2010 年度 修士論文

日本語プログラミング言語における
関数型パラダイムの表現と
実装した言語“クロガネ”の開発

提出日：2011 年 2 月 4 日
指導：筧 捷彦 教授

早稲田大学 理工学術院 基幹理工学研究科 情報理工学専攻
学籍番号：5109B017-6
太田 大地

目次

1. はじめに 3
2. 日本語プログラミング言語 4
   2.1. 既存の言語 4
   2.2. 特徴 4
   2.3. 問題 5
      2.3.1. 名詞節 6
      2.3.2. 助詞 7
3. 関数型言語 9
   3.1. 既存の関数型言語 9
   3.2. 関数型言語の機能 9
      3.2.1. 匿名関数 9
      3.2.2. 対とリスト 10
      3.2.3. クロージャ 11
4. 提案 12
   4.1. 重文 12
   4.2. 高階関数とサ変名詞 13
   4.3. 助詞“と”による対 14
   4.4. “それぞれ”による map 処理 16
5. プログラミング言語「クロガネ」の開発 18
   5.1. 設計方針 18
      5.1.1. 括弧の利用 18
      5.1.2. 破壊的代入不可能な変数 18
      5.1.3. 構文解析が用意な構文 18
   5.2. 実装 19
      5.2.1. クラス設計 19
      5.2.2. 相互運用 22
      5.2.3. プロデルライブラリの利用 23
      5.2.4. テスト 23
6. 評価 24
   6.1. 全体 24
   6.2. 重文 24
   6.3. 助詞“と”による対の作成 25
   6.4. “それぞれ”による map 処理 25
   6.5. 高階関数 26

7. まとめ 28
8. 参考文献 29

1. はじめに

プログラムを日本語らしく記述出来ることを目的とした言語，日本語プログラミング言語は，昔から開発され続けている。日本語らしく記述とは言うものの，既存の構文を日本語で表現できるだけであり，言語パラダイム，つまり手続き型言語やオブジェクト指向型言語などといったプログラムミング言語にある考え方や構造化の仕方は，従来の言語と同様のものをそのまま流用している。

日本語プログラミング言語は今までに多数開発されているが，それらの言語パラダイムは全て，手続き型言語およびその延長であるオブジェクト指向型言語である。関数型言語というパラダイムを持つ日本語プログラミング言語は存在しない。本研究では，関数型言語を日本語で表現するための構文の提案，それらの構文を実装した言語クロガネ [1]の開発，そしてクロガネを使い提案した構文の評価，の三つを行う。

2. 日本語プログラミング言語

この章では，日本語プログラミング言語について説明する。

2.1. 既存の言語

日本語プログラミング言語は昔から研究・開発されており，古くには和漢 [2]が 1983 年には存在している。現在では，なでしこ [3]が Office などの定型処理のため，開発されている。また汎用的なプログラミング言語として，プロデル [4]や，言霊 [5]など，が開発されている。プロデルは動的に型を持ったオブジェクト指向言語であり，“.NET Framework”上で動作する。言霊は静的に型を持ったオブジェクト指向言語であり，Java 仮想マシン上で動く。Java へのトランスレータとして用いることも可能である。

2.2. 特徴

日本語プログラミング言語の特徴として，日本単語，語順，助詞の利用があげられる。

日本語プログラミング言語では，変数名や関数名はもちろんのこと制御構文も日本語で記述する。C 言語では，例えば

```
if (condition) printf(“condition is true”);
```

のように英単語を用いて記述するが，日本語プログラミング言語では

```
もし条件なら,「条件を満たしています」を表示する
```

のように全て日本語で文章らしく記述する。Java などの最近の言語では関数名や変数名に日本語を用いることが出来る。Java で日本語を用いたプログラムの例を示す。

```
// 関数宣言
public 顧客 顧客を ID から検索する(int 顧客 ID) { … }
// 呼び出し
顧客 対象顧客 = null;
if (id > 0) 対象顧客 = 顧客を ID から検索する(id);
```

このように部分的に日本語が入り交じり，プログラム全体の一貫性が無くなる。

日本語プログラミング言語では，関数を呼ぶ際の語順も日常使っている日本語と同じ順序になっている。C 言語で“Hello, world”と表示するプログラムの例を示す。

```
printf(“Hello, world”);
```

同様のプログラムを Ruby [6]では次のように記述する。

```
puts “Hello, world”
```

言語により括弧の違いなどはあるが，関数の呼び出しは常に，関数，引数の順で書く。一方，日本語プログラミング言語では，引数，関数の順に書く。プロデルで“こんにちは，世界”と表示するプログラムを示す。

```
「こんにちは，世界」を表示する
```

さらに，日本語プログラミング言語では引数の順番ではなく，引数と対で記述される助詞によって，実引数と仮引数を結びつける。まず，C 言語で引き算する関数を呼び出す例を示す。

```
// 関数宣言
int sub(int a, int b) { return a - b; }
// 呼び出し
sub(5, 4);
```

このプログラムで結果は 1 になる。この関数呼び出しで a が 5 として sub が実行されるのは，a が 1 番目の仮引数であり，5 が 1 番目の実引数であるからである。次に，プロデルでの関数呼び出しの例を挙げる。

```
// 関数宣言
【A】から，【B】を，引く手順
    A-B を返す。
おわり
// 呼び出し
4 を 5 から引く
```

このプログラムでも同様に結果は 1 になる。関数呼び出しで A が 5 として扱われるのは，A に付いている助詞が“から”であり，5 に付いている助詞も同じ“から”であるからである。

2.3. 問題

日本語プログラミング言語の問題として，構文が自然な日本語に束縛されることがある。従来の C 言語のような文字を記号的に用いる言語では，プログラムを分かりやすく書くことさえ出来ればよい。しかし日本語プログラミング言語では，書いたプログラムが日本語

の表現として読めることが求められる。その結果，日本語プログラミング言語では，構文が冗長になる。

2.3.1. 名詞節

日本語プログラミング言語での関数呼び出しは，“A を B する”のような表現になる。この時プログラムが自然な日本語であることが求められるので，“A”は名詞あるいは名詞節，“B する”は動詞でなくてはならない。さて，C 言語で次のような多重に関数を呼ぶようなプログラムがあったとする。

```
program = compile(syntaxAnalyze (lexicalAnalyze(text)));
```

“lexicalAnalyze”，“syntaxAnalyze”，“compile”はそれぞれ字句解析，構文解析，意味解析を行う関数であり，“text”はプログラムのソースコード，“program”はコンパイル結果とする。同様のプログラムを日本語プログラミング言語で表現した場合，冗長な表現になってしまう。プロデルで例を示す。

```
プログラムは，（（テキストを字句解析したもの）を構文解析したもの）を意味解析したもの
```

多重の関数呼び出しを行う場合，引数が名詞あるいは名詞節でなくてはならないため，動詞の後ろに余計な“したもの”をつける必要がある。さらに，どの引数がどの関数の引数であるかを示すために括弧を用いなくてはならない。日常使う日本語では，何が何に係るかを示すための括弧は用いない。この様な括弧を用いる表現は不自然な日本語と言えるだろう。

この冗長さを避ける従来の手法として，特殊な変数“それ”と暗黙的な引数の二つがある。順に説明する。

特殊な変数“それ”は，直前に実行した結果を参照することができる変数である。なでしこやプロデルなど，採用している言語はいくつかある。日本語プログラミング言語ではないが，Perl では同様に“$_”で直前の結果を参照することができる。なでしこで“テキスト”を“プログラム”に変換する例を示す。なでしこでは，関数名のうち“する”は省略できる。

```
テキストを字句解析。
それを構文解析。
それを意味解析。
それをプログラムに代入。
```

元のプログラムと比較した場合，一時変数は増えるが，その一時変数への代入を記述する必要がないため，見やすいプログラムになっている。さらに，不自然であった括弧も無く

なっている。プログラムの行数が長くなることを除けば，有効な表現だと言える。

なでしこではさらに暗黙的引数を利用することができる。関数が呼ばれる際に，もし引数の数が足りなければ自動的に“それ”を引数として与える。同様の機能が Perl にも存在する。暗黙的引数を用いた例をなでしこで示す。

```
テキストを字句解析。構文解析。意味解析。プログラムに代入。
```

プログラムを短く記述することができた。しかし暗黙的引数にも問題もある。暗黙的引数を用いた場合，呼び出されている関数に実際に引数が渡されているかが明確ではないことである。このプログラムを読む場合，“構文解析”関数がもし引数を必要とするのなら，“テキストを字句解析”した結果を引数として“構文解析”し，そうでなければ引数を与えず“構文解析”する，と読まなくてはならない。よく使われる組み込み関数，例えば“表示する”などなら省略されていてもまだ良いが，そうでもない場合，そのプログラムの可読性・保守性を著しく下げてしまう。

2.3.2. 助詞

日本語プログラミング言語では，実引数と仮引数を順序ではなく，助詞によって結びつける。そのため，適切な助詞が存在しない，あるいは助詞が重複するような例ではうまく関数を記述できない。C 言語で曜日を計算する関数“calcWeek”があったとする。第一引数，第二引数，第三引数はそれぞれ年，月，日であり，戻り値は週（0～6）とする。

```
int week = calcWeek(2011, 1, 1);
```

このような例では年，月，日に対して適切な助詞を設定できないため，関数をうまく定義出来ない。なでしこでは，C 言語のような引数の渡し型も認めているので C 言語と同様に記述することができる。

```
曜日計算（2010, 1, 1）を曜日に代入。
```

しかしこれでは自然な日本語とは言えないだろう。プロデルなどオブジェクト指向言語では日付をオブジェクトとすることで，日本語らしく表現することができる。

```
日付を作り，正月とする。
正月の年は，2010。
正月の月は，1。
正月の日は，1。
曜日は，正月から曜日計算したもの。
```

プログラムは分かりやすく，かつ自然な日本語で記述することができた。しかし，元の C

言語のプログラムと比較するとプログラムが長くなってしまっている。

3. 関数型言語

この章ではいくつかの関数型言語と，関数型言語に実装されている機能について説明する。

3.1. 既存の関数型言語

1958 年に登場した LISP [7]をはじめとし,現在関数型言語と呼ばれる言語は数多くある。Scheme [8]は，LISP の方言の一つである。LISP 同様，動的なスクリプト言語である。Haskell [9]は，型付き関数型言語である。すべての処理が副作用を持たないため，純粋な関数型言語とも言われる。OCaml [10]は型付き関数型言語である。純粋性より実用性を重視し，オブジェクト指向型言語でもある。JavaScript [11]は，プロトタイプベースオブジェクト指向プログラミング言語である。Web アプリケーションに主に使われる。手続き型言語ではあるのだが，関数が第一級オブジェクト（後述）であることや，クロージャ（後述）を持つことから，関数型言語として扱われることもある。

3.2. 関数型言語の機能

この節では，関数型言語でよく実装されている機能について説明する。

3.2.1. 匿名関数

匿名関数とは，名前の付けられていない関数を指す。ラムダ式あるいはラムダ関数と呼ばれることもある。すべての関数型言語は，関数を通常の値と同じように扱うことができる。このように通常の値のとして扱うことができるものを第一級オブジェクト（First-Class Object）と呼び，関数を引数とする関数や，関数を戻り値とする関数を高階関数と呼ぶ。

匿名関数を Haskell [9]で説明する。Haskell で引数を 1 だけ増やした値を返す関数 succ は次のように定義する。

```
succ x = x + 1
```

関数を引数に取り，その引数に 1 を適用する関数 applyOne は次のように定義する。

```
applyOne f = f 1
```

この applyOne 関数に succ 関数を渡すことで，2 を結果として得ることができる。

```
two = applyOne succ
```

もし，この succ 関数がこの場所でしか利用されない場合，succ を渡す代わりに匿名関数を

用いて，次のように記述することができる。

```
anotherTwo = applyOne (¥x -> x + 1)
```

このように匿名関数を用いることで，関数の不要な宣言がなくなり，プログラムを短く記述することが可能になる。さらに関数の中身が記述されている場所と関数に渡されている場所が近くなることで，可読性の向上も見込まれる。一方，匿名関数が何重にも入れ子にすることで，対応が分かりづらくなり，外側の関数が肥大化し，可読性は低下する。匿名関数の利用は注意しなくてはならない。

3.2.2. 対とリスト

対は，順序のある二つの要素からなる最も単純なデータ構造である。ここでは対の各要素を，前者，後者と呼ぶことにする。さらに，後者がさらに対であるような対をリストと呼ぶ。対とリストは関数型言語において頻繁に用いられるデータ構造である。

複数のデータを扱いたいとき，手続き型言語では主に，配列を用いる。一方，関数型言語では，リストを用いることが多い。この理由として，関数型言語では破壊的代入を避けることが挙げられる。配列の場合，要素を一つ変更するたびに配列を新しく作りなおさなくてはならない。一方，対の連続であるリストの場合，先頭を変更する場合に限れば，最初の要素を新しく作り直すだけで要素の変更が行える。Java [12]で要素が 1 から 10 までの配列を作るプログラムを示す。

```
int[] numbers = new int[10];
for (int i = 0; i < numbers.length; i++)
    numbers [i] = i + 1;
```

Haskell [9]で要素が 1 から 10 であるリストを作成するプログラムを示す。

```
numbers = mkList 1 10 where
  mkList start end =
    if start > end then [] else start : mkList (start + 1) end
```

また，これらのデータの集合をプログラムで扱うことを考える。手続き型言語の場合，配列の各要素への処理は for ループを用いて行われ, その時, 添字を書き換えながら操作する。さらに集約するための変数も書き換えることが多い。Java で要素の合計を計算するプログラムの例を挙げる。

```
int sum = 0;
for (int i = 0; i < numbers.length; i++)
    sum += numbers[i];
```

この例では，合計を記憶するための変数“sum”と添字である“i”が何回も書き換えられている。関数型言語では，破壊的代入を避けるため，ループ文を書けない。必然的に再帰関数で各要素の処理が行われる。その場合も配列よりリストであったほうがプログラムを書きやすい。Haskell で要素の合計を計算するプログラムの例を挙げる。

```
sum = sum_iter numbers where
    sum_iter (x:xs) = x + sum_iter xs
    sum_iter []     = 0
```

このようにリストを扱うプログラムは，関数を要素がある場合と要素がない場合の二つに分けることができる。再帰関数でプログラムを書く場合，リストを用いたものは，配列と添字を用いたのと比べ綺麗に書くことができる。

3.2.3. クロージャ

C [13]や Java [12]といった言語と異なり，関数型言語では関数を抜けたあとも関数で用いられる変数が保持される場合がある。この変数が保持される機能をクロージャ(closure)と呼ぶ。Haskell でを例を示す。“cons”は対を作る関数であり，“head”および“tail”はそれぞれ対の前者と後者を取り出す関数である。

```
cons h t = (¥x -> if x then h else t)
head pair = pair True
tail pair = pair False
```

次にこの関数を用い，対を作り，その後その対の要素を取り出すプログラムを示す。

```
pair = cons 3 5
three = head pair
```

最初に 3 と 5 を引数に cons 関数が呼ばれている。この関数は高階関数であり，戻り値として,真偽値を受け取り対の要素を返す関数を返す。次の head 関数で受け取った関数に“True”を与えることで対の前者である 3 を得る。C や Java などの言語では，この cons 関数が実行された後は，引数として渡された 3 や 5 はメモリ上には残らず，参照することができない。また，そのような言語はそもそも関数を返せるような仕組みにはなっていない。一方，クロージャを持つ言語では，関数が呼ばれたときに渡された環境やその実行途中の状態をメモリ内に保存する。そのため，参照する方法さえあれば，関数を抜けたあとも参照することができる。

4. 提案

この章では，日本語プログラミング言語における，いくつかの新しい表現を提案する。また必要であれば，提案を実装した言語クロガネ [1]を説明に用いる。

4.1. 重文

既存の日本語プログラミング言語において，関数を連続して呼ぶ表現は，やや冗長に表現しなくてはならない。

```
プログラムは，（（テキストを字句解析したもの）を構文解析したもの）を意味解析したもの
```

これは日本語らしくプログラムを書くことを追求した結果，引数を必ず名詞として表現する必要が出てきたためである。

この問題は，重文と関数の部分適用を用いることで解決することができる。例えば先ほどのプログラムは次のように表現することができる。

```
テキストを字句解析し，構文解析し，意味解析し，プログラムとする。
```

“テキストを字句解析し”の部分は今まで通りの関数の実行と同じように扱う。次の，“構文解析し”の部分では，直前の“字句解析”した戻り値を“構文解析”の引数として関数を実行する。さらに “構文解析”した結果を“意味解析”の引数として渡す。このように直前の関数の戻り値を次の関数の引数として，順に実行することで，名詞として表現するといった冗長さを取り除くことができる。

この方法は，関数の引数が二つ以上ある場合でも問題なく動く。二つの引数を受け取る関数を用いた例をあげる。“反転”は受け取ったリストを逆順に並べたリストを返す関数，“連結”は二つのリストを受け取り，“に”で受け取った方を左側，“を”で受け取った方を右側として連結する関数とする。

```
[1,2,3]を反転し，[1,2,3]に連結する。※戻り値は[1,2,3,3,2,1]。
```

このように，引数が二つ必要な関数であっても，重文の各“句”の中で他の引数を与えればよい。このような結果をひとつ持ちながら順に実行する構文は他の関数型言語でも見られる。例えば F# [14]では“順次パイプライン演算”として演算子が定義されている。さらに助詞を用いることを，日本語プログラミング言語としての長所として利用できる。

```
[1,2,3]を反転し，[1,2,3]を連結する。※戻り値は[3,2,1,1,2,3]。
```

これは先ほどのプログラムとほぼ同じである。しかし“連結”関数で用いられている助詞が“に”ではなく“を”である。その結果プログラムの実行結果は異なる。これは日本語プログラミング言語の特徴である，引数の順序ではなく助詞によって実引数と仮引数を関連付ける機能を応用したものである。従来の関数型言語では，引数を先頭からしか適用できなかったが，日本語プログラミング言語では任意の場所の引数を適用できる。

4.2. 高階関数とサ変名詞

日本語プログラミング言語において，引数は名詞の形で，関数は動詞の形で表現される。言語が高階関数を認める場合，関数を引数として与えることができなくてはならない。そこでサ変名詞を関数名として用いることにする。サ変名詞とは名詞の直後に“する”を付けることで動詞として扱える名詞のことを指すものとする。

先行評価(eager evaluation)される言語において， 1 と実行結果が 1 である関数とは異なる。Scheme [8]を例にする。“a”と“c”の値は 1 である。 “b”と“d”の値は 1 を返す関数である。Scheme では関数自身を関数名によって表し，関数の実行は“(b)”の様に，その関数名を括弧で括り表す。

```
(define a 1)
(define (b) 1)
(define c (b))
(define d b)
```

同様の例を JavaScript [11]で示す。JavaScript の場合，関数の後ろに括弧をつけ“b()”のようにすることで関数を実行する。

```
var a = 1;
var b = function(){ return 1; };
var c = b();
var d = b;
```

日本語プログラミング言語においても同様に，関数そのものの表現と関数を実行した結果を表す表現は異なっていなくてはならない。そこで，関数名をサ変名詞に限定し，関数名の後ろに“し”あるいは“する”を付けることで関数の実行を示すものとする。“し”は重文にするときに用い，“する”はそうでないときに用いる。Scheme や JavaScript と同様の例をクロガネで示す。甲と丙の値は 1 である。乙の丁の値は 1 を返す関数である。

```
１を甲とする。
以下の定義を乙とする。１である。以上。
```

```
乙し，丙とする。
乙を丁とする。
```

4.3. 助詞“と”による対

関数型言語では，対や連続した対であるリストはよく用いられるデータ構造である。対は頻繁に用いられるため構文的に簡単に作れることが望ましい。一方，日本語で物を並列に表現しようとした場合，助詞“と”を用いる。助詞“と”は一つではなく，例えば“甲と乙と丙”のように，“と”が複数あっても日本語として自然である。そこで対をつくるために “と”を特殊な助詞として扱うことにする。例えば１と２からなる対は，次のように作成する。

```
１と２である。
```

また，１，２，３の三つの要素からなるリストは次のように作成する。

```
１と２と３と無である。
```

ただしこの“無”はプログラム中で用いられる特殊な値であり，リストの終端を示す。このように特殊な値を終端に用いることはリスト構造で一般的である。 Scheme では“'()”, Haskell [9]や OCaml [10]では“[]”が用いられる。リストは，リストの先頭から取得されることが想定されているため，助詞“と”は通常の演算子とは逆の右優先結合でなくてはならない。対の各要素はそれぞれ特殊な助詞“の”を用いて，“頭”，“体”によって取得できるものとする。例として“雷神”，“風神”を順に表示するプログラムを挙げる。

```
「雷神」と「風神」を神とする。
神の頭を表示する。
神の体を表示する。
```

助詞“の”は，助詞“と”よりも結合優先度を高く設定する必要がある。リストの１番目と２番目を入れ替え，第二リストに代入するプログラムを例に挙げる。

```
リストの体の頭とリストの頭とリストの体の体を第二リストとする。
```

助詞“の”の結合優先度が助詞“と”よりも低い場合，同じプログラムは次のように括弧が必要になってしまう。

```
(リストの体の頭)と(リストの頭)と(リストの体の体)を第二リストとする。
```

“と”の結合優先度が“の”の結合優先度より高い場合，得られる恩恵は

```
（「雷神」と「風神」）の頭を表示する。※「雷神」が表示される。
```

の様に対を作った直後に対の要素を参照する場合だけであり，そのようなプログラムは基本的に書かれない。

助詞“と”を，対を作る助詞とした場合，そのままでは関数宣言時の助詞として“と”が利用できない。例えば次のようなプログラムを考える。

```
3 と 5 を加算する。
```

この例では３が“と”に，５が“を”に結び付けられるのではなく，３と５の対が“を”に結び付けられてしまう。そこで，関数宣言時に“と”が利用されている場合，対を自動的に展開すれば，関数宣言時に“と”が利用できる。例を示す。

```
以下の定義で A と B を加算する。
A+B である。
以上。
```

このように定義された関数は,

```
以下の定義で対を加算する。
    対の頭を A をとする。
    対の体を B とする。
    A+B である。
以上。
```

であるかのように振舞う。そうすることで,

```
3 と 5 を加算する。
```

のような関数呼び出しにおいて,“と”がまるで通常の助詞であるか利用することができる。また当然

```
2 と 6 を対とする。対を加算する。
```

のように,関数呼び出しの際に対を作らない場合であっても実行することができる。逆に,本来の特徴であった助詞によって引数の順序を入れ替えることができなくなっている。例えば次のように対を作らないプログラムは動かない。

```
5 を 3 と加算する。
```

“と”を対の展開とすることで，関数宣言内に複数の“と”を利用することができる。三

つの値を加算するプログラムを示す。

```
以下の定義で A と B と C を三重加算する。A+B+C である。以上。
3 と 5 と 7 を三重加算する。※結果は 15。
```

さらに,“と”がそれぞれ別の対を展開するプログラムも記述することができる。2 点間の距離を計算するプログラムの例を示す。ただし“√”は平方根を計算する単項演算子とする。

```
以下の定義で X1 と Y1 から X2 と Y2 まで距離計算する。
    √((X2-X1)×(X2-X1) + (Y2-Y1)×(Y2-Y1))である。
以上。
1 と 1 から 4 と 5 まで距離計算する。※結果は 5。
```

4.4. “それぞれ”による map 処理

集合の各要素に対して同じ処理を繰り返すようなプログラムはよく記述される。これは関数型言語として map 関数として一般的な処理である。Scheme で map 関数を利用して各要素を 1 だけ増やすプログラムを示す。このプログラムの結果は“(2 3 4 5 6)”となる。

```
(define (succ x) (+ x 1))
(map succ (list 1 2 3 4 5))
```

これをそのまま日本語にすると次のように表現できる。

```
以下の定義で値を増加する。値と 1 を加算する。以上。
[1, 2, 3, 4, 5]を増加で射影する。
```

Scheme [8]では lambda 式を用いて,一つの関数で同様のプログラムを書くことができる。

```
(map (lambda (x) (+ x 1)) (list 1 2 3 4 5))
```

このような表現を日本語で記述したいと考えたとき，問題がある。それは射影する関数に対する引数が与えられないことである。ラムダ式を使えない場合，このようなプログラムを書くことができない。この問題を“それぞれ” キーワードを用いて解決する。射影処理は必ず射影されるリストは常に一つの引数として与えられる。一方，射影する関数はいくつ引数が与えられるかわからない。そこで射影される値を先頭にし，その直後に“それぞれ”を記述し，その後ろに射影する関数を書くことで，射影する関数にも任意の数の引数を与えることができる。“それぞれ”を用いてリストの要素を一つずつ増加させるプログラ

ムは次のように記述できる。

[1, 2, 3, 4, 5]をそれぞれ１と加算する。

このプログラムは，結果として“[2, 3, 4, 5, 6]”を得ることができる。

5. プログラミング言語“クロガネ”の開発

本研究では関数型言語としての機能を持った日本語プログラミング言語“クロガネ”を開発した。この章ではクロガネについて説明する。

5.1. 設計方針

この節では，クロガネを作成する上での設計方針を説明する。これは，4 章で提案した内容以外で，クロガネの言語仕様に大きく影響する。

5.1.1. 括弧の利用

日常的に使う日本語において，文章中に括弧は，省略や補助的な説明を示す。優先順位や，係りの対応を示すために用いることはない。そこでクロガネでは，文章中に括弧を用いることができないような文法に制限する。

5.1.2. 破壊的代入不可能な変数

関数型言語では, 変数の破壊的代入を避ける。例えば Haskell [9]では行うことができず, Scheme [8]や OCaml [10]では破壊的代入のために特別な変数や構文を用意している。そこでクロガネも同様に破壊的代入を行えないような仕様にする。

5.1.2.1. 変数の例外

スクリプト言語は，プログラムの途中でプログラムを動的に生成し実行することで，柔軟に動作する。これはスクリプト言語の大きな長所である。動的なコード生成は LISP [7]や Scheme [8]，JavaScript [11]，Ruby [6]などで広く利用されている。クロガネでもこれを踏襲し動的な側面を持たせたい。しかしこれは破壊的代入を不可能とする仕様と競合する。そこでクロガネでは，グローバル変数のみを破壊的代入可能な変数とし，それ以外のローカル変数では破壊的代入を許可しない。

5.1.3. 構文解析が用意な構文

日本語プログラミング言語で書かれたプログラムを構文解析するのは難しい。プロデルで例を挙げる。

```
登録し終えた顧客を表示する。
```

このようなプログラムでは，変数と助詞と関数を見分けるのが難しい。人間が読む場合は文脈から判断することも可能だが，プログラミング言語であるため，これを機械的に処理しなくてはならない。そこで，クロガネでは変数にはひらがなを使うことができず，逆に

助詞にはひらがなのみを利用できる仕様にする。これにより，クロガネでの字句解析器を簡単に実装することができる。

また，このような言語仕様にすることで，プログラムの可読性を上げることができる。漢字とひらがなではほとんどの場合，漢字のほうが文字の密度が高い。英単語を用いてプログラミングする言語において，キャメル記法（単語の頭を大文字にし，続く文字を小文字で表記する方法）を用いて単語を読みやすくするのと同程度の可読性の向上が見られると考えられる。これにより，プログラムを読む際に，変数と助詞の区別がしやすい。

5.2. 実装

この節では，クロガネの実装について説明する。今回クロガネは C# [15]で実装した。クロガネは“.NET Framework 4 Client Profile”上で動作する。クロガネのソースコードは http://kurogane.codeplex.com/ でオープンソース（Apache License 2.0）で公開している。

5.2.1. クラス設計

クロガネは次のように設計した，主要なクラスと名前空間を列挙する。

- Kurogane.Engine クラス
- Kurogane.Compiler 名前空間
  - Lexer クラス
  - Token 抽象クラス
  - Parser クラス
  - Ast 抽象クラス
  - ExpressionGenerator クラス
- Kurogane.Expressions.ExpressionOptimizer クラス
- Kurogane.RuntimeBinder 名前空間
  - ArithmeticBinder クラス
  - KrgnInvokeBinder クラス
- Kurogane.Libraries
  - StandardIO クラス
- Kurogane.Shell.ReplEngine クラス
- Kurogane.SimpleEditor.StartForm クラス

Kurognae.Engine クラスは，クロガネのスクリプトを動かすための最初のクラスである。また，クロガネのスクリプトを動かすだけであれば，このクラスのみを参照すれば良い。C#からクロガネを用いて，“こんにちは”と表示するプログラムを示す。

```
var engine = new Kurogane.Engine();
```

```
engine.Execute(“「こんにちは」を表示する。”);
```

Kurogane.Compiler 名前空間は，クロガネのスクリプトを実行開始時に抽象構文木に変換するためのクラス群を含む名前空間である。5.1.3 節で説明した構文解析器しやすい文法に限定することにより，実行前に抽象構文機に変換することが可能になる。またこの抽象構文木を標準ライブラリに存在する式木（System.Linq.Expressions.Expression のサブクラスによる木）に変換し，コンパイルすることで，高速化する。これにより，プロデルなど，実行時にしか構文解析できない言語と比較してクロガネは高速に動作する。

Kurogane.Expression.ExpressionOptimizer クラスは作成された式木に対し最適化を書けるためのクラスである。このクラスにより，クロガネで記述された末尾再帰している関数を GOTO 文に置き換える。ただし，この末尾再帰最適化は自身を呼んでいるような再帰関数にしか適用できない。

Kurogane.RuntimeBinder 名前空間に属するクラスは，クロガネの動的な型処理を助ける。例えば，ArithmeticBinder は算術演算を助ける。例えば加算を行うようなプログラムでは，左辺と右辺が共に整数であった場合，整数の算術命令を生成する。さらにキャッシュを作成し，同じコードを実行する限り，以降両辺が整数である限り，すでに生成された算術命令を利用する。これにより二回目以降の速度が大幅に上昇する。KrgnInvokeBinder クラスは助詞を用いた関数呼び出しをキャッシュする。この名前空間内には同様のクラスが多数存在する。

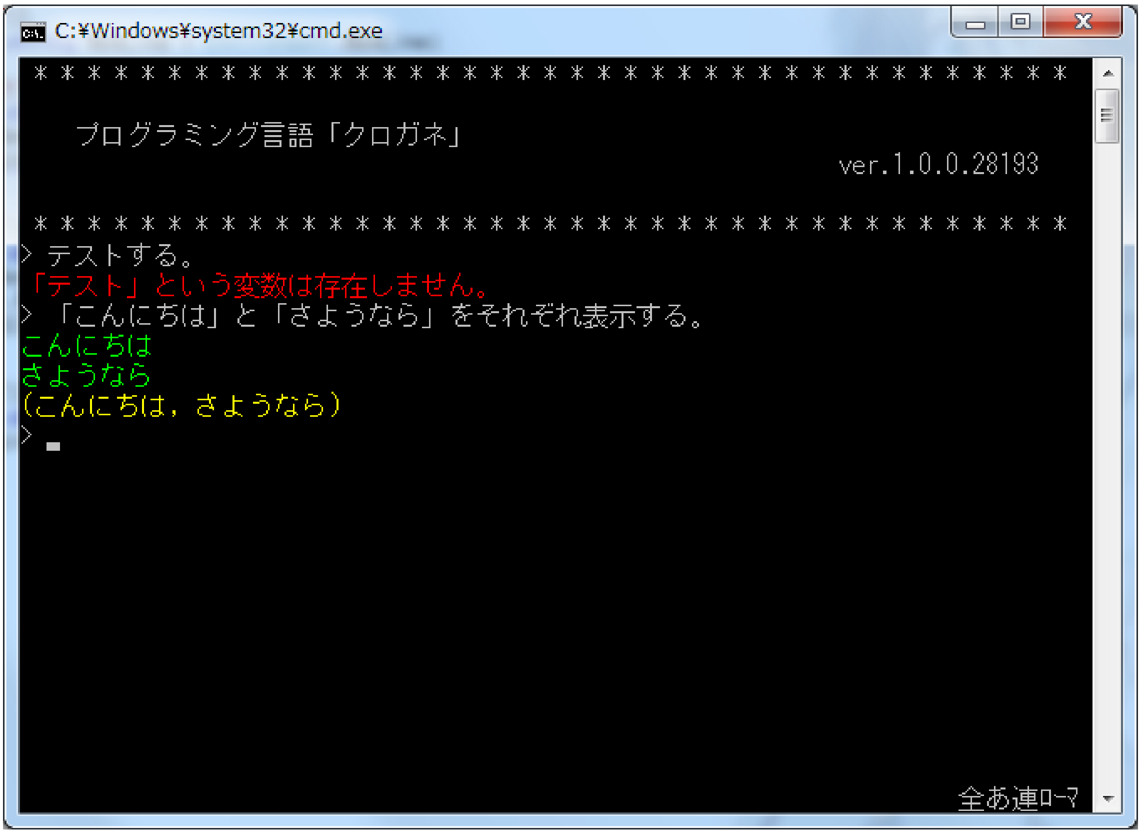
Kurogane.Libaries 名前空間は，クロガネで用いられる標準ライブラリを担当する。例えば，StandardIO クラスでは，“出力する” などといった入出力用の関数が定義されている。StandardIO クラスのプログラムを一部示す。

```
[Library]
public static class StandardIO {
    [JpName("出力")]
    public static object Write(
        this Engine engine, [Suffix("を")]object obj)
{
        engine.Out.Write(StringLib.ToString(obj));
        return obj;
    }
}
```

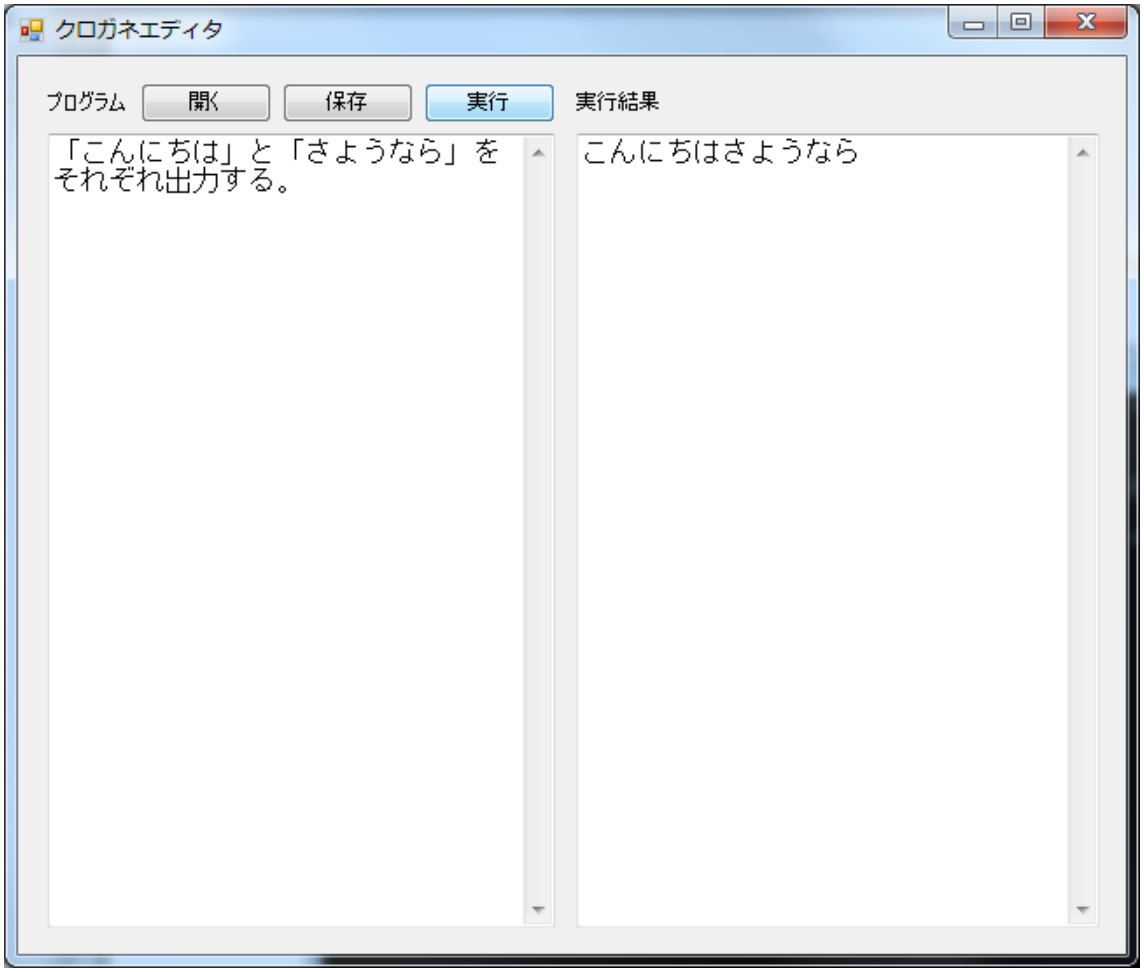
クラスに “Library” 属性を付けることで，そのクラスにクロガネで用いられる関数や定数が定義されていることを示す。“JpName” 属性で名前を指定することで，クロガネで用いられる関数名を指定することができる。“Suffix” 属性で助詞を指定することで，その関数

で用いられる助詞を定義することができる。

Kurogane.Shell.ReplEngine は，クロガネをコンソールで実行するためのクラスである。REPL とは Read-Eval-Print-Loop の略である。このクラスによりクロガネはコンソールで対話的にプログラムを実行することができる。



Kurogane.SimpleEditor.MainForm クラスは，GUI でのテキストエディタのクラスである。このテキストエディタにより，プログラムを実行できる。さらにこのテキストエディタは，エラーが発生した際，に発生したプログラムの行番号などを報告する。



5.2.2. 相互運用

クロガネは，他の言語から使われることを想定して作られた。他の.NET 言語からクロガネのプログラムを利用することができる。クロガネで関数を定義し，C#から利用する例を挙げる。このプログラムを実行することで，変数“seven”の値は 7 になる。

```
var engine = new Kurogane.Engine();
engine.Execute(@"
以下の定義で A に B を加算する。
    A+B である。
以上。
");
dynamic 足す = engine.Scope.GetVariable("加算");
int seven = 足す(に:3, を:4);
```

5.2.3. プロデルライブラリの利用

プロデル [4]は，C# [15]で実装されており，標準ライブラリも C#で書かれている。クロガネも C#で実装されているため，原理的にはプロデルの標準ライブラリを利用することが可能である。プロデルはオブジェクト指向型言語，クロガネは関数型言語と言語パラダイムが異なるため，標準ではプロデルのライブラリの利用をサポートしなかった。しかし，実験的に，プロデルのライブラリをクロガネから呼んでみたところ動作した。クロガネからプロデルのライブラリを参照できる処理系は Kurogane.Interop.Produire 名前空間に存在する。

5.2.4. テスト

クロガネは，クロガネの持つ言語仕様全てと，入出力など言語内で完結しないライブラリを除く全ての標準ライブラリに対して単体テストを記述してある。例として，整数の計算をテストする関数を挙げる。

```
[TestMethod]
public void 整数の計算() {
    Assert.AreEqual(3, Execute<int>("（１＋２）である。"));
    Assert.AreEqual(3, Execute<int>("（１９９－１９６）である。"));
    Assert.AreEqual(6, Execute<int>("（３×２）である。"));
    Assert.AreEqual(2, Execute<int>("（１１÷４）である。"));
    Assert.AreEqual(3, Execute<int>("（１１%４）である。"));
}
```

このテストには Visual Studio 2010 に付属しているテストツールを用いた。クロガネの処理系が扱う値は.NET での値と同等のものである。よって，クロガネのテストを行うために，Visual Studio をそのまま使うことができる。クロガネの処理系は適切に構造化されており，このようにクロガネのプログラムを C#のコードに直接埋め込んでテストを行うことが可能である。なお，IDE 付属のテストツールを用いることで，IDE から一連のテストを全て行うことができ，テストを通過したかの確認も容易に行える。

6. 評価

この章では，今回作成した言語クロガネを評価する。最初の節ではクロガネの全体の評価を行う。続く節では 4 章で提案した各機能を三つの観点からそれぞれ評価する。

- ✓ 可読性：使いやすいか，プログラムの意図を理解しやすいか
- ✓ 汎用性：利用される機会は多かったか
- ✓ 適用性：他の日本語プログラミング言語でも利用できるか，した方がよいか

6.1. 全体

クロガネの日本語プログラミング言語的側面，すなわちプログラムを日本語の文らしくかけることに注目した場合，クロガネは良い言語だと言える。続く節で詳しく評価するが，新しく取り入れた様々な構文は，プログラムを日本語の文章として読むのを大いに助けた。

クロガネを関数型言語として評価した場合，クロガネは従来の関数型言語と比べてそれほど簡潔にプログラムを記述できない。これはクロガネが日本語としての表現に縛られてしまった結果である。特に日本語らしい表現のまま，匿名関数を表現することができない。関数名は省略したまま，助詞を利用できるような関数宣言は難しく，また一つの文の中に入れ子の状態で関数を宣言することは日本語として不自然な形になった。これらの制限により，従来の関数型言語で書かれたものと同様のプログラムをクロガネで書こうとしても綺麗にいかない。

クロガネを実用的な言語として評価した場合，まだまだ不十分な言語である。今回作成したクロガネは実験的に開発した言語であったので仕方ないとも言える。

6.2. 重文

重文を用いることで可読性を向上させることができる。なでしこ [3]で実装されている暗黙的に引数を渡す構文と比較した場合，重文では引数が渡されていることが明確である点が優れている。さらに多くの言語で実装されている，直前の値を示す“それ”を用いた場合よりもプログラムを短く記述することが可能である。

汎用性も高い。ほとんどのプログラムは計算した結果をそのまま別の計算に利用する。そのため重文を用いる機会は多い。さらに言語の構文次第では，代入にも重文の構文を適用することが可能である。

適用性は高めである。重文の構文は従来の構文と競合しないため，新しく追加することが容易であるといえる。ただしクロガネでは用いられる関数はすべてサ変名詞であった。従来の日本語プログラミング言語では“加える”などの動詞を用いることもあり，これは“加え”のように語尾が変化する。重文を表現するためには動詞の連用形がわからないと

解析できないので，何らかの工夫が必要になる。プロデルなど，動詞を“～したもの”と活用し，引数に利用できる言語もある。これらの言語では，すでに“加えたもの”のようなプログラムも構文解析する機能が備わっており，これを利用することでそのまま連用形を解析することが可能である。

6.3. 助詞“と”による対の作成

“と”を用いた関数の呼び出しや対の展開により可読性は向上した。従来では無理やり構造体をつくるか，あるいは C [13]の様な，括弧と読点による引数の渡しをするしかなかった。“と”を用いることで，日本語らしさを失わず，C 言語と同等のものが表現できる。クロガネでは変数にひらがなを用いることができないので，引数と助詞の区切りが明白になり，ますます分かりやすい。一方，複数の要素を持つリストを一気に作成するような場合，“と”による表現をするのは可読性が落ちる。Haskell [9]や OCaml [10]の様な言語では，対を作るときに括弧や空白などで対の作成であることを明示する。日本語プログラミング言語の場合，わざわざ日本語として不自然な括弧を付けることはためらいがある。要素が三つ以上つまり二つ以上の対を一行の文で作成する場合は，“と”による作成より記号を用いた構文を利用したほうが良いだろう。

汎用性は高い。プログラム中ではデータの集合を利用することはよくあり，初期化などを除き要素は概ね，一つずつ追加あるいは削除される。“と”による対はよく使われた。

適用可能性は高くない。従来の言語では“と”を通常の助詞として利用出来るものが多い。それらにこの機能を適用してしまうと，今までのプログラムが動かなくなってしまう可能性が高い。もしこの機能を実装するのであれば，新規の言語に適用するか，あるいは Python [16]のバージョン 3 や, Perl6 [17]ように従来のプログラムが動かなくなるという告知が必要になる。

6.4. “それぞれ”による map 処理

“それぞれ”による map 処理は高い可読性を持つ。要素が一つである場合，

```
要素を表示する。
```

要素が複数であった場合は，

```
要素をそれぞれ表示する。
```

と，“それぞれ”というキーワードの違いで単数か複数かの処理を判別できる。またクロガネの場合は，変数名や関数名にひらがなを用いることができない。これにより“それぞれ”のような長いひらがなの文字列はここでしか現れない。それにより，プログラムが複数の

要素を対象とした物であることを分かりやすくさせている。

この構文の汎用性は低い。少なくともクロガネにおいて，それほど多くは使われなかった。最大の理由としては，出力を除き多くのプログラムは射影より集約を用いることが多いということである。それぞれを二倍する場合や別に型に変換する場合と比べ，和や最大値を求める場合や，条件に一致する要素を探す場合などのほうが多い。破壊的代入を認めるような言語であればもう少し利用される状況は広がると考えられる。

適用可能性は高いと考えられる。この構文は既存の構文に対して“それぞれ”というキーワードが入っただけの文であるので，容易に拡張できる。振る舞いとしては，この構文を既存の言語に適用した場合，走査できるのはリストだけなのか，配列やJava [12]でいうIteratorクラスの様な場合でも利用出来るのか考えなくてはならない。また今回は関数型言語に適用したため，全ての関数は値を返した。しかし手続き型の多くは値を返さない関数も多く用いられる。そのような場合，“それぞれ”がどのように振舞うべきなのかは考えなくてはならない。

6.5. 高階関数

今回の構文では，可読性が高いまま，高階関数を表現することができた。関数名をサ変名詞に統一することで，高階関数を分かりやすく引数として渡すことができた。さらに，戻り値が関数である高階関数も，内部で定義した関数をそのまま返すだけであるので，可読性の低下にはつながらなかった。

汎用性は低い。実際のプログラムで高階関数が使われる場面は少なかった。射影や集約で高階関数が使われることを想定していた。しかし射影を用いる場面はそれほど多くなく，“それぞれ”を用いるだけで十分だった。集約を行うような場面は多かったのだが，クロガネでは匿名関数を宣言することができない。その結果，集約関数に合わせた関数を宣言するより，自分自身でループする再帰関数を書いたほうが，可読性が高くなり，そちらを利用することが多かった。

高階関数を他の日本語プログラミング言語で用いることを考えた場合，難しい。関数を引数として与える場合を考える。従来の日本語プログラミング言語では関数名をサ変名詞として制限していない。そのため，引数として渡すための特殊な構文が必要になる。例えば

```
リストを加算で集約する。
```

というプログラムの代わりに，

```
リストを加えることで集約する。
```

というように，連体形＋“こと”という文を用いて動詞を名詞節として扱う必要があるだ

ろう。関数を戻り値として返すような関数も同様に記述することができる。ただしそうする場合，クロガネと同様，関数内に関数を宣言できるような言語仕様でなくては高階関数を有効に利用することができない。また，クロージャを作る必要があるため，言語仕様を大きく変更する必要が出てくる。JavaScript [11]や C# [15]のような手続き型言語において，関数オブジェクトを用いるような例に，関数のソートや，GUI のイベントハンドリングなどがある。 Java [12]などの関数オブジェクトを持たない言語では関数オブジェクトを作成する代わりに，例えば“actionPerformed()”関数を持つインタフェースを実装したインスタンスを渡す。プロデルや言霊のようなオブジェクト指向の言語では，関数オブジェクトを用いなくても，代わりにオブジェクトを用いることでイベンドハンドリングを用いたプログラムは作成できる。

7. まとめ

本研究により，関数型言語を表現しようとしたとき，全てを日本語らしく書くのは難しいことがわかった。ただし，関数型言語を表現する上で用いた表現は，関数型言語ではない日本語プログラミング言語においても，有用であると言えた。これらの表現が既存の日本語プログラミング言語に適用されたとしたら，今以上に簡潔に日本語らしくプログラムが書けるような言語になるだろう。

8. 参考文献

1. 日本語プログラミング言語「クロガネ」. (オンライン) http://kurogane.codeplex.com/.

2. **鈴木孝則.** 日本語プログラミング言語『和漢』. 出版地不明 : 報処理学会マイクロコンピュータ研究会資料(29-2), 1983.

3. プログラミング言語「なでしこ」公式ホームページ. (オンライン) http://nadesi.com/.

4. 日本語プログラミング言語「プロデル」. (オンライン) http://rdr.utopiat.net/.

5. 言霊コミュニティサイト - 日本語で楽しくプログラミング. (オンライン) http://garuda.crew.sfc.keio.ac.jp/kotodamaCommunity/.

6. オブジェクト指向スクリプト言語 Ruby. (オンライン) http://www.ruby-lang.org/ja/.

7. **McCarthyJohn.** Recursive Functions of Symbolic Expressions. 1960.

8. The Revised6 Report on the Algorithmic Language Scheme. (オンライン) http://www.r6rs.org/.

9. The Haskell Programming Language. (オンライン) http://www.haskell.org/haskellwiki/Haskell.

10. The Caml Language. (オンライン) http://caml.inria.fr/.

11. **InternationalEcma.** Standard ECMA-262 ECMAScript Language Specification . (オンライン) http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf.

12. Standard ECMA-334 C# Language Spedcification. (オンライン) http://www.ecma-international.org/publications/standards/Ecma-334.htm.

13. java.com: あなたと JAVA, 今すぐダウンロード. (オンライン) http://www.java.com/ja/.

付録：クロガネ言語仕様

Ground

2011/02/04

内容

全体 3
値 4
  値の種類 4
    無 4
    真偽値 4
    整数 4
    小数 4
    文字列 4
    対 4
    関数 5
  値の演算 5
    等価演算 5
    論理演算 5
    算術演算 5
    大小比較演算 5
    文字列結合演算 6
変数とスコープ 7
  変数 7
  スコープ 7
  代入 7
関数 8
  関数の実行 8
構文 9
  プログラム 9
  文 9
  関数定義文 9
  条件文 10
  通常文 10
  句 10
  代入句 11
  定義句 11
  呼び出し句 11
  式 12
  要素 13

リスト 14
リテラル値 14
標準ライブラリ 15
入出力 15
その他 15
サンプルプログラム 16
サンプル１ 16
サンプル２ 16
サンプル３ 16

全体

クロガネは，プログラムを日本語らしく書くことを目的としたプログラミング言語である。クロガネは，動的に型を持つスクリプト言語である。

クロガネのプログラムは一つ以上のファイルから成り立つ。プログラムの実行は，クロガネ処理系にプログラム文章の書かれているテキストファイルを渡すことで行う。渡すことのできるファイルは一つであるが，渡されるファイルの中で別のファイルを参照することで，複数のファイルを利用したプログラムを作成することも可能である。

プログラムを実行している途中でエラーになった場合，プログラムの実行は終了する。その時処理系は，可能な限りエラーになった原因や位置を報告する。

値

値の種類

クロガネの中で扱える値は次のうちのいずれかに分類できる。

- 無
- 真偽値
- 整数
- 小数
- 文字列
- 対
- 関数

無

無は，何も無いことを示す値である。検索関数において発見されなかった場合や，リストの最後の要素などとして利用される。

真偽値

真偽値は，真か偽かのいずれかの値である。真は正しいことを示し，偽は間違っていることを示す。分岐式の条件などで，よく用いられる。

整数

整数は，任意長の整数を表す。整数の計算において，オーバーフロー，アンダーフローは発生しない。

小数

小数は，IEEE 754 で規定された 64bit で表される小数である。

文字列

文字列は，文字の順序を持つ列である。プログラムが印字する際に必要である。

対

対は，二つの値を持つ値である。二つの値は前者，後者で区別される。対の後者が対であり，さらにその対の後者が対であり，と続くような対を特別にリストと呼ぶ。リストの最後の対の後者は無である。

対の前者と後者は，プログラム中では“頭”，“体”で表される。

関数

関数は，実行することができる値であり，実行すると結果として値を得ることができる。関数を実行するためには，適切な引数を助詞とともに与える必要があり，適切でなかった場合，実行にエラーが発生する。

値の演算

クロガネは，一つあるいは二つの値に対して，演算することができる。各演算について説明する。

等価演算

あらゆる二つの値に対して，等価比較演算および論理演算を行うことができる。等価比較演算の結果は，等しい場合には真であり，等しくない場合には偽である。

論理演算

あらゆる二つの値に対して，論理演算を行うことができる。論理演算は，AND 演算，OR 演算のいずれかである。二つの値 A，B に対して，AND 演算を行った場合，A が無あるいは偽であれば演算結果は A である。そうでなければ B である。二つの値 A，B に対して OR 演算を行った場合，A が無あるいは偽であれば演算結果は B である。そうでなければ A である。値が真偽値であるとき，NOT 演算を行うことができる。真を NOT 演算した結果は，偽であり，偽を NOT 演算した結果は真である。

算術演算

二つの値が整数あるいは小数であるとき，二つの値に対して，加算，減算，乗算，除算が行える。ただし，両方の値が，整数である時，除算した値も整数（切り捨て）になる。二つの値がともに整数であるとき，剰余算（割った時の余りを求める演算）が行える。値が整数あるいは小数であるとき，符号反転演算が行える。

大小比較演算

大小比較演算は，二つの値に対して行い，“小さい”，“小さいか等しい”，“大きいか等しい”，“大きい”のいずれかである。結果は真偽値である。大小比較演算は，整数同士，小数同士，整数と小数，文字列同士，に対して行える。文字列の大小比較演算の結果は，処理系に依存するが，等価比較演算と矛盾しない。さらに，“A より B が小さく，B より C が小さいとき，A は B より小さい”といった条件も満たす。

文字列結合演算

結合演算は，あらゆる二つの値に対して行うことができ，結果は文字列である。二つの値を演算したとき，二つの値を文字列に変換し，結合する。例えば，二つの値がそれぞれ“ABC”と“DEF”であった場合，演算結果は“ABCDEF”である。文字列を文字列に変換する場合，元の文字列を得る。無を文字列に変換する場合，長さが 0 である文字の列を得る。それ以外の値を文字列に変換する場合，得る文字列は処理系に依存する。ただし，同じ値を変換する場合は必ず同じ文字列を得ることができる。

変数とスコープ

この章では，クロガネにおける変数の扱いとそれをまとめるスコープについて説明する。

変数

変数は，変数名によって参照でき，値を持つ。ある値 A をある変数名 B によって参照できるようにする処理を代入とし，短く「A を B に代入」と呼ぶ。

スコープ

変数は必ず，いずれかのスコープに属する。スコープは，変数を変数名によって参照できる範囲をさす。さらにスコープは入れ子になることがある。最も外側のものをグローバルスコープとし，グローバルスコープに属する変数をグローバル変数と呼ぶ。内側のスコープは，ローカルスコープと呼び，それに属する変数をローカル変数と呼ぶ。

代入

グローバル変数は再代入可能である。グローバルスコープで A に 3 が代入されていたとする。その後 A に 5 を代入した場合，以降グローバルスコープにおいて変数名 A で参照できる変数の値は 5 になり，前の値 3 を参照することはできなくなる。一方，ローカル変数は再代入不可能である。ローカル変数に対して再び代入を行った場合，エラーが発生する。

関数

クロガネの関数は，一つのブロックと複数の仮引数からなる。ブロックは，文の集合で実行可能である。仮引数は，仮引数名と助詞の対からなる。また，仮引数の助詞は一つの関数の中で重複しない。

関数の実行

1. 引数の確認

関数の呼び出しは，複数の実引数を与えて行う。実引数は，値と助詞の対からなる。まず，呼び出し時に与えられた実引数の助詞と，呼び出される関数の仮引数の助詞を比較する。この集合が一致していなかった場合，関数の呼び出しに失敗しエラーとなる。

2. 実行の前準備

関数の前準備として，スコープを作成する。このスコープの外側のスコープは，関数が呼び出されているスコープではなく，宣言されているスコープである。その後，各仮引数に対し，仮引数名で参照できる変数に対して，その仮引数の助詞と一致する助詞を持つ実引数の値を代入する。これらの変数は先ほど作成したスコープに属する。

3. ブロックの実行

前準備において作成したスコープにおいて，ブロックの実行を行う。ブロックを実行した後，そのブロックの中で最後に実行された文の実行結果を，関数の戻り値とする。ブロックの中で一つの文も実行されなかった場合，関数の戻り値は無とする。関数の戻り値は，関数の呼び出し元において，呼び出しの実行結果として扱われる。

構文

この章ではクロガネの構文を，EBNF で記述する。EBNF で記述できない範囲は定義の下に文章で記述する。なお，仕様の中で構文要素が実行可能であると記述されている場合，それはさらに実行した結果を得ることができるものとする。

プログラム

```
プログラム文章 ::= { 文 }
```

プログラム文章は実行可能である。プログラム文章の実行は，グローバルスコープにおいて文を順に実行する。

文

```
文 ::= 関数定義文 | 分岐文 | 通常文
```

文は，関数定義文，分岐文，通常文のうちのいずれかである。文は実行可能である。

関数定義文

```
関数定義文 ::=
 ‘以下の定義で’ { 仮引数 } 関数名 ‘する’句点 ブロック ‘以上’ 句点
仮引数 ::= 仮引数名 助詞
仮引数名 ::= 変数名
ブロック ::= { 文 }

読点 ::= ‘,’ | ‘，’ | ‘、’
句点 ::= ‘。’ | ピリオド
ピリオド ::= ‘.’ | ‘．’
```

関数定義文は，仮引数名と助詞の対の集合と，一つのブロックを持つ。関数定義文の実行は，まず，仮引数とブロックから関数を作成する。次に，現在のスコープに属しており関数名で参照される変数に，作成した関数を代入する。関数定義文の実行結果は作成した関数である。

条件文

```
条件文 ::= ‘もし’ 条件文内容 { 条件文内容 }
条件文内容::= 条件式 ‘なら’ 通常文
条件式 ::= 式
```

条件文は，一つ以上の条件文内容を持つ。条件文内容は実行可能であるが，結果を持たない場合もある。条件文の実行は，条件文内容を順に実行する。まず，最初の条件文内容を実行する。もしその条件文内容が実行結果を持つなら，その実行結果を条件文の実行結果とし，条件文の実行を打ち切る。条件文内容が結果を持たない場合，次の条件文内容に進む。この実行を実行結果を返せるようになるか，最後の条件文内容を実行するまで続ける。もし最後の条件文内容が実行結果を持たない場合，条件文は無を実行結果として実行を終える。

条件文内容は，条件要素と通常文を持ち，条件要素は実行可能である。条件文内容の実行は，まず条件要素を実行する。条件要素の実行結果が偽あるいは無であった場合，条件文内容は実行結果を持たないまま実行を終える。条件要素の実行内容が偽でも無でもなかった場合，条件文内容は通常文を実行し，その実行結果を条件内容の実行結果として，実行を終了する。

通常文

```
通常文 ::= 終止句 句点
         | 連用句 読点 { 継続連用句 読点 } 継続終止句 句点
```

通常文は一つ以上の句で構成され，通常文の実行は，句を先頭から実行する。句は実行可能である。連用で始まる名前を持つ句には読点が続き，さらに別の句が続く。そのとき，実行結果を続く句に渡す。継続で始まる名前を持つ句は，直前に句が存在する。継続句の実行は直前の実行結果を用いて実行する。継続連用句は継続句と連用句の両方の仕様を満たす。

句

```
終止句 ::= 終止代入句 | 終止定義句 | 終止呼び出し句 | 決定句
連用句 ::= 連用代入句 | 連用定義句 | 連用呼び出し句
継続終止句 ::= 継続終止代入句 | 継続終止定義句 | 継続終止呼び出し句
```

```
継続連用句 ::= 継続連用代入句 | 継続連用定義句 | 継続連用呼び出し句
```

連用句，継続終止句，継続連用句はそれぞれ代入句，定義句，呼び出し句のいずれかである。終止句はさらに三種類の句であるか，決定句である。またいずれの句も実行可能である。継続終止句と継続連用句をまとめて，継続句と呼ぶ。

代入句

```
終止代入句 ::= 式 ‘を’ 変数名 ‘に代入する’ | 変数名 ‘に’ 式 ‘を代入する’
連用代入句 ::= 式 ‘を’ 変数名 ‘に代入し’ | 変数名 ‘に’ 式 ‘を代入し’
継続終止代入句 ::= 変数名 ‘に代入する’
継続連用代入句 ::= 変数名 ‘に代入し’
```

終止代入句，連用代入句，継続終止代入句，継続連用代入句，をまとめて代入句と呼ぶ。代入句の実行は，変数名で参照されるグローバル変数に対して代入を行う。継続句でない代入句は要素を実行し，その結果を代入する。継続句である代入句は直前の句の実行結果を代入する。代入句の実行結果は代入された値である。

定義句

```
終止定義句 ::= 式 ‘を’ 変数名 ‘とする’
連用定義句 ::= 式 ‘を’ 変数名 ‘とし’
継続終止定義句 ::= 変数名 ‘とする’
継続連用定義句 ::= 変数名 ‘とし’
```

終止定義句，連用定義句，継続終止定義句，継続連用定義句，をまとめて定義句と呼ぶ。定義句の実行は，現在のスコープの変数名で参照される変数に対して代入を行う。継続句でない定義句は，式を実行し，その結果を代入する。継続句である定義句は，直前の句の実行結果を代入する。定義句の実行結果は代入された値である。

呼び出し句

```
終止呼び出し句 ::= { 引数要素 助詞 } 関数要素 ‘する’
連用呼び出し句 ::= { 引数要素 助詞 } 関数要素 ‘し’
```

```
継続終止呼び出し句 ::= { 引数要素 助詞 } 関数要素 ‘する’
継続連用呼び出し句 ::= { 引数要素 助詞 } 関数要素 ‘し’
引数要素 ::= 式
関数要素 ::= 式
助詞 ::= ひらがな { ひらがな }
```

終止呼び出し句，連用呼び出し句，継続終止呼び出し句，継続連用呼び出し句をまとめて，呼び出し句と呼ぶ。呼び出し句の実行は，まず関数要素を実行し実行結果を関数とみなす。次に各引数要素を実行する。これらの結果を引数要素結果とする。関数要素を実行した結果が関数でなかった場合，エラーになる。最後に引数要素結果と助詞の対の集合と，継続句であれば直前の句の実行結果を関数に渡し，実行する。呼び出し句の実行結果は関数の戻り値である。

式

```
式 ::= 結合式
結合式 ::= 生成式 | 結合式 結合演算子 生成式
生成式 ::= 真偽式 | 真偽式 生成演算子 生成式
論理式 ::= 等価式 | 真偽式 論理演算子 等価式
等価式 ::= 比較式 | 等価式 等価演算子 等価式
比較式 ::= 加算式 | 比較式 比較演算子 加算式
加算式 ::= 乗算式 | 加算式 加減算演算子 乗算式
乗算式 ::= 単項演算式 | 乗算式 乗除算演算子 単項演算式
単項演算式 ::= 要素 | ( NOT 演算子 | 符号反転演算子 ) 要素

結合演算子 ::= ‘…’
生成演算子 ::= ‘:’ | ‘・’
論理演算子 ::= AND 演算子 | OR 演算子
AND 演算子 ::= ‘&’ | ‘＆’ | ‘∧’
OR 演算子 ::= ‘|’ | ‘｜’ | ‘∨’
等価比較演算子 ::= 等しい演算子 | 等しくない演算子
等しい演算子 ::= ‘=’ | ‘＝’
等しくない演算子 ::= ‘!=’ | ‘≠’
大小比較演算子 ::=
小さい演算子 | 小さいか等しい演算子 | 大きいか等しい演算子 | 大きい演算子
小さい演算子 ::= ‘<’ | ‘＜’
```

```
小さいか等しい演算子 ::= ‘<=’ | ‘≦’
大きいか等しい演算子 ::= ‘>=’ | ‘≧’
大きい演算子 ::= ‘>’ | ‘＞’
加減算演算子 ::= 加算演算子 | 減算演算子
加算演算し ::= ‘+’ | ‘＋’
減算演算子 ::= ‘-’ | ‘－’
乗除算演算子 ::= 乗算演算子 | 除算演算子 | 剰余算演算子
乗算演算子 ::= ‘*’ | ‘＊’ | ‘×’
除算演算子 ::= ‘/’ | ‘／’ | ‘÷’
剰余算演算子 ::= ‘%’ | ‘%’
NOT 演算子 ::= ‘!’ | ‘！’ | ‘¬’
符号反転演算子 ::= ‘-’ | ‘－’
```

式は実行可能である。結合式を実行した場合，左辺と右辺を文字列結合演算する。生成式を実行した場合，左辺と右辺からなる対を得る。論理式を実行した場合，論理演算を行う。この時，論理演算子が AND 演算子であれば AND 演算を行い，OR 演算子であれば OR 演算を行う。論理式以外の式（等価式，比較式，加算式，乗算式，単項演算式）の実行も同様に，演算子により演算が決まる。式が要素である場合，式を実行した結果は，要素の示す値である。

要素

```
要素 = ‘(‘ 式 ‘)’ | リスト | 属性式 | 単純要素
属性式 ::= 単純要素 | 属性式 ‘の’ 属性名
単純要素 ::= 変数名 | リテラル値
変数名 ::= 変数利用可能文字 { 変数利用可能文字 }
```

要素は値を示す。

属性式が“A の B”と表現されている場合，A が対でなければプログラムはエラーになる。A が対であって B が“頭”であれば，属性式の値は，対 A の前者を示す。A が対であって B が“体”であれば，属性式の値は対 A の後者を示す。

変数名の値は，そのスコープに属する変数名で参照できる変数の値を示す。変数利用可能文字は，漢字かカタカナである。処理系により変数利用可能文字を拡張しても良い。

リスト

```
リスト ::= ‘[]’ | ‘[’ 式 { 読点 式 } ‘]’
```

リストが“[]”であるとき，値は無である。リストが“[式]”であるとき，値は，“式：無”とする。リストが“[式１ 読点 式２... 式Ｎ]”であるとき，リストは“式１:[式２... 式Ｎ]”とする。このリストの展開は再帰的に説明されていることに注意すること。

リテラル値

```
リテラル値 ::= ‘無’
             | ‘真’ | ‘偽’
             | 整数リテラル
             | 小数リテラル
             | 文字列リテラル
整数リテラル ::= 数字 { 数字 }
小数リテラル ::= 数字 { 数字 } ピリオド 数字 { 数字 }
文字列リテラル ::= ‘「’ { 文字列用文字 } ‘」’
数字 ::= ‘0’ | ‘1’ | … | ‘9’ | ‘０’ | ‘１’ | … | ‘９’
```

リテラル値で示される値は，“値”の章で示したものである。文字列リテラルで示される文字列は左右のカギ括弧 “「”，“」”の二つを除く，内側の文字の列である。文字列用文字は，閉じカギ括弧“」”を除く任意の文字である。

標準ライブラリ

クロガネの処理系は，次の組み込みの関数や定数を利用できる。

入出力

改行

処理系が動作する環境での改行コードが入っている変数。例えば，Windows 環境では，CR(0x0D)+LF(0x0A)である文字列が入っている。

行入力する

標準入力から文字列を改行が来るまで読み取り，読み取った結果を文字列として返す関数。

値を出力する

仮引数“値”に渡された値を文字列に変換し，標準出力に出力する。実行結果は，文字列に変換された値である。

パスを参照する

パスが文字列であれば，実行環境においてパスで指定されるファイルをクロガネのプログラムファイルとして実行する。“参照する”関数の実行結果は，そのプログラムファイルを実行した際の，最後に実行された文の実行結果である。

その他

テキストを文字分割する

引数のテキストが文字列であれば，実行結果は，長さが 1 である文字列のリストである。例えばテキストが“「ABC」”であれば，実行結果は“[「A」，「B」，「C」]”である。

サンプルプログラム

クロガネで記述された，プログラムのコードを例示する。

サンプル１

“山田さんこんにちは”と表示するプログラム

```
以下の定義で彼に挨拶する。
    彼…「さん，こんにちは」を表示する。
以上。
「山田」に挨拶する。
```

サンプル２

２０番目のフィボナッチ数を表示するプログラム。

```
以下の定義でNをフィボナッチする。
    もし（N≦１）なら，
        Nである。
    他なら，
        (N-1)と(N-2)をそれぞれフィボナッチし，加算する。
以上。
20をフィボナッチし，表示する。
```

サンプル３

リストから重複する要素を取り除き表示するプログラム

```
以下の定義でリストをユニークする。
    もし（リスト＝[]）ならリストである。
    リストの体をユニークし，ユニーク体とする。
    ユニーク体から【□＝リストの頭】を検索し，発見とする。
    もし発見なら，ユニーク体である。
        他なら，リストの頭とユニーク体である。
以上。
```

[3, 1, 4, 1, 5, 9, 2, 6, 5]をユニークし，出力する。