平成 26 年度

筑波大学情報学群情報科学類

卒業研究論文

題目 レキシカル環境にメソッドを定義するオブジェクト指向言語 Suzu

主専攻　　ソフトウェアサイエンス主専攻

著者　　林 拓人

指導教員　前田敦司

要　旨

従来のオブジェクト指向言語は変数をローカル定義できるのに対しメソッドをローカル定義することはできない．ローカル変数の有用性は広く認められており，ローカルメソッドを定義できればこれも有用であると考えられる．

本研究はローカルメソッドを定義可能なオブジェクト指向言語 Suzu を開発しその有用性を実証する．Suzu ではローカルメソッドを用いてメソッドを局所的に追加・再定義することで，グローバル環境を汚染しない可読性の高い内部 DSL を構築できる．またローカルメソッドの実現にあたり変数定義とメソッド定義のシンタックスおよびセマンティクスを統一したことで，変数を扱うのと同じモジュールシステムによりメソッドを柔軟にグループ化・共通化・再利用できる．

従来の言語の類似する機構と比較した利点は，メソッドのスコープをブロックという細かい単位で制御できることと，モジュールによる衝突回避が利用できることである．

目 次

第1章　序論

オブジェクト指向言語とは，オブジェクト指向プログラミングの支援機構を持つプログラミング言語の総称である．多くのオブジェクト指向言語が持つ概念としてクラスとメソッドが挙げられる．クラスはオブジェクトの属性，メソッドはオブジェクトに対する操作である．あるクラスを属性として持つオブジェクトをそのクラスのインスタンスと呼ぶ．オブジェクトに対しメソッド名を指定してメソッド呼び出しを行うと，オブジェクトのクラスに応じて適切なメソッドが選択され呼び出される．

オブジェクト指向言語に限らずほとんどのプログラミング言語ではローカル変数を定義できる．ローカル変数とは定義したブロックや関数内でのみ参照可能な変数のことである．変数を参照可能な範囲のことをその変数のスコープと呼ぶ．ローカル変数はスコープがブロックや関数単位で細かく区切られており，そのブロックや関数内でのみ使いたい変数を新たに定義したり，外側のブロックや関数で定義された変数を局所的に再定義したりすることができる．ほとんどのプログラミング言語においてローカル変数を定義できることは，これらローカル変数の有用性が広く認められている証拠である．

従来のオブジェクト指向言語は，メソッドを変数のようにローカル定義することはできない．クラス定義の内側でメソッド定義を行う言語においてメソッドのスコープはあくまでクラス単位でのみ制御でき，ブロックや関数単位で局所的にメソッドを定義することは不可能である．しかしながらローカル変数が有用であるように，定義したブロックや関数内でのみ参照可能なローカルメソッドを定義できれば，メソッドを局所的に追加・再定義することができ有用であると考えられる．

そこで，本研究はローカルメソッドを定義可能なオブジェクト指向言語 Suzu を開発し，その有用性を実証する．ローカルメソッドを実現するにあたっては，変数定義とメソッド定義のシンタックスおよびセマンティクスを統一するというアプローチをとる．これらが統一されれば必然的にローカル変数に対応するローカルメソッドを定義できる．変数とメソッドの違いは識別に必要な識別子の数である．変数は変数名 1 つによって識別されるのに対し，メソッドはクラス名とメソッド名の 2 つによって識別される．ここでクラス名とメソッド名の 2 つを組にして 1 つの識別子としてとらえれば，変数名を用いる代わりにクラス名とメソッド名の組を用いることで変数と同じシンタックスおよびセマンティクスでメソッドを定義することができる．

ローカルメソッドの活用法として内部 DSL の構築が挙げられる．DSL とは Domain Specific Language の略で，ある領域の問題を解決することに特化した言語のことである．特に内部 DSL とは，他の言語のプログラム内に記述できる DSL のことを指す．Suzu は演算子の適用

をメソッド呼び出しとして解釈するため，対応するメソッドを定義することでユーザーが自由に演算子を追加・再定義できる．演算子をローカルメソッドとして定義すれば，演算子を用いた可読性の高い内部 DSL をメソッド衝突のリスクを避けつつ利用することができる．

また Suzu では変数定義とメソッド定義のシンタックスおよびセマンティクスを統一したことで，変数とメソッドを同じモジュールシステムによって統一的に扱うことができる．Suzu にはクラスを継承する機能が存在しないが，このモジュールシステムを用いることで多重継承に相当するメソッドの共通化・再利用が可能である．

従来の言語において Suzu と似た柔軟性を持つ機構は存在する．それらと比較した時の Suzu の利点は，メソッドのスコープをブロックという細かい単位で制御できることと，モジュールシステムによる衝突回避が利用できることである．

本稿は次のような構成をとる．第 2 章ではローカルメソッドを実現するために変数定義とメソッド定義のシンタックスおよびセマンティクスを統一する手法を提案する．第 3 章ではこの手法を適用し設計した言語の実例としてローカルメソッドを定義可能なプログラミング言語 Suzu の言語仕様を解説する．第 4 章ではローカルメソッドなどの特徴を生かした Suzu のプログラム例によってその有用性を検証する．第 5 章では Suzu と似た柔軟性を持つ従来の機構との関連性について述べる．第 6 章で Suzu の仕様と実装にまつわる課題を述べ，第 7 章で結論を述べる．

第2章　提案手法

従来のオブジェクト指向言語は変数をローカル定義することはできてもメソッドをローカル定義することはできない．本研究はメソッドのローカル定義を可能にしその有用性を検証するため，変数定義とメソッド定義のシンタックスおよびセマンティクスを統一する手法と，これを適用したことでメソッドをローカル定義できる新しいオブジェクト指向言語を提案する．

2.1　従来のオブジェクト指向言語

多くのオブジェクト指向言語は定義を記述したブロックや関数・メソッド内でのみ参照可能なローカル変数を定義することができる．例えばC++やJavaでは

```
{
  int v = ...
  ...
}
```

のようにすれば，ブロック{〜}内でのみ参照可能な変数vを定義できる．Rubyでは

```
1.times do
  v = ...
  ...
end
```

のようにすれば，ブロックdo〜end内でのみ参照可能な変数vを定義でき，

```
def m
  v = ...
  ...
end
```

のようにすれば，メソッドm内でのみ参照可能な変数vを定義できる．Pythonでは

```
def f():
  v = ...
  ...
```

のようにすれば，関数 f 内でのみ参照可能な変数 v を定義できる．

ローカル変数はブロックや関数・メソッド単位で局所的に変数を追加・再定義することができ有用である．オブジェクト指向言語に限らずほとんどのプログラミング言語でローカル変数を定義できることはこの有用性が広く認められている証拠だといえる．これに対しメソッドのローカル定義をサポートするオブジェクト指向言語は著者の知る限り存在せず，その有用性が検証されているとは言えない．しかしながらローカル変数が有用であるように，ローカル変数に対応するローカルメソッドを定義できれば，ブロックや関数・メソッド単位で局所的にメソッドを追加・再定義でき有用だと考えられる．

変数とメソッドの根本的な違いは識別に必要な識別子の数である．メソッドを定義する際にはメソッドの内容の他にクラス名とメソッド名の 2 つを指定する必要がある．例えばクラス C のインスタンスを操作するメソッド m を定義するには，C++では

```
class C {
public:
  void m(){
    ...
  }
};
```

Java では

```
class C {
  void m(){
    ...
  }
}
```

Ruby では

```
class C
  def m
    ...
  end
end
```

Python では

```
class C:
  def m(self):
    ...
```

のようにする．どの言語においてもメソッドを定義するにはクラス名 C とメソッド名 m の 2 つが必要である．変数を識別する識別子が変数名 1 つであるのに対し，メソッドを識別する識別子はクラス名とメソッド名の 2 つであると言える．

2.2 提案するオブジェクト指向言語

従来のオブジェクト指向言語はメソッドをローカル定義できないが，ローカル変数の有用性からローカルメソッドも有用であると考えられる．そこで本研究はローカルメソッドが定義可能なオブジェクト指向言語 Suzu を開発し，その有用性を検証する．

変数とメソッドの違いは識別に必要な識別子の数であることはすでに述べた．変数は変数名 1 つによって識別されるのに対し，メソッドはクラス名とメソッド名の 2 つによって識別される．ここでクラス名とメソッド名の 2 つを組にして 1 つの識別子としてとらえれば，変数名を用いる代わりにクラス名とメソッド名の組を用いることで変数と同じシンタックスおよびセマンティクスでメソッドを定義できる．例えば，

```
let v = ...
```

のようにして変数を定義し，

```
let C#m = ...
```

のようにしてメソッドを定義する．`C#m` はクラス名 `C` とメソッド名 `m` の組である．これにより，`begin`〜`end` をブロックとすると，

```
begin:
  let v = ...
  ...
end
```

のようにしてブロックローカルな変数を定義でき，

```
begin:
  let C#m = ...
  ...
end
```

のようにしてブロックローカルなメソッドを定義できる．

ローカル変数と同じようにローカルメソッドを定義できれば，ブロックや関数・メソッド単位で局所的にメソッドを追加・再定義でき有用だと考えられる．第 3 章ではこのようにクラス名とメソッド名の組を 1 つの識別子としてとらえてメソッドを定義することで変数定義とメソッド定義のシンタックスおよびセマンティクスを統一し，ローカル変数に対応するローカルメソッドを定義可能にしたオブジェクト指向言語 Suzu の言語仕様を解説する．第 4 章ではローカルメソッドや変数とメソッドの統一的なシンタックスおよびセマンティクスを生かした Suzu のプログラム例によってその有用性を検証する．

第3章　プログラミング言語Suzu

ローカル変数に対応するローカルメソッドが定義可能なプログラミング言語 Suzu の言語仕様をコード例を交えて解説する．Suzu はクラス名とメソッド名の組を用いて変数と同じようにメソッドを定義することで，変数定義とメソッド定義のシンタックスおよびセマンティクスを統一している．コード例において式を省略する際には... を用いる．

3.1　基本的な要素

変数は `let` を用いて定義する．例えば変数 `v` を... の値で初期化して定義するには以下のようにする．

```
let v = ...
```

Suzu では関数リテラルを記述できる．引数 `param1`，`param2` を受け取って... を実行する関数は以下のように書ける．

```
^(param1, param2){ ... }
```

関数は `def` を用いて定義できる．

```
def func(param1, param2):
  ...
end
```

これは以下のコードと等価である．

```
let func = ^(param1, param2){ ... }
```

関数 `func` を引数 `arg1`，`arg2` で呼び出すには以下のようにする．

```
func(arg1, arg2)
```

最後の引数として関数リテラルを渡す際は専用の構文が使える．例えば

```
func(arg1, arg2, ^(param1, param2){ ... })
```

という関数呼び出しは

```
func(arg1, arg2)^(param1, param2):
  ...
end
```

と書ける．他の実引数や関数リテラルの仮引数，またはその両方がない場合，

```
func^(param1, param2):
  ...
end

func(arg1, arg2):
  ...
end

func:
  ...
end
```

のように省略して書ける．

Suzu では:からインデントの終わりまで（または{から}まで）をブロックと呼ぶ．ブロック内のコードを実行する際は新たなレキシカル環境 [1] が生成され有効になり，実行が終わると破棄される．変数は有効なレキシカル環境に定義される．よってブロック内で定義された変数はブロック内でのみ参照可能なローカル変数となる．例えば

```
def begin(thunk):
  thunk()
end
```

のような無引数の関数 thunk を受け取って呼び出す高階関数 begin に

```
begin:
  let v = ...
  ...
end
```

のように無引数の関数リテラルを渡して呼び出させた場合，変数 v は begin から end の間のみで有効なローカル変数となる．

3.2 オブジェクトシステム

Suzu では変数名の代わりにクラス名とメソッド名の組を指定し，メソッドを変数と同様レキシカル環境に定義する．つまりメソッドを定義するには，

```
def C#m(self, param1, param2):
  ...
end
```

または

```
let C#m = ^(self, param1, param2){ ... }
```

のようにする．C はクラス名，m はメソッド名，C#m はクラス名とメソッド名の組である．

定義したメソッドは C#m と書くことで変数のように参照できる他，inst をクラス C のインスタンスとすると以下のようにして呼び出せる．

```
inst.m(arg1, arg2)
```

これは以下の呼び出しと同じ結果となる．

```
C#m(inst, arg1, arg2)
```

このように，メソッドの第 1 引数にはメソッド呼び出しの対象となったオブジェクト自身が，第 2 引数以降には実引数が渡される．つまり self には inst，param1 には arg1，param2 には arg2 の内容がそれぞれ代入される．

クラスは以下のようにして定義する．

```
class Point = make_point:
  x
  y
end
```

これにより，クラス Point，コンストラクタ関数 make_point，ゲッターメソッド Point#x，Point#y が定義される．make_point は x と y というフィールドを持った Point のインスタンスを生成する関数である．フィールドへのアクセスは Point#x，Point#y を介してのみ行える．例えば

```
let p = make_point(1, 2)
```

とすると，p.x は 1，p.y は 2 となる．

クラス定義の際，

```
class Point = make_point:
  mutable x
  mutable y
end
```

のようにフィールド名の前に mutable と書くと，上記に加えてセッターメソッド Point#(x=)，Point#(y=) が定義される．

```
p.x = 3
p.y = 4
```

のような式は

```
p.(x=)(3)
p.(y=)(4)
```

というメソッド呼び出しに解釈されるため，これを用いて `Point#(x=)` や `Point#(y=)` を呼び出すことで `p` のフィールド `x`，`y` を書き換えられる．

Suzu では演算子の適用もメソッド呼び出しとして解釈される．正規表現 `[-+*/%&|=<>]+` にマッチする文字列は演算子として扱われ，優先順位や結合則は演算子の 1 文字目に依存する．例えば

```
x <- y
```

という式は

```
x.(<-)(y)
```

と解釈される．優先順位や結合則は`<`と同じである．

ここまで述べてきたことから分かるように，Suzu ではクラス定義とメソッド定義を分けて記述する．つまりはクラスを定義した後に，そのクラスのインスタンスを操作するメソッドを自由に追加できる．また，メソッドは変数と同様レキシカル環境に定義されるため，特定のブロック内でのみ有効なローカルメソッドを定義できる．さらに演算子の適用もメソッド呼び出しとして解釈されるため，ローカルメソッドを定義することでクラスごとの演算子の意味を局所的に変えることもできる．

3.3 モジュールシステム

モジュールとは，変数や関数などの複数の定義をまとめ，指定した名前の変数や関数のみを限定的に公開（エクスポート）する機能である．エクスポートされていない名前はモジュールの外から見えないため，実装の詳細を隠蔽したり，名前の衝突を避けたりするのに有用である．

Suzu はメソッドを変数と同様レキシカル環境に定義するため，モジュールによって変数と同じようにメソッドの可視性を制御できる．モジュールを定義するには以下のようにする．

```
module M:
  let f = ...
  let C#m = ...
  ...
  export f, C#m
end
```

これにより，変数 f とメソッド C#m をエクスポートするモジュール M が定義される．let（または def）を用いて変数やメソッドを定義した後，export の後にエクスポートしたい変数の変数名や，メソッドを表すクラス名とメソッド名の組を並べて指定する．エクスポートされている変数やメソッドは::を用いて，M::f，M::(C#m) のように参照できる．モジュール内で定義した変数やメソッドをエクスポートしなかった場合，それらはモジュール内でのみ有効なモジュールローカルな変数，またはメソッドとなる．

モジュールは任意のスコープで open できる．例えば

```
begin:
  open M
  ...
end
```

とすると，begin から end までのブロックで M からエクスポートされている変数やメソッドを修飾子なしで参照できるようになる．M を先ほど定義したものとすると，f，C#m のように変数やメソッドを参照可能となる．

既存のモジュールを活用して新しいモジュールを定義する際は include が便利である．

```
module B:
  include A
  ...
end
```

この例ではモジュール B 内でモジュール A を include している．include A とするとまず open A が行われ，次に A からエクスポートされている変数やメソッドをすべて B からもエクスポートする．これによりモジュール A を拡張した新たなモジュール B を容易に定義できる．

open や include によって新しく参照可能になる変数やメソッドが衝突する場合，そのコードはエラーとなる．例えば以下のコードは open Y で C#q が衝突するためエラーとなる．

```
module X:
  ...
  export C#p, C#q
end
module Y:
  ...
  export C#q, C#r
end
begin:
  open X
  open Y
```

```
  ...
end
```

これを回避するには except を用いる．

```
begin:
  open X except C#q
  open Y
  ...
end
```

この例では X を open する際 C#q を except している．これによりこの時点で C#q はインポートされない．よって open Y としても衝突は起こらずエラーは回避される．

Suzu にはパラメータ化されたモジュールとしてのトレイトという機構がある．トレイトは値を受け取ってモジュールを返す関数である．

```
trait T(param1, param2):
  let f = ...
  let C#m = ...
  ...
  export f, C#m
end
```

この例では param1 と param2 を受け取って，f と C#m をエクスポートするモジュールを返すトレイト T を定義している．トレイトの活用法については第 4 章で述べる．

第4章　考察

Suzu のローカルメソッドやモジュールシステムなどを生かしたプログラム例によって，変数定義とメソッド定義のシンタックスおよびセマンティクスを統一することによりもたらされる有用性を検証する．なお，p は引数として受け取った値を出力する関数，//から行末まではコメントである．本章ではコメントとして//=>の後にプログラムの出力内容を記述する．

4.1　組み込みオブジェクトに対するメソッドの追加

Suzu ではメソッド定義がクラス定義から独立しているため，組み込みオブジェクトに対するメソッドもユーザーが自由に追加できる．例えば，

```
p("program".pluralize)     //=> "programs"
p("programs".singularize) //=> "program"
p("person".pluralize)      //=> "people"
p("people".singularize)    //=> "person"
```

のように，文字列を複数形および単数形に変換するメソッド pluralize，singularize を追加したければ，

```
def String::C#pluralize(self):
  ...
end
def String::C#singularize(self):
  ...
end
```

のようにして定義できる．String::C というのはモジュール String の変数 C で，組み込みオブジェクトである文字列のクラス名を指している．

定義をブロック内で行えばこれらはローカルメソッドとなるため，メソッドの衝突を未然に防ぐことができる．

```
begin:
  def String::C#pluralize(self):
    ...
  end
```

```
  def String::C#singularize(self):
    ...
  end
  ...
end
```

上の例では 2 つのメソッド定義は begin から end の間でのみ有効である．
また，複数のメソッドをまとめてモジュールとして提供することもできる．

```
module Noun:
  def String::C#pluralize(self):
    ...
  end
  def String::C#singularize(self):
    ...
  end
  ...
  export String::C#pluralize, String::C#singularize
end
```

これをブロック内で open すれば直接定義したのと同じ効果が得られる．

```
begin:
  open Noun
  ...
end
```

このように，利便性の高いメソッド群をユーザーがモジュールとして提供し，局所的に有効にすることでメソッド衝突の危険を避けながら利用できる．

4.2 演算子の局所的な再定義

Suzu では演算子の適用がメソッド呼び出しとして扱われる．そのため演算子に対応するメソッドを局所的に再定義することで，その振る舞いを変えることができる．
例えば整数同士の割り算を行うメソッド Int::C#(/) は，組み込みのものは整数を返す．これを浮動小数点数を返す関数 Int::quo で置き換えたい場合，単に let を用いればよい．

```
p(3 / 2) //=> 1
begin:
  let Int::C#(/) = Int::quo
  p(3 / 2) //=> 1.5
```

```
end
p(3 / 2) //=> 1
```

また以下のようなモジュール `Quotient` を定義しておけば，ブロック内で `open` することでそのスコープでのみ浮動小数点数を返すよう変えることができる．

```
module Quotient:
  let Int::C#(/) = Int::quo
  export Int::C#(/)
end

p(3 / 2) //=> 1
begin:
  open Quotient
  p(3 / 2) //=> 1.5
end
p(3 / 2) //=> 1
```

Suzu のメソッドは変数と同じレキシカルスコープなので，ブロック内から呼び出した先の関数に影響を与えることなく安全に演算子の再定義ができる．

4.3 内部 DSL

DSL とは Domain Specific Language の略で，特定の問題を解決するためのミニ言語のことである．特に内部 DSL とは，プログラム内に記述できる式として構築した DSL を指す．

Suzu は演算子として使用できる文字列に自由度があるため，演算子をローカルメソッドとして定義すれば，グローバル環境を汚染せず可読性の高い内部 DSL を作成できる．以下の例は演算子式によって正規表現を構築できる DSL を提供するモジュール `PrettyRegex` を定義している．`String::format` は文字列整形関数である．

```
module PrettyRegex:
  def String::C#(|)(lhs, rhs):
    String::format("({0}|{1})", lhs, rhs)
  end
  def String::C#(+)(lhs, rhs):
    String::format("{0}{1}", lhs, rhs)
  end
  def Char::C#(-)(lhs, rhs):
    String::format("[{0}-{1}]", lhs, rhs)
  end
```

```
  def String::C#one_or_more(exp):
    String::format("{0}+", exp)
  end
  export String::C#(|)
  export String::C#(+)
  export String::C#one_or_more
  export Char::C#(-)
end
```

これは以下のように使用できる．

```
let regex = begin:
  open PrettyRegex
  ("foo"|"bar")+('0'-'9').one_or_more
end
p(regex) //=> "(foo|bar)[0-9]+"
```

縦棒やプラス，ハイフンといった衝突の危険が高いと思われる短い演算子も，スコープを限定して定義することで安心して使用できる．付録 A ではより規模の大きな DSL の例として PEG パーザコンビネータライブラリを紹介している．

4.4 メソッドの柔軟なグループ化

Suzu のメソッド定義はクラス定義とは独立しており，モジュールに所属する形で定義することができる．そのため対象のクラスによらず関連性の高いメソッドを集めて柔軟にグループ化できる．

例えばオブジェクトを JSON 形式でシリアライズするメソッド `to_json` を定義したい場合，クラスにメソッドを定義する言語では定義が各クラスに分散してしまう．

Suzu では以下のように複数のクラスのインスタンスに対するメソッドを提供するモジュールを定義することで，関連するメソッドを 1 箇所にまとめて定義できる．

```
module ToJSON:
  let    Int::C#to_json = ...
  let String::C#to_json = ...
  let   List::C#to_json = ...
  ...
end
```

利用する側は有効にしたいスコープで `open ToJSON` とすればよい．

また，ユーザーが新たに定義したデータ型などさらに多くのオブジェクトにメソッドを提供したい場合，

```
module ToJSONExt:
  include ToJSON
  let UserDefinedData1#to_json = ...
  let UserDefinedData2#to_json = ..
  ...
end
```

のように include を用いて，既存のモジュールを拡張した新たなモジュールを定義できる．

4.5 トレイトによるメソッドの共通化

複数のクラスに同じ内容のメソッドを定義したい場合，トレイトによってメソッドの定義を共通化できる．

例えば銀行口座を表すクラス BankAccount と株式口座を表すクラス StockAccount にメソッドを定義することを考えてみる．BankAccount は残高を表すフィールド balance を持つ．

```
class BankAccount = make_bank_account:
  mutable balance
end
```

預け入れを行うメソッド deposit!と引き出しを行うメソッド withdraw!は以下のように書ける．

```
def BankAccount#deposit!(self, x):
  self.balance = self.balance + x
end

def BankAccount#withdraw!(self, x):
  self.balance = self.balance - x
  if(self.balance < 0):
    self.balance = 0
  end
end
```

フィールドの参照およびフィールドへの代入には，クラス定義によって自動的に定義されたゲッターメソッド balance とセッターメソッド balance=を使用している．

これに対し StockAccount は株数を表すフィールド num_shares と株単価を表すフィールド price_per_share を持つ．

```
class StockAccount = make_stock_account:
```

```
  mutable num_shares
  price_per_share
end
```

残高を取得するメソッド balance と残高を変更するメソッド balance= は以下のように定義できる.

```
def StockAccount#balance(self):
  self.num_shares * self.price_per_share
end

def StockAccount#(balance=)(self, x):
  self.num_shares = x / self.price_per_share
end
```

預け入れを行うメソッド deposit!と引き出しを行うメソッド withdraw!は以下のように書ける.

```
def StockAccount#deposit!(self, x):
  self.balance = self.balance + x
end

def StockAccount#withdraw!(self, x):
  self.balance = self.balance - x
  if(self.balance < 0):
    self.balance = 0
  end
end
```

ここで, メソッド deposit!および withdraw!の内容は, BankAccount と StockAccount とで同一であることが分かる. 違いは定義対象のクラスと, 使用している balance および balance=の定義である.

Suzu ではこのようなメソッドの定義をトレイトによって共通化できる.

```
trait Account(C, C#balance, C#(balance=)):
  def C#deposit!(self, x):
    self.balance = self.balance + x
  end

  def C#withdraw!(self, x):
    self.balance = self.balance - x
    if(self.balance < 0):
```

```
        self.balance = 0
      end
    end

    export C#deposit!, C#withdraw!
  end
```

トレイト Account は，クラス C，メソッド C#balance，C#(balance=) を引数として受け取り，メソッド C#deposit!，C#withdraw!が定義されたモジュールを返す．Account に BankAccount と StockAccount それぞれのクラスと balance および balance=を渡して呼び出し戻り値を open すれば，それらを使用した deposit!および withdraw!が定義される．つまり，

```
open Account(BankAccount,
             BankAccount#balance,
             BankAccount#(balance=))
```

とすれば BankAccount#deposit!と BankAccount#withdraw!が定義され，

```
open Account(StockAccount,
             StockAccount#balance,
             StockAccount#(balance=))
```

とすれば StockAccount#deposit!と StockAccount#withdraw!が定義される

このように，Suzu ではクラスとメソッドによってパラメータ化したモジュールとしてトレイトを定義することでメソッドの定義を共通化することができ，その呼び出し結果を open することで同じ内容のメソッドを繰り返し書く必要がなくなる．Suzu には継承機構が存在しないが，メソッドがモジュール等のレキシカル環境に所属するため実装の継承をモジュールシステムの機能によって代替できることを示している．

4.6 多重継承の代替

Suzu には継承という名の機能は存在しない．しかしながらトレイトを用いれば，多重継承に相当するようなメソッドの再利用ができる．

コレクションに対しシーケンシャルアクセスを行うための 3 つのクラス，ReadStream，WriteStream，ReadWriteStream を実装したいとする．ReadStream は read と seek，WriteStream は write と seek，ReadWriteStream は read と write と seek をそれぞれ可能にしたい．read はコレクションの現在位置から 1 つの要素を読み出し，現在位置を進める．write はコレクションの現在位置に 1 つのアイテムを書き込み，現在位置を進める．seek は現在位置を変更する．

多重継承を持つ言語ならばまずReadStreamとWriteStreamを定義し，その後これら2つを継承したReadWriteStreamを定義するだろう．Suzuでは同等のことをトレイトを用いて実現できる．

まず，クラスにreadとseekを提供するトレイトReadableと，クラスにwriteとseekを提供するトレイトWritableを定義する．

```
trait Readable(C, C#collection,
               C#position, C#(position=)):
  def C#read(self):
    let elem = self.collection[self.position]
    self.position = self.position + 1
    elem
  end

  def C#seek(self, position):
    self.position = position
  end

  export C#read, C#seek
end

trait Writable(C, C#collection,
               C#position, C#(position=)):
  def C#write(self, item):
    self.collection[self.position] = item
    self.position = self.position + 1
  end

  def C#seek(self, position):
    self.position = position
  end

  export C#write, C#seek
end
```

ReadableとWritableはともに，引数としてクラスC，メソッドcollection，position，position=を要求している．

その後，クラスReadStreamを定義してReadableを適用，クラスWriteStreamを定義してWritableを適用，クラスReadWriteStreamを定義してReadableとWritableを適用する．ReadWriteStreamにReadableとWritableを適用する際は，seekが重

複しないようどちらかの定義を except してやる必要がある.

```
class ReadStream = make_read_stream:
  mutable collection
  mutable position
end
open Readable(
  ReadStream, ReadStream#collection,
  ReadStream#position, ReadStream#(position=),
  )

class WriteStream = make_write_stream:
  mutable collection
  mutable position
end
open Writable(
  WriteStream, WriteStream#collection,
  WriteStream#position, WriteStream#(position=),
  )

class ReadWriteStream = make_read_write_stream:
  mutable collection
  mutable position
end
open Readable(
  ReadWriteStream, ReadWriteStream#collection,
  ReadWriteStream#position, ReadWriteStream#(position=),
  )
open Writable(
  ReadWriteStream, ReadWriteStream#collection,
  ReadWriteStream#position, ReadWriteStream#(position=),
  )
  except ReadWriteStream#seek
```

このように，Suzu ではクラスにメソッドを提供する機能としてトレイトを用いることで，モジュールの生成と open によるメソッドの再利用を行える．メソッドが重複する際にはモジュールに対する操作である except を使い，衝突を適切に回避できる．

4.7 本章のまとめ

Suzu ではメソッド定義がクラス定義から独立しているので，組み込みクラスに対しメソッドを自由に追加・再定義できる．メソッドはレキシカル環境に定義されるので，追加や再定義の影響範囲はグローバルにもローカルにもできる．ローカルな影響範囲はブロックという細かい単位で指定できる（4.1 節）．演算子の適用はメソッド呼び出しとして解釈されるので，演算子の追加や再定義も可能である（4.2 節）．演算子にあたるメソッドをローカル定義することで，グローバル環境を汚染しない可読性の高い内部 DSL を構築できる（4.3 節）．メソッド定義がクラス定義から独立していることは，対象のクラスに縛られないメソッドのグループ化も可能にしている（4.4 節）．またパラメータ化されたモジュールであるトレイトを使えば，メソッドを共通化し衝突を回避しながら再利用することができる（4.5 節，4.6 節）．

第5章　関連研究

Suzu のオブジェクトシステムおよびメソッドシステムがもたらすのと類似した柔軟性を持つ従来の機構との関連性について述べる．

GluonJ[2] はアスペクト指向プログラミングを支援する Java の拡張である．Glue と呼ばれるクラスを定義することで，既存のクラスに対しメソッドを追加・再定義するなどの拡張が可能である．しかし拡張はグローバルに反映され，その影響範囲を限定することはできない．

C#の拡張メソッド [3] は既存のクラスにメソッドを追加したように見せることができる機能である．実際に呼ばれるのは第 1 引数に `this` と指定した静的メソッドである．拡張メソッドは `using` ディレクティブによって有効無効を制御できるが，同じシグネチャを持つ拡張メソッドが複数存在する場合エラーが起き，この衝突を個別に解決する手段を持たない．また既に通常の方法でクラスに定義されているインスタンスメソッドを再定義することもできない．

Scala[4] の implicit conversion も既存のクラスに対しメソッドを追加したように見せられる機能である．実際は存在しないメソッドが呼ばれた際に型変換を行うことで，別のクラスのメソッドを呼び出している．スコープはブロック単位で制御できるが，同じシグネチャのメソッドを提供する型への変換が複数存在する場合エラーが起き，この衝突を個別に解決する手段を持たない．またこれも拡張メソッドと同様，既に通常の方法でクラスに定義されているメソッドを再定義することはできない．

Haskell の型クラス [5] はオブジェクト指向プログラミングのための機構ではないが，データ型の定義と独立して型ごとに操作を定義できる点が Suzu のメソッドに類似している．型クラスは導入に静的な型を必要とするが，Suzu のメソッドはこれを必要としない．また型クラスにおいて操作の実体を定義するインスタンス宣言はモジュールシステムによって個別にエクスポートするかどうかの指定ができない．ただし型クラスは戻り値の型に応じて関数の振る舞いを変えさせることができるのに対し，Suzu ではこれは不可能である．

OCaml[6] の local open は Suzu と同様モジュール内の変数を局所的にインポートできる．これとユーザー定義演算子を組み合わせると演算子の意味を局所的に変えられる．しかし OCaml にはオーバーロードが無いため演算子の振る舞いを型ごとに変えることができず，既存の演算子を上書きしてしまうため内部 DSL を構築するには不便である．

Traits[7, 8, 9] は Suzu のトレイトの元となった概念である．メソッドの集合であるトレイトについて合成やリネームなどの演算を行うことで，メソッドの衝突を回避しつつクラスにメソッドを提供できる．Suzu はクラスではなくレキシカル環境にメソッドを定義するため，トレイトを値を受け取ってモジュールを返す関数としてとらえ直した．この方式では必要なメソッドと提供するメソッドが関数の引数と戻り値として明確となり，モジュール演算によっ

てメソッドの衝突を回避できるということを示した．

CLOS[10] は Common Lisp の言語仕様に含まれるオブジェクト指向プログラミングの支援機構で，Suzu と同様メソッドをクラスとは独立して定義する．メソッドはクラスではなく総称関数に対して追加され，総称関数は引数として与えられたオブジェクトに応じて適切なメソッドを選択し呼び出す．CLOS は CLtL2[11] と呼ばれる仕様では既存の総称関数の内容をコピーしてローカルな総称関数を定義する `with-added-methods` が含まれていたが，最新の仕様からは削除されている．これはメソッドのローカル定義が有用でないと判断されたためである．Suzu は非 S 式文法の言語として，ユーザー定義演算子と組み合わせることでグローバル環境を汚染しない可読性の高い内部 DSL を構築し，ローカルメソッドの有効な利用法を示した．

Classbox[12]，Refinements[13]，Method Shells[14] は，いずれも既存のオブジェクト指向言語に対しモジュール単位でスコープを限定したメソッドの追加・再定義を行うための機構である．違いはメソッドのスコープルールであり，Classbox はダイナミックスコープ，Refinements はレキシカルスコープ，Method Shells はそれらを切り替えられる．Suzu のモジュールシステムは Refinements に近い．ただし Suzu はモジュール単位のみならずブロック単位でメソッドの追加・再定義が行え，内部 DSL の構築の際により利便性が高い．

MixJuice[15] はクラス定義を複数のモジュールに分割して書くことができる言語である．MixJuice によるモジュール分割は Suzu が行えるそれと非常に類似している．MixJuice にはパラメータ化されたモジュールがないが Suzu にはあり，Suzu には静的な型検査がないが MixJuice にはある．また Suzu のメソッドのスコープはモジュール単位ではなくブロック単位でありより細かい単位で制御できる．

第6章　今後の課題

Suzu の仕様と実装にまつわる今後の課題を述べる．

6.1　継承機構

Suzu にはクラスを継承する機能を実装していない．これはローカルメソッドと組み合わせた際にどのように振舞うのが適切か定かではないためである．クラス A を継承したクラス B があるとする．ネストしたブロックにおいて，

```
begin:
  let B#m = ..
  begin:
    let A#m = ...
    inst.m
  end
end
```

のように，外側のブロックでサブクラス B のメソッド m を，内側のブロックでスーパークラス A のメソッド m を定義し，サブクラス B のインスタンス inst に対しメソッド m を呼び出した場合，どちらの m が呼び出されるべきか．

つまり，内側に定義されたスーパークラスのメソッドと外側に定義されたサブクラスのメソッドのどちらを優先すべきかという問題である．この問題を解決するには，実際のプログラムにおいてこのようなケースが生じた場合どちらが優先されてほしいかを調査する必要がある．

6.2　多重ディスパッチ

1 つのオブジェクトのクラスに応じてメソッドの選択を行う単一ディスパッチに対し，複数のオブジェクトのクラスに応じてメソッドの選択を行うのが多重ディスパッチである．Suzu は単一ディスパッチにしか対応していないが，メソッドはクラスではなくレキシカル環境に定義されているためディスパッチで考慮するクラスを 1 つに限定する必要はなく，多重ディスパッチが可能な CLOS に倣えば原理上比較的容易に多重ディスパッチに対応できると考えられる．

問題は適切なシンタックスを考案することである．内部 DSL の構築にあたっては語順も重要な要素なので，なるべく語順を崩すことのない文法が必要である．

6.3 ダイナミックスコープ

Suzu のメソッドは変数と同じレキシカルスコープである．これにより演算子を用いた内部 DSL を構築するにあたって，スコープ外の関数を呼び出した際演算子の振る舞いが変わっているために結果がおかしくなってしまうようなことが起きないようになっている．

しかしながらダイナミックスコープであるほうが有用な場合もある．例えば値を出力する関数 `print` が内部で値に対しメソッド `to_string` を呼び出しているとすると，ダイナミック環境の `to_string` を再定義することで局所的に出力内容を変えることができる．

レキシカルスコープ・ダイナミックスコープはともに有用であるため，ユースケースを分析し適切な方法で共存させつつ導入する必要がある．

6.4 オブジェクト固有のメソッド

Ruby の特異メソッド，Common Lisp の EQL スペシャライザ，あるいは ECMAScript [16] のようなプロトタイプベースのオブジェクト指向言語を用いると，オブジェクトに対しクラスによらない固有のメソッドを定義できる．Suzu は現状そのような機能は持っておらず，オブジェクト固有のメソッドは定義できない．しかしながらまずはこれら既存の機能の有用性自体を検証する必要がある．

6.5 効率的な実装

Suzu ではブロックという細かい単位でメソッドの再定義ができる．これはつまり同じクラスのオブジェクトに対し同じ名前のメソッドを呼び出しても，プログラムのどの位置で呼び出したかによって呼び出されるメソッドが異なるということである，これは従来のオブジェクト指向言語にはない性質であり，メソッド呼び出しに対する既存の最適化手法は適用できない可能性がある．現在 Suzu の処理系はメソッド呼び出しに関して特に最適化を施しておらず，適切な最適化手法を考え実装することが課題である．

第7章　結論

ローカル変数に対応するローカルメソッドを定義可能なオブジェクト指向言語 Suzu を開発し，その有用性を実証した．

Suzu は変数名の代わりにクラス名とメソッド名の組を用いることで変数と同じシンタックスおよびセマンティクスでメソッドを定義でき，これによりメソッドを変数と同じようにローカル定義することを可能にしている．ローカルメソッドをユーザー定義演算子と組み合わせることで，グローバル環境を汚染せず可読性の高い内部 DSL を利用できることを示した．また変数定義とメソッド定義のセマンティクスを統一することでモジュールシステムを統一でき，メソッドの柔軟なグループ化・共通化・再利用が可能となることを示した．

従来の研究と比べてメソッドのスコープをブロックという細かい単位で制御でき，かつモジュールシステムによる衝突回避を利用できることが Suzu の優れている点である．今後は他のオブジェクト指向言語が持つ機能への対応や効率的な実装が課題である．

謝辞

本研究を行うにあたり，多大なるご指導とご助言を下さった筑波大学システム情報系前田敦司准教授に深く感謝いたします．また第 56 回プログラミング・シンポジウムにて有益なコメントを下さった方々に感謝いたします．最後に貴重なご意見を下さった筑波大学インタラクティブ・アーキテクチャ研究室の皆様と OB の水島宏太さんに感謝いたします．

参考文献

[1] Franklyn Turbak and David Gifford. *Design Concepts in Programming Languages*. The MIT Press, 2008.

[2] Shigeru Chiba, Atsushi Igarashi, and Salikh Zakirov. Mostly Modular Compilation of Crosscutting Concerns by Contextual Predicate Dispatch. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pp. 539–554, New York, NY, USA, 2010. ACM.

[3] Microsoft. 拡張メソッド（C#プログラミングガイド）. `http://msdn.microsoft.com/ja-jp/library/bb383977.aspx`, 2015.

[4] Martin Odersky, Lex Spoon, BillVenners（著）, 長尾高弘（翻訳）, 羽生田栄一（監修）, 水島宏太（特別寄稿）. Scala スケーラブルプログラミング第 2 版. インプレスジャパン, 2011.

[5] P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pp. 60–76, New York, NY, USA, 1989. ACM.

[6] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The OCaml system release 4.02 Documentation and user ' s manual. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`, 2014.

[7] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable Units of Behaviour. In Luca Cardelli, editor, *ECOOP 2003 Object-Oriented Programming*, Vol. 2743 of *Lecture Notes in Computer Science*, pp. 248–274. Springer Berlin Heidelberg, 2003.

[8] Andrew P. Black, Nathanael Schärli, and Stéphane Ducasse. Applying Traits to the Smalltalk Collection Classes. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pp. 47–64, New York, NY, USA, 2003. ACM.

[9] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, Vol. 28, No. 2, pp. 331–388, March 2006.

[10] 井田昌之, 元吉文男, 大久保清貴（編）. bit 別冊 Common Lisp オブジェクトシステム— CLOS とその周辺—. 共立出版, 1989.

[11] Guy L.Steele Jr.（著）, 井田昌之（翻訳監修）. COMMON LISP 第 2 版. 共立出版, 1992.

[12] Alexandre Bergel, Stphane Ducasse, and Roel Wuyts. Classboxes: A Minimal Module Model Supporting Local Rebinding. In László Böszörményi and Peter Schojer, editors, *Modular Programming Languages*, Vol. 2789 of *Lecture Notes in Computer Science*, pp. 122–131. Springer Berlin Heidelberg, 2003.

[13] Shugo Maeda, Benoit Daloze, Hiroshi Shibata, and Eric Hodel. Refinements. `https://github.com/ruby/ruby/blob/trunk/doc/syntax/refinements.rdoc`, 2014.

[14] 竹下若菜, 千葉滋. 破壊的クラス拡張で生じるメソッド衝突を回避可能なモジュール機構 Method Shells とその実装方法. 情報処理学会論文誌. プログラミング, Vol. 7, No. 3, pp. 12–21, Jul 2014.

[15] Yuuji Ichisugi and Akira Tanaka. Difference-Based Modules: A Class-Independent Module Mechanism. In Boris Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming*, Vol. 2374 of *Lecture Notes in Computer Science*, pp. 62–88. Springer Berlin Heidelberg, 2002.

[16] Ecma International. ECMAScript Language Specification - ECMA-262 Edition 5.1. `http://www.ecma-international.org/ecma-262/5.1/`, 2011.

[17] ケン・アーノルド, ジェームズ・ゴスリン, デビッド・ホームズ（著）, 柴田芳樹（翻訳）. プログラミング言語 Java 第 4 版. ピアソン・エデュケーション, 2008.

[18] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. ContextJ: Context-oriented Programming with Java. *Information and Media Technologies*, Vol. 6, No. 2, pp. 399–419, 2011.

[19] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A Monotonic Superclass Linearization for Dylan. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pp. 69–82, New York, NY, USA, 1996. ACM.

[20] Tamiya Onodera and Hiroaki Nakamura. Optimizing smalltalk by selector code indexing can be practical. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 - Object-Oriented Programming*, Vol. 1241 of *Lecture Notes in Computer Science*, pp. 302–323. Springer Berlin Heidelberg, 1997.

[21] 小野寺民也. オブジェクト指向言語におけるメッセージ送信の高速化技法. 情報処理, Vol. 38, No. 4, pp. 301–310, Apr 1997.

[22] 笹田耕一. プログラム言語 Ruby におけるメソッドキャッシング手法の検討. 情報処理学会第 67 回全国大会, 4N-6, Vol.1, pp. 305–306, 2005.

[23] 浅井健一. shift/reset プログラミング入門. http://pllab.is.ocha.ac.jp/~asai/cw2011tutorial/main-j.pdf, 2011.

[24] 増子萌, 浅井健一. MinCaml コンパイラにおける shift/reset の実装. 第 11 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論文集, pp. 163–177, 2009.

[25] 増子萌, 浅井健一. 例外と限定継続命令をサポートする評価器からの仮想機械とコンパイラの導出. 第 15 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論文集, 2013.

[26] 林拓人, 前田敦司. 環境にメソッドを直接格納する新しいオブジェクトシステムの提案. 第 56 回プログラミング・シンポジウム予稿集, pp. 121–130, 2015.

[27] Bryan Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pp. 111–122, New York, NY, USA, 2004. ACM.

[28] Bryan Ford. Packrat Parsing: Simple, Powerful, Lazy, Linear Time. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pp. 36–47, New York, NY, USA, 2002. ACM.

付 録A　PEGパーザコンビネータライブラリ

Suzu を用いて作成した内部 DSL の例として PEG パーザコンビネータライブラリを紹介する．PEG（Parsing Expression Grammar）[27] とは形式言語を認識する構文規則を記述するための文法で，曖昧さがないためパーザを構成するのに向いている．パーザコンビネータとは高階関数として定義されたパーザ群のことである．このライブラリは PEG の項（parsing expression）についてそれぞれに対応するコンビネータを定義しており，Suzu のプログラム内に PEG を内部 DSL として記述することでパーザを構成できる．構文解析アルゴリズムには Packrat Parsing[28] を用いている．これはバックトラックを伴う再帰下降構文解析にメモ化を組み合わせたもので，任意の PEG を解析可能かつ解析時間を入力の長さに対して線形時間で抑えられる．

以下に PEG パーザコンビネータライブラリのソースコードの全文を示す．

```
1  module PEG:
2    class PE:
3      def MkPE(proc)
4    end
5
6    class Result:
7      def Success(pos)
8      def Failure()
9    end
10
11   def parse(MkPE(proc), str):
12     proc(str, 0, Hash::create(16))
13   end
14
15   trait New():
16     let nonterms = Hash::create(16)
17
18     def def_nonterm(name, MkPE(proc)):
19       nonterms[name] = proc
20     end
21
22     def nonterm(name):
23       MkPE^(str, pos, caches):
24         match(Hash::get(caches, (name, pos))):
25         case(Some(result)):
26           result
27         case(None()):
28           let result = nonterms[name](str, pos, caches)
29           caches[(name, pos)] = result
30           result
```

```
31          end
32        end
33      end
34
35      def char(c):
36        MkPE^(str, pos, caches):
37          if(pos < String::length(str) && str[pos] == c):
38            Success(pos + 1)
39          else:
40            Failure()
41          end
42        end
43      end
44
45      def string(s):
46        let len = String::length(s)
47        MkPE^(str, pos, caches):
48          if(pos + len <= String::length(str) && String::sub(str, pos, len) == s):
49            Success(pos + len)
50          else:
51            Failure()
52          end
53        end
54      end
55
56      def char_of(cs):
57        MkPE^(str, pos, caches):
58          if(pos < String::length(str) && String::contain?(cs, str[pos])):
59            Success(pos + 1)
60          else:
61            Failure()
62          end
63        end
64      end
65
66      let any_char = MkPE^(str, pos, caches):
67        if(pos < String::length(str)):
68          Success(pos + 1)
69        else:
70          Failure()
71        end
72      end
73
74      let empty = MkPE^(str, pos, caches):
75        Success(pos)
76      end
77
78      def seq(MkPE(proc1), MkPE(proc2)):
79        MkPE^(str, pos, caches):
80          match(proc1(str, pos, caches)):
81          case(Success(pos)):
82            proc2(str, pos, caches)
83          case(Failure()):
84            Failure()
```

```
85          end
86        end
87      end
88
89      def choice(MkPE(proc1), MkPE(proc2)):
90        MkPE^(str, pos, caches):
91          match(proc1(str, pos, caches)):
92          case(Success(pos)):
93            Success(pos)
94          case(Failure()):
95            proc2(str, pos, caches)
96          end
97        end
98      end
99
100     def zero_or_more(MkPE(proc)):
101       MkPE^(str, pos, caches):
102         def loop(pos):
103           match(proc(str, pos, caches)):
104           case(Success(pos)):
105             loop(pos)
106           case(Failure()):
107             Success(pos)
108           end
109         end
110         loop(pos)
111       end
112     end
113
114     def not(MkPE(proc)):
115       MkPE^(str, pos, caches):
116         match(proc(str, pos, caches)):
117         case(Success(pos)):
118           Failure()
119         case(Failure()):
120           Success(pos)
121         end
122       end
123     end
124
125     def one_or_more(pe):
126       seq(pe, zero_or_more(pe))
127     end
128
129     def optional(pe):
130       choice(pe, empty)
131     end
132
133     def and(pe):
134       not(not(pe))
135     end
136
137     let String::C#(<-) = def_nonterm
138     let PE#(&) = seq
```

```
139     let PE#(|) = choice
140
141     export def_nonterm, nonterm, char, string, char_of, any_char, empty
142     export seq, choice, zero_or_more, not
143     export one_or_more, optional, and
144     export String::C#(<-), PE#(&), PE#(|)
145   end
146
147   export PE, Result, Success, Failure, parse, New
148 end
149
150 open DefaultEq(PEG::Result)
151
152 def PEG::Result#failure?(self):
153   self == PEG::Failure()
154 end
155
156 def PEG::Result#success?(self):
157   !self.failure?
158 end
159
160 /*
161 S <- &(A 'c') 'a'+ B !.
162 A <- 'a' A? 'b'
163 B <- 'b' B? 'c'
164 */
165
166 let s = begin:
167   open PEG::New()
168
169   "S" <- and(nonterm("A") & char('c'))
170        & one_or_more(char('a'))
171        & nonterm("B")
172        & not(any_char)
173
174   "A" <- char('a')
175        & optional(nonterm("A"))
176        & char('b')
177
178   "B" <- char('b')
179        & optional(nonterm("B"))
180        & char('c')
181
182   nonterm("S")
183 end
184
185 assert(PEG::parse(s, "").failure?)
186 assert(PEG::parse(s, "abc").success?)
187 assert(PEG::parse(s, "aabbcc").success?)
188 assert(PEG::parse(s, "aaabbbccc").success?)
189
190 assert(PEG::parse(s, "aaabbcc").failure?)
191 assert(PEG::parse(s, "aaabbbcc").failure?)
192 assert(PEG::parse(s, "aabbbccc").failure?)
```

```
193 assert(PEG::parse(s, "abababb").failure?)
194 assert(PEG::parse(s, "aabbccdd").failure?)
195 assert(PEG::parse(s, "ccbbaa").failure?)
196
197 /*
198 S <- E !.
199 E <- T ('+' T)*
200 T <- P ('*' P)*
201 P <- [0123456789]+ / '(' E ')'
202 */
203
204 let arith = begin:
205   open PEG::New()
206
207   "S" <- nonterm("E") & not(any_char)
208   "E" <- nonterm("T") & zero_or_more(char('+') & nonterm("T"))
209   "T" <- nonterm("P") & zero_or_more(char('*') & nonterm("P"))
210   "P" <- one_or_more(char_of("0123456789"))
211        | char('(') & nonterm("E") & char(')')
212
213   nonterm("S")
214 end
215
216 assert(PEG::parse(arith, "123").success?)
217 assert(PEG::parse(arith, "1+2+3").success?)
218 assert(PEG::parse(arith, "1*2*3").success?)
219 assert(PEG::parse(arith, "1+2*3").success?)
220 assert(PEG::parse(arith, "(1+2)*3").success?)
221
222 assert(PEG::parse(arith, "1++").failure?)
223 assert(PEG::parse(arith, "*3").failure?)
224 assert(PEG::parse(arith, "(1+2*3").failure?)
225 assert(PEG::parse(arith, "1+*3").failure?)
226 assert(PEG::parse(arith, "1**3").failure?)
```

1 行目から 158 行目までがライブラリのソースコードで，160 行目以降はそのライブラリを用いて実際にパーザを構成しテストを行っている．

160 行目から 195 行目までは典型的な文脈自由でない言語 $\{a^n b^n c^n \mid n \geq 1\}$ のパーザを構成しテストしている．PEG は無限先読みが可能であるためこのような言語もパーズできる．160 行目から 164 行目までは PEG による構文規則をコメントで示したもので，166 行目から 183 行目までが対応するパーザの定義である．トレイト `PEG::New` を呼び出し `open` することで，連接（`&`），選択（`|`），0 回以上の繰り返し（`zero_or_more`）等，PEG の項に対応するコンビネータが局所的に定義され，PEG とほぼ 1 対 1 に対応する形でパーザを定義できる．

197 行目から 226 行目までは単純な算術演算の式をパーズするパーザを構成しテストしている．197 行目から 202 行目までは PEG による構文規則をコメントで示したもので，204 行目から 214 行目までが対応するパーザの定義である．こちらも PEG とほぼ 1 対 1 に対応する形でパーザを定義できている．Suzu は関数や演算子をモジュールやトレイトによって局所的にまとめて定義することで，このような可読性の高い内部 DSL をグローバル環境を汚染することなく利用することができる．