．Postfix 形式とは $\_\ op_0\ \_\ op_1\ \ldots\ \_\ op_n$ という文法を持ったもった演算子である．この演算子を使用して Ruby の配列の要素にアクセスするような演算子を作成した．

ソースコード 5.1. N 項演算子，mixfix 演算子の prefix，postfix 形式の定義

```
1  # N項の演算子
2  Operator(args)
3    defs a (a: origin)
4     a
5    end
6
7    defs a b (a: origin, b: args)
8      a + b
9    end
10 end
11 Operator(origin)
12   defs sum argument (argument: args)
13     argument
14   end
15 end
16 sum 1 2 3 4 5 6 7 8 9 10 # => 55
17
18 # Mixifix 演算子の prefix 形式の演算子
19 Operator(origin)
20   defs cond ? t_body : f_body (
21     cond: origin, t_body: origin, f_body: origin)
22     if cond
23       t_body
24     else
25       f_body
26     end
27   end
28 end
29 true ? "true" : "false" # => true
30
31 # Mixifix 演算子 の postfix 形式の演算子
32 Operator(origin)
33   defs ary $ idx $ (ary: origin, idx: origin)
34    ary[idx]
35   end
36 end
37 a = [1, 2, 3]
38 a$2$ # => 3
```

5.2 シェルスクリプトのパイプ

Erbs でシェルスクリプトのパイプを表した例をソースコード 5.2 に示す．このコードは .gitignore のファイルの中身を次のパイプに渡して，受け取った先で “ensim_cache” という文字列があれば，その行を標準出力に出力するというコードである．このコードでは cat filename ， grep pat，cmd1 | cmd2 という三つの演算子を定義している．演算子 cat filename

ソースコード 5.2. Erbs を使用したシェルのパイプ定義

```
1  Operator(command, origin)
2    defs cat filename(filename: origin)
3      Proc.new do
4        File.read(filename).split("\n")
5      end
6    end
7
8    defs grep pat (pat: origin)
9       Proc.new do |lst|
10        lst.select { |f| f.include?(pat) }
11      end
12   end
13 end
14
15 Operator(origin)
16   defs cmd1 | cmd2 (cmd1: command && !@token(|), cmd2: origin)
17     out1 = cmd1.call
18     out2 = cmd2.call(out1)
19     puts out2
20   end
21 end
22
23 cat ".gitignore" | grep "ensime_cache"
```

, grep pat にはタグ command と origin を付与し，演算子 cmd1 | cmd2 には origin というタグを付与している．演算子 cmd1 | cmd2 の引数は cmd1 にはタグ command でかつ |(パイプ) を含んでいない演算子に制限している．このように cmd1 に許す構文を制限することで，曖昧性を回避できる回避できる．

5.3　Terraform

Terraform [31] というインフラ環境の構築，変更，バージョニングを安全に効率良く行うことのできるツールがある．昨今，Infrasturcutre as a code という，インフラ環境構築をコードに落とし込み，インフラ環境の構築，変更をトラッキングできるようにするという動きがある．Infrasturcutre as a code を実現するために Terrform の他にも Chef，Ansible などのツールがある．Chef，Ansible などの構成管理ツールはサーバの中でどのアプリケーションをインストールするか，どのファイルをダウンロードするかということををコードとして記述できる，一方 Terraform はどのプラットフォームのサーバ (Heroku，Docker コンテナ，EC2 のコンテナ，OpenStack など) を立ち上げるかなどの構成管理をすることができる．

Terraform を使用することで，Amazon Elastic Compute Cloud (以下，EC2) のサーバのリソースを決めて，コンテナの立ち上げる部分をコードとして書いて自動化することができる．EC2 とは，AWS (Amazon webserverice) が提供しているクラウド上で CPU やメモリ

ソースコード 5.3. EC2 インスタンスを作成すること Terrafor のコード

```
1  provider "aws" {
2    access_key = "anaccesskey"
3    secret_key = "asecretkey"
4    region = "us-west-2"
5  }
6
7  resource "aws_instance" "web" {
8    ami = "${data.aws_ami.ubuntu.id}"
9    instance_type = "t2.micro"
10 }
```

ソースコード 5.4. Erbs で定義した Terrform 構文の使用部分

```
1  provider docker {
2    host = "tcp://127.0.0.1:2376/"
3  }
4
5  resource docker_container "foo" {
6    image = "${docker_image.ubuntu.latest}"
7    name = "foo"
8  }
9
10 resource docker_image "ubuntu" {
11   name = "ubuntu:latest"
12 }
```

サイズなどが変更が可能なサーバを提供するサービスがある．ソースコード 5.3 は EC2 インスタンスを一つ作成するコードの例である．一行目の provider から始めるコードブロックで，どのプラットフォームを使用するか決定する．この例では “aws” と書いてあり AWS のサービスに対する設定をすべて記述している．三，四行目の access_key と secret_key は AWS を使用する際のユーザ認証に使用するデータである．四行目の “us-west-2” の部分はサーバを設置する地域を設定する部分である．五行目の キーワード resource で始まるコードブロックで，AWS 上のどのサービスを使用し，またその設定はどういったものかということを指定する．resource のあとに続く “aws_instance” が EC2 であることを表している．次に続く “web” はユーザがどの用途に使用するサーバかを記述する部分である．EC2 にはコンテナごとに一意な ID がついており，六行目の ami の部分はその ID を指定している部分である．七行目の instance_type とはコンテナのリソースのタイプをしている．リソースタイプとはどの CPU を使いメモリはどれかくらいかということを指定できる．Terraform を使用することで，本来であれすべて WEB サイト上で手動で行わなければならない作業を自動化することができる．

Terraform の DSL を Erbs で実現するとソースコード 5.4 のようになる．このように実行するためにはソースコード 5.5 のようなという演算子の定義が必要である．ただし実際に環境を構築する部分のコードを記述すると冗長になるので，今回はコメントでどのような処理を行

うかを示した．ソースコード 5.5 の一から一七行目で docker についての provider を構文解析する演算子を定義している．docker provider は最低限 host を設定すれば良いので，設定項目として二行目の host = 演算子を定義した．この定義によりソースコード 5.4 の一から三行目を実現することができる．これは docker がどのポートを使用してホストマシンと通信をするかという設定を行っている．ソースコード 5.5 の一九から五六行目で docker provider 用の resouce を定義している．これらの定義により 5.4 の五から十二行目を実現している．ここでは，docker コンテナにどのイメージを使うか設定している．

ソースコード 5.5. Erbs で定義した Terrform 構文の定義部分

```
1  Operator(terraform, docker_provider)
2    defs host = e (e: origin)
3      # host設定
4    end
5  end
6
7  Operator(terraform, provider)
8    defs docker { body } (body: terraform && docker_provider)
9      # dockerの設定
10   end
11 end
12
13 Operator(origin, terraform)
14   defs provider configs (configs: terraform && provider)
15     # providerの設定
16   end
17 end
18
19 Operator(terraform, docker_resource)
20   defs image = path (path: origin)
21     # image typeを設定
22   end
23
24   defs name = n (n: origin)
25     # コンテナの name 設定
26   end
27 end
28
29 Operator(terraform, docker_resource_multiple)
30   defs a (a: terraform && docker_resource)
31     # 一つの設定を記述できる
32   end
33
34   defs a b (a: terraform && docker_resource,
35             b: terraform && docker_resource_multiple)
36     # 二つの設定を記述できる
37   end
38 end
39
40 Operator(terraform, resource)
41   defs docker_container name { body } (
```

```
42      name: origin, body: terraform && docker_resource_multiple)
43      # docker_conitanerの設定
44    end
45
46    defs docker_image name { body } (
47      name: origin, body: terraform && docker_resource_multiple)
48      # docker_image の設定
49    end
50  end
51
52  Operator(origin, resource)
53    defs resource configs (configs: terraform && resource)
54      # resouceの設定
55    end
56  end
```

ソースコード 5.6. Erbs で定義した Terraform 構文の GCP の定義

```
1   Operator(terraform, google_provider)
2     defs credentials = e (e: origin)
3       # host設定
4     end
5
6     defs project = e (e: origin)
7       # host設定
8     end
9
10    defs region = e (e: origin)
11      # host設定
12    end
13  end
14
15  Operator(terraform, provider)
16    defs google { body } (body: terraform && google_provider)
17      # googleの設定
18    end
19  nend
20
21  Operator(terraform, resource)
22    defs google_compute_instance name { body } (
23      name: origin, body: terraform && google_resouce_multiple)
24       # google compute instanceの設定
25    end
26  end
```

Erbs を使う利点は，terraform の構文で使用していた文字列を減らすことができることである．ソースコード 5.4 の五行目，一〇行目の docker_container と docker_image と言うのは本来は文字列で書かなければいけない．Erbs ではこの部分を演算子を用いて定義することで，実行時にしかわからなかった文字列の書き間違いなどを，構文解析時に判定し，事前にエラーを出すことができる．Erbs を使用することで構文解析のフェーズで resouce の設定項目の中で不足してる項目を検出することができる．Terrform は，resource ごとに設定の項目が違っ

ており．たとえば，aws_instance であれば，ami と instance_type という設定項目は必須であるが，docker_container の場合は image と name が必須の設定項目となっている．このように，resoruce により必要な設定項目が異なっている．terrform を使用すると実行してみて初めて足りない項目がわかるが，Erbs を使用してそれぞれの構文解析を作成することで，構文解析時にエラーを見つけることができる．必須項目を実行時ではなく構文解析時にバリデーションできるということ非常に重要なことである．Erbs は複数のタグを使用できるので既存の定義をそのまま再利用して，新たな設定項目を追加することができる．例えば，新たな resource として Google Colud Platform を追加したいとする．この時，ソースコード 5.6 と定義すると既存のコードを再利用しつつ新たに Google Cloud Platform の設定も記述することができる．

第 6 章

まとめと今後の課題

6.1 まとめ

本研究では，構文拡張を行う際に発生する拡張構文同士やホスト言語と拡張構文同士の文法の衝突に留意して，Ruby 上で構文拡張が行える構文拡張機構 Erbs を提案した．Erbs が定義できる演算子は，名前部分と引数部分が任意の順番で使用できる．Erbs を使用することで，Ruby 構文では表すことのできない構文を簡潔に定義および使用できること，構文を拡張した際に発生する曖昧性を回避手段を提供していること，動的および静的型付け言語の両方で使用できることを示した．

言語拡張をする際に既存の構文と拡張構文との間または，拡張構文間に曖昧性発生が発生して構文解析エラーになったり，構文解析エラーになるべき構文が成功してしまうなどの問題が発生する．そのため，構文拡張をする際には，曖昧性の発生の回避手段を持つことは重要であることを述べた．既存の構文拡張手法では，曖昧性の回避方法として文字列によるパターンマッチ，構文木によるパターンマッチ，型を使用するなどの回避方法を提供していた．しかし，文字列や構文木では機構が単純すぎて曖昧性が発生しやすかったり，構文木を意識してプログラムを書かないといけないなどの問題があった．また，型を使用した曖昧性の回避方法は動的型付け言語では使用できないという問題もあった．そこで，本研究ではすべての Ruby 式に対してタグを付与し，定義した演算子の引数部分では特定のタグを持つ 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，曖昧性の発生やユーザの意図しない動作を回避できようにしている．さらに，Erbs では同一演算子に複数のタグをつけ，それらの論理演算をとることができる．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．この機能があることによって，演算子を定義した後に，第三者が既存の演算子の定義を壊さないようにしていることについての述べた．また，Erbs の制限についても述べた．更に我々は Erbs を実装し，実際にこのアイデアが機能するか確認した．Erbs がどのような演算子を実現できるか示すために様々な例を使用して紹介した．

6.2　今後の課題

今後の課題としては，Erbs が持っているセパレータの制限をなくすこと．また，式よりも更に細かく構文拡張を行えるようにすることなどがある．また，タグはユーザが今後のことを考えて適切に設計しなければならないので，タグ以外にも設計が簡単になるような物があれば導入する必要がある．

発表文献と研究活動

(1) 岩間雄太，市川和央，千葉滋．動的型付き言語上での構文拡張手段の提供．日本ソフトウェア科学会第 32 回大会，2016.9.6-9.
(2) 岩間雄太，市川和央，千葉滋．動的型付け言語における構文拡張による曖昧性の回避手段の提供．情報処理学会 第 78 回全国大会，2016.3.
(3) 岩間雄太，市川和央，千葉滋．動的型付き言語上での構文拡張により発生する曖昧性の回避について．第 18 回プログラミングおよびプログラミング言語ワークショップ (ポスター)

参考文献

[1] Martin Fowler. Internaldslstyle. `http://martinfowler.com/bliki/InternalDslStyle.html`, 2006.

[2] David Heinemeier Hansson. Ruby on rails. `http://rubyonrails.org/`, 2005.

[3] Yukio Matsumoto and K Ishituka. Ruby programming language, 2002.

[4] Martin Odersky. The scala language specification. 2014.

[5] Martin P Ward. Language-oriented programming. *Software-Concepts and Tools*, Vol. 15, No. 4, pp. 147–161, 1994.

[6] Stephen C Johnson. *Yacc: Yet another compiler-compiler*, Vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[7] Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. An idealized metaml: Simpler, and more expressive. In *European Symposium on Programming*, pp. 193–207. Springer, 1999.

[8] Kevin Hammond, Jost Berthold, and Rita Loogen. Automatic skeletons in template haskell. *Parallel processing letters*, Vol. 13, No. 03, pp. 413–424, 2003.

[9] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: Library-based syntactic language extensibility. In *Proc. of OOPSLA 2011*, pp. 391–406. ACM, 2011.

[10] Kazuhiro Ichikawa and Shigeru Chiba. Composable user-defined operators that can express user-defined literals. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pp. 13–24, New York, NY, USA, 2014. ACM.

[11] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, Vol. 35, No. 6, pp. 26–36, June 2000.

[12] Arie van Deursen and Paul Klint. Little languages: Little maintenance. *Journal of Software Maintenance*, Vol. 10, No. 2, pp. 75–92, March 1998.

[13] R. M. Herndon, Jr. and V. A. Berzins. The realizable benefits of a language prototyping language. *IEEE Trans. Softw. Eng.*, Vol. 14, No. 6, pp. 803–809, June 1988.

[14] Jon Bentley. Programming pearls: Little languages. *Commun. ACM*, Vol. 29, No. 8, pp. 711–721, August 1986.

[15] John Ellson, Emden Gansner, Eleftherios Koutsofios, Stephen North, and G Wood-

hull. Graphviz. *URL: http://www. research. att. com/sw/tools/graphviz*, 1998.

[16] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pp. 151–161, New York, NY, USA, 1986. ACM.

[17] Jon Rafkind and Matthew Flatt. Honu: syntactic extension for algebraic notation through enforestation. In *ACM SIGPLAN Notices*, Vol. 48, pp. 122–131. ACM, 2012.

[18] 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, p. 6p, sep 2015.

[19] Jean D Ichbiah, Robert Firth, Paul N Hilfinger, Olivier Roubine, Mike Woodger, John GP Barnes, Jean-Raymond Abrial, Jean-Loup Gailly, Jean-Claude Heliard, Henry F Ledgard, et al. *Reference manual for the Ada programming language*. Castle House Publications Limited, 1983.

[20] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[21] Nils Anders Danielsson and Ulf Norell. Parsing mixfix operators. In *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages*, IFL'08, pp. 80–99, Berlin, Heidelberg, 2011. Springer-Verlag.

[22] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 8.6*. PhD thesis, Inria, 2016.

[23] Lawrence C Paulson, et al. *The Isabelle reference manual*. Citeseer, 2016.

[24] Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Wyvern: A simple, typed, and pure object-oriented language. In *Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, pp. 9–16. ACM, 2013.

[25] Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In *European Conference on Object-Oriented Programming*, pp. 105–130. Springer, 2014.

[26] Cyrus Omar, Chenglong Wang, and Jonathan Aldrich. Composable and hygienic typed syntax macros. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pp. 1986–1991. ACM, 2015.

[27] Nils Anders Danielsson and Thorsten Altenkirch. Mixing induction and coinduction. *Draft paper*, 2009.

[28] Bryan Ford. Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl. *SIGPLAN Not.*, Vol. 37, No. 9, pp. 36–47, September 2002.

[29] Alessandro Warth, James R Douglass, and Todd D Millstein. Packrat parsers can support left recursion. *PEPM*, Vol. 8, pp. 103–110, 2008.

[30] Charles Donnelly and Richard Stallman. Bison. *Free Software Foundation*, Vol. 51, pp. 02110–1301, 1995.

[31] HashiCorp. Terraform. `https://terraform.io/`.

謝辞

本研究を進めるにあたり，研究方針や論文構成，発表手法など助言と指導をしていただいた指導教員の千葉滋教授に心より感謝致します．また，日々本研究の議論を行う際，様々な助言を頂いた市川和央氏に感謝致します．最後に，日頃から研究室生活やミーティングを通じてご支援くださった千葉研究室の皆様に感謝致します．

付録 A

実装した Ruby サブセット構文の例

ソースコード A.1. サブセット構文の例一覧

```
# グローバル変数, ローカル変数, インスタンス変数
$var = foobar + @v + 1 + 2.0

# 定数, 名前空間のラップ
v = ESAMPLE::METHODS

# 数値リテラル(整数、符号付き整数、浮動小数点数)

v = 123 + -12 * 123.0

# 文字列リテラル
a = "string" + 'string' + "\n"

# TrueClass, FalseClass
true == false

# 配列式
ary = [[1, 2], "apple"]

# Hash式
hash = {
  "key1" => 1,
  :key2 => 2,
  "key3": "3"
}

# 代入
foo = bar
foo[0] = bar
foo.bar = baz

# 自己代入
a += 1
b ||= 1

# 演算子
```

```
1 + 2 - 1 * 10 / 20
a && b || c

# if
if age >= 12
  puts "0"
elsif age <= 0
  puts "1"
else
  puts "2"
end

# if 修飾子
return 10 if 10 == a

# unless
unless true
  puts "0"
end

# unless 修飾子
return 10 unless 10 == a

# while
while i >= 0
  print i
  i -= 1
end

# untill
while i >= 0
  print i
  i -= 1
end

# メソッド呼び出し
v = call()
v = call
v = call(a)
v = call a
v = call(a, b)
v = call a, b

# ブロック付きメソッド呼び出し
call(10) { |x| x + 1 }

a.call(10) do |x|
  x + 1
end

# クラス定義
class Foo < Super
end
```

```
90
91  # モジュール定義
92  module Foo
93  end
94
95  # メソッド定義
96  def fact                              # 引数なし
97  end
98
99  def fact(n, k)
100 end
101
102 # コメント
103 # this is a comment line
```