論文

アルゴリズムとプログラム作成の学習を容易にするプログラミング言語の提案

Proposition of A Programming Language for Easy Learning of Algorithms and Programming

今福　啓[*1]
Kei Imafuku

Email: imafuku@dokkyo.ac.jp

本研究では、コンピュータを利用した問題解決手法であるアルゴリズムの学習と、そのプログラム作成を容易にすることを目的とした新たなプログラミング言語を提案する。一般的なプログラミング言語とは異なり、提案する言語ではプログラムを文章と同様に左から右に記述でき、さまざまなアルゴリズムを一行で記述できる。また構文が一定の長さごとに役割を持つため、プログラムを部分的に説明することが容易となる。提案する言語を用いたさまざまなプログラム例を通じて、実際に一行でさまざまなアルゴリズムを実現できることを示す。

In this paper, we propose a new programming language to facilitate the learning of algorithms and the programming of the algorithms. In usual, a programming language requires to describe the program on multiple lines. The proposed programming language has a syntax that allows to write from left to right just like sentences, and can describe various algorithms on a single line. The syntax of the proposed language has a meaning for each fixed length, therefore it is easy to understand partially. Some examples using the proposed programming language shows that various algorithms can be expressed by one line.

[*]1：獨協大学経済学部経営学科

1. はじめに

社会には多くのプログラミング言語が存在し、さまざまな用途に用いられている。その総数は2017年7月の時点で1500種類を超えるとされる(1)。本研究では新たなプログラミング言語を提案するが、これだけ多くのプログラミング言語が存在する中で、新たな言語を作る意味があるだろうか。

理由の一つは、あらゆるプログラムを作成可能な汎用的な用途ではなく、作成したいプログラム内容を容易に実現できるよう、用途を特化した言語が求められているためである。たとえば子供向けのプログラミング教育にはScratch、事務処理にはCOBOL、人工社会のモデリングにはartisocやNetLogo、ディープラーニングにはPython、推論にはPrologやOTTERが用いられる。

本研究では、現在社会で不可欠なコンピュータを用いた問題解決手法であるアルゴリズムの理解と、それをコンピュータプログラムとして作成する方法を同時に効率的に学ぶことを目的としたプログラミング言語を提案する。教育現場では、アルゴリズムの理解とプログラミング言語の理解を切り離して学習する場面がみられる。しかし、アルゴリズムがコンピュータで問題解決する方法である以上、プログラムとして作成し動作させることを身につけることは不可欠といえる。

アルゴリズムの理解を深めるため、実際のプログラミング言語の代わりに疑似言語を用いることもある。疑似言語は説明のための仮想的な言語であり、コンピュータ上で実際に動作するものではない。そのため、疑似言語では実際に動作するプログラム作成を学ぶことはできない。

また、一般のプログラミング言語では処理が上から下の行に向かって進み、各行では左から右に処理される場合もあれば、右から左に処理される場合もある。そのため、1行ごとではなく一定程度まとまった行ごとに内容を理解することが必要であり、プログラムを部分的に読みつつ理解を深めるといったことが難しい。これは、プログラム言語の知識を多く持たない学習者にとって障害になると考えられる。

提案するプログラミング言語は、通常の文章のように左から右にプログラムを作成することが可能であり、さまざまなアルゴリズムを一行で記述できる。また、構文が一定の長さごとに役割をもつ。そのため、語学の学習を文節で区切って行うように、プログラムを途中で区切りつつ理解を進めることが可能となっており、学習効率の向上が期待できる。

以下では、2章で一般のプログラミング言語の特徴について述べる。3章では提案するプログラミング言語の詳細、4章で提案する言語のプログラム例を示し、5章で本研究のまとめを述べる。

2. 一般のプログラミング言語

最初に、プログラミング言語として幅広く使われているPythonを例にして、一般的なプログラミング言語の特徴を述べる。

通常のプログラミング言語では、作成されたプログラムは上から下の行に向かって処理される。また、命令は複数行で一つの意味をもつ。各行の処理は左から右に進む場合もあれば、逆に処理結果を左に受け渡する場合もあり、処理の流れは一つの方向に限定されない。

数値の1から5を合計して結果を表示する内容をPythonで作成すると、以下のようになる。なお、各行の左端の数値は行番号であり、プログラムには含まれない（これ以降のプログラムでも同様である）。

```
1. sum = 0
2. for  x  in  range(1, 6):
3.       sum += x
4. print(sum)
```

上記のプログラムでは、2, 3行目がくり返しを実行する1つの内容となっている。また、2, 4行目は左から右に処理されるが、1, 3行目は = や += の右側の内容を左側に代入する内容であり、右から左に処理が進む。

ここで示したプログラムは、プログラミング言語についての知識があれば容易に理解できる単純なものである。しかし、各行における処理の流れは命令によって異なり、上から下、左から右にプログラムを読み進めれば内容を理解できるわけではない。さらに、プログラミング言語には記述にあたって様々なルールが存在するため[1]、それが身についていなければ簡単な内容であっても読解は難しいものとなる。そのため、アルゴリズムを学習しつつ、プログラミング言語も学習してプログラムを作成することは大変な困難といえる。

3. 提案するプログラミング言語

3.1 言語の概要

本論文で提案するプログラミング言語は、日本語や英語のような通常の文章と同様に、左から右に作成することが可能である。また、複数行で1つの命令を構成する一般のプログラミング言語とは異なり、1行で1つのまとまった処理を記述することが

[1] Pythonであれば複数行にわたる命令の開始時にはコロン : が必要であり、続く行ではインデント（字下げ）が必須になるという独自のルールがある。

可能となっている。

提案する言語のプログラム例は以下ようになる。なお、プログラム例は紙面の横幅に限りがあり複数行になっているが、行を分けず1行で記述できる。以下でもプログラムが複数行となり、左端に行番号をつけた例があるが、すべて1行で記述している。

```
1.  [1:5] |: 0 ≫ sum ； #+sum ≫ sum ； sum :|
2.   ≫ Print
```

これは先に Python の例としてあげた内容と同じく1から5の値を合計して表示するが、通常の言語では複数行を要する内容を、一般的な文章のように左から右に向かって一行で記述できる。提案する言語は、このように一つのアルゴリズムを一行で記述でき、構文が一定の長さごとに役割をもつため、大学における講義などで解説を行う際に、プログラムを途中まで記述しつつ、部分的に説明することが容易となる。なお、提案言語では同じ内容をさらに短く記述することが可能であり、内容を容易に理解できるよう作成できる。

```
[1:5] ≫ Sum ≫ Print
```

また、後に述べるスタックを通じて使用する値を変数や関数への入出力に使用する点、無名関数と同様の使い方で再帰を記述できる点も一般の言語にはみられない特徴である。

文章のようにプログラミング言語を記述することが理解のしやすさにつながることについて、以下の2つの例をもとに説明する（文献 (2) 7章）。

```
if （ length >= 10 ）
if （ 10 <= length ）
```

この例では、ほとんどのプログラマが最初のほうが読みやすいと言うとだろうと指摘している。その理由として、条件部の左側に変化する内容、右側に変化しない内容を記述するという指針が英語の用法と合っており「もし君が18歳以上ならば」と言うのは自然だが「もし18歳が君の年齢以下ならば」と言うのは不自然であると述べている。そのため、最初から文章のようにプログラムを記述できれば、人にとって理解しやすい内容となることが期待できる。

本論文で提案するプログラミング言語は、入力されたプログラムを1行ごとに解析して実行する REPL により、対話的に作成することが可能となっている。実装は Go 言語 (3) version go1.12 windows/amd64 で行っている。入力されたプログラムは、字句解析により入力されたプログラム中の要素を、トークンという最小構成単位として認識する。続いて構文解析により入力内容がどの命令に相当するか認識され、各命令が実行される。

なお、現在はエラーチェックを行っていないため、構文の間違いなどが含まれると Go 言語のランタイムパニック（実行時に発生するエラー）や無限ループが発生することがある。

3.2 トークン

提案する言語で記述したプログラムは、字句解析の際にトークンという単位で認識される。トークンには小文字のアルファベットで始まる変数（ x, ys など）、大文字のアルファベットで始まる関数（ Print, Sum など）、数値（ 5, −0.85 など）、演算子（ + − * / % ）、比較演算子（ = > >= < <= ）、記号（ , . ； |: :| $ # （ ） { } ≫ ）、スタック操作（ ≫ +> − > *> /> %> ）がある。これらを個々に認識した後、構文解析でプログラムのどの命令に相当するか特定される。

トークンは構造体として実装しており、Go 言語の構文にもとづいて type の後に定義する型の名称である Token が、struct の後に定義する型が記述される。

```
type Token struct {
  Type    string
  Literal  string
  Val     interface{}
  Bool    bool
  Slice    []interface{}
}
```

Type では、トークンの種類を文字列で指定する。Literal は変数名または関数名を記憶するために用いる。提案する言語では、Literal が小文字から始まる場合は変数名、大文字で始まる場合は関数名として処理される。

Val は int か float の値を記憶する際に使用する。interface{} 型 [2] は Go 言語に特有のもので、あらゆる型を代入できるが演算に使用することはできない。つまり、interface{} 型として定義した変数 a や b には int 型の 10 や float 型の 6.8 といった値を代入できるが、a + b のような演算はできない。

Bool は比較演算子の結果を保存するために使用される、一般的なブール型である。

Slice は複数の要素を格納できる、リストともよばれるデータ構造である [3]。提案言語ではスライスに任意の数だけ interface{} 型の要素を配置でき、スライスの要素として特定の種類のトークンだけでな

[2] interface{} 型は、最後にある {} まで含めて型の名称となっている。

[3] Go 言語ではリストではなくスライスとよぶことから、本研究で提案する言語でもそのように名付ける。

く、さまざまな種類のトークンを複数置くことが可能である。

3.3 基本データ型

提案する言語で使用される基本データ型は、整数型（int）、実数型（float）、ブール型（bool）である。本論文では文字列型を使ったプログラム例を作成しないため、現在のところ文字列型を実装していない。また、ブール型は比較演算子の結果のみに使用され、変数に代入して使用することはできない[4]。

本言語は動的型付けとなっていることから、記述した値がintかfloatかは構文解析の際に評価される。たとえばトークンが5であればintとなり、3.8であればfloatと解釈される。動的型付けのため、変数の型を宣言する必要はない。

3.4 変数と関数

小文字ではじまるアルファベットは変数、大文字ではじまるアルファベットは関数となる。変数は、小文字で始まるものであれば途中に数値を含むこともできるが、記号（ . , : など）を混在させることはできない。また、プログラム中で値を代入していない変数を使用することはできない。

関数は以下の組み込み関数のみを使用でき、一般のプログラミング言語のように新たな関数は作成できない[5]。各関数の引数は、関数ごとに決められた引数の数をスタック（次節を参照）から自動的にpopして使用する。関数はPrint, Sum, Len, Head, Flattenの5種類を定義している。Flattenはスタックに複数置かれたスライスを1つに結合するが、スタックに置かれているスライスをすべてpopするため、引数は可変となる。

表1 組み込み関数

関数名	引数	動作
Print	1	標準出力に出力（表示）
Sum	1	スライスを合計してpush
Len	1	スライスの長さをpush
Head	1	スライスの先頭要素を取り出し、値としてpush
Flatten	可変	スタックにある複数のスライスをpopし、結合してpush

3.5 スタック操作

提案するプログラミング言語では、データの入出力にスタックを用いる。スタックとは、複数のデータを1次元状に並べて保持できる構造である。スタックに対する操作は、データ1つを取り出すpop、データ1つを追加するpushの2種類に限定される。データをpopすると、最も新しくpushしたデータが取り出される。たとえば数値10, 20, 30の順にpushした後にpopすると、取り出されるのは30となり、スタックには手前に20, 奥に10が残る。さらにpopすると20が取り出され、スタックには10のみが残る。提案言語では、演算結果、変数、関数の間でのデータ入出力はスタックを通じて行われる。

スタックと同様に、データの追加と取り出しのみを実行できるデータ構造にキューがある。キューにデータを追加すると、すでに追加済みのデータの最後に保存される。データを取り出すと、最も古くに格納されたデータが取り出される。

本研究においてキューではなくスタックをデータ入出力に使うのは、プログラムの可読性を考慮するためである。スタックでは、次に取り出すデータは現在プログラムを作成中の箇所の直近にpushされた値となる。しかしキューを用いると、キューの先頭にあるデータがプログラム中のどの箇所で追加されたのか、プログラムを遡って処理を把握する必要となる。このような煩雑な作業を防ぐため、入出力にスタックを利用する。

なお、使用されない値や変数がスタックに残りプログラム実行時に問題となることがあり得る。これは、スタックを空にする命令を実装することで解決できる。

提案する言語では、プログラム中に値や変数を単独で記述すると自動的にpushされる。popには≫を使用する。スタックに対する操作は以下のとおりである。

表2 スタック操作

記号	例	動作
なし	5	スタックに5をpush
≫	5≫x	5をpushした後、popして変数xに代入
+>	5 +>x	5とxを足してxに代入
−>	5 −>x	xから5を引いてxに代入
*>	5 *>x	5とxを掛けてxに代入
/>	5 />x	xを5で割った解をxに代入
%>	5 %>x	xを5で割った余りをxに代入

3.6 算術演算

int, floatは一般的なプログラミング言語と同様の算術演算が可能である。カッコ内（ ）を先に演算するといった処理も実行できる。

[4] 文字列型もブール型も、プログラム中で使用できるように追加する予定である。

[5] 将来的には可能とする予定である。

表 3　算術演算子

記号	機能	例（上段は int 型、下段は float 型、→の次は結果）
+	加算	2 + 3 → 5 3.5 + 4.5 → 8.0
−	減算	6-2 → 4 7.8 − 11.4 → − 3.6
*	乗算	2 * 3 → 6 3.5 * 2 → 7.0
/	除算	7 / 2 → 3 8.4 / 4 → 2.1
%	剰余	7 % 3 → 1 float 型に対する演算はない

一般的なプログラミング言語では計算結果を変数に代入するが、本研究における言語では結果はスタックに自動的に push される。変数名だけを記述した場合も、変数が記憶する値がスタックに push される。スタックから pop する場合には記号 ≫ を使用する。変数に値を代入する場合は、≫ の右側に変数名を記述する。実際のプログラムで説明すると、以下のようになる。

5 * 10 ≫ x　x + 20 ≫ y

上記の例では、最初に 5 * 10 の演算結果 50 がスタックに push される。その値が ≫ によって pop され、変数 x に代入される。そして変数 x と 20 が合計され、その結果である 70 がスタックに push された後に変数 y に代入される。

提案言語では、各命令を記号や改行で区切る必要がない。そのため、上記のプログラムでは 5 * 10 ≫ x と x + 20 ≫ y は別の命令として構文解析が行われる。

Slice に対する操作も同様で

[1 2 3 4 5] ≫ xs ≫ Print

ならば [1 2 3 4 5] がスタックに push され、その後に ≫ で pop され変数 xs に代入される。そして、変数 xs の内容がスタックに自動的に push され、その値が ≫ によって pop されて関数 Print に引数として渡される。

なお、スライスでは 1 ずつ増加する int 型の [1 2 3 4 5] は [1:5] と記述することもできる。スライス内の値を区切る際に、一般の言語のようにカンマ , は不要である。

3.7　比較演算と条件分岐

条件に合致した場合のみ処理を実行する条件分岐の命令は、以下のようになる。

条件 ？ 成立時の処理 .

条件では、比較演算子を用いて変数や値の大小や等しいか比較する。条件が成立した場合には結果がブール型の true としてスタックに push され、成立時の処理が実行される。なお、成立時の処理の最後にはピリオド . が必要である。不成立なら false が push される。条件の記述に使用する比較演算子は以下のとおりである。

表 4　比較演算子

記号	機能	結果が true となる例
=	等しい	5 = 5
>	より大	5 > 2
>=	以上	5 >= 5
<	より小	5 < 7
<=	以下	5 <= 7

条件には複数の比較演算を並べることが可能であり、すべての比較演算が成立した場合に条件成立する AND 演算となる。現在のところ、複数の比較演算のうち、いずれか 1 つでも成立すれば条件成立となる OR 演算は実装していない。将来的にはカンマ , で区切るといった構文による実装を検討している。

3.8　スライス分割と連結

スライスを先頭の値とその他に分割するには記号 ! を使用する。たとえば

[1 2 3] ≫ x ! ys

とすると、変数 x には値 1 が、変数 ys には残りのスライス [2 3] が代入される。

値、スライス、変数を連結して新たなスライスを作成する場合にも記号！を使用する。スタックから pop した値と！の右側の要素を連結し、結果を push する。たとえば

1 ! 2

であれば、スタックに push された値 1 が ! により pop され、値 2 を連結してスライス [1 2] を作成し、それを push する。

連結を使用したいくつかのプログラム例を以下に示す。→ の右側は実行結果である。

1 ! [2 3] → [1 2 3]
[1 2] ! 3 → [1 2 3]
[1 2] ! [3 4] → [1 2 3 4]
[1 2] ≫ x　3 ≫ y　[4 5] ≫ z　x ! y ! z → [1 2 3 4 5]

3.9 スライスくり返し処理

スタックにあるスライスから値を 1 つずつ取り出して処理をくり返すには |: と :| を使用する。この処理をスライスくり返し処理と名付ける。スライスくり返し処理は、あらかじめスタックに push されたスライスを引数として処理する命令で、セミコロン ; で 3 つの部分に区切られる。

```
|: 初期化 ；くり返し処理 ；終了処理 :|
```

初期化は、くり返し処理を開始する前に 1 度だけ実行する内容である。

くり返し処理では、引数となるスライスから値を 1 つ取り出してして処理する。取り出された値は、記号 # で定義される変数に自動的に代入される。くり返し処理は、スライスが空になるまで実行される。そしてスライスが空になると、最後に終了処理を実行してスライスくり返し処理は終了する。

具体例を使って、スライスくり返しの処理を説明する。

```
[1:5]  |:0 ≫ sum ； # + sum ≫ sum ； sum :|
```

最初に [1:5] が int 型のスライス [1 2 3 4 5] と解析され、スタックに push される。スライスくり返し処理は、このスライスが空になるまでくり返される。

0 ≫ sum は初期化である。int 型の 0 をスタックに push した後、それを pop して変数 sum に代入する。

くり返し処理では、最初にスライスから値 1 が取り出され、変数 # に代入される。そして変数 # の値と変数 sum の値が加算され、結果が変数 sum に上書きされる。続いてスライスから 2 が取り出され、同様にして変数 sum に足し合わされる。この手順が、スライスから 5 を取り出して処理されるまでくり返される。

最後に、終了処理で変数 sum の値がスタックに push され、スライスくり返しが終了となる。

提案する言語では、くり返し処理において、スライスから取り出した値を置くための変数名を # に固定している。このように自由な変数名を使用できないようにすることでプログラム記述の自由度は下がるが、記述の方法が限定されるため作りやすさが向上し、作成後の可読性の良さにもつながる。

なお、条件を満たした際にくり返し処理を終了する、一般のプログラミング言語における break のような処理はできない。

3.10 再帰

再帰とは、さまざまなデータに対して同じ処理をくり返す際に用いる手法である。提案言語では、記号 { } を使用する。この処理はセミコロン ； で 4 つの部分に分けられる。

```
{ 終了条件 ；再帰中処理 ；再帰呼び出し ；終了処理 }
```

再帰では、スタックに push された複数の値やスライスを利用できる。スタックから pop した場合、それを代入する変数名として $1 を使用する。さらにスタックにある値を pop して使用する場合は、変数名を $2 , $3 , ? とする。

終了条件では、再帰を終了するための条件を比較演算子を用いて記述する。条件を満たさない場合は、再帰中処理と再帰呼び出しが実行される。なお条件分岐とは異なり、最後にピリオド . は不要である

一般のプログラムで再帰を行うには、定義した関数の中で自分自身を呼び出すよう関数を作成する必要がある。当然のことながら、関数内で呼び出す関数名を指定する必要があるため、無名関数で再帰を行うことはできない。

提案する言語では、再帰呼び出しにおいて { } を記述すれば再帰が可能である。そのため、無名関数のように再帰を使用できる。また、再帰に不可欠な「終了条件」「再帰呼び出し」について、一般のプログラムでは関数内のどの箇所に記述してもよいが、記述する箇所を固定している。このことはプログラム記述の自由度を減少させるが、再帰の動作は理解しやすくなる。

再帰を用いて作成できるプログラムの一つに階乗がある。ここでは 5! = 5 × 4 × 3 × 2 × 1=120 を計算する例をもとに、提案言語の再帰について解説する。

```
1. 5 { $1<=1 ?1 ； $1 − 1 ≫ x ； x { } ；
2. *>$1  $1 } ≫ Print
```
[6]

最初に int 型の値 5 がスタックに push される。そして { から再帰の処理が開始され、スタックにある値が自動的に pop されて変数 $1 に代入される。

$1<=1 ？1 は、再帰の終了条件である。ここでは変数 $1 が 1 以下かどうかを判定し、成立する場合は成立時の処理として値 1 がスタックに push される。その上で終了処理 *>$1 $1 を実行し、再帰を終了する。

再帰を継続する場合は、続いて再帰中処理 $1-1 ≫ x を実行する。ここでは再帰処理の引数となっている変数 $1 から 1 を引いた値が、新たな変数 x に代入される。

x { } では、変数 x をスタックに push し、新たな再帰を開始する。

終了条件が満たされると、終了処理 *>$1 $1 が実行される。ここではスタックにある値が pop さ

[6] このプログラムは以下も同じ内容である。
5 { $1 ≤ 1 ?1 ; ;$1-1 { } ; *>$1 $1 }

れ、後に続く変数 $1 と掛けた値が変数 $1 の新たな値となる。続く $1 では、スタックに変数の値を push する。そして、この再帰を開始した他の再帰が終了していない場合には、そちらの処理に戻る。

すべての再帰が終了すると、スタックにある値が pop されて関数 Print に引数として渡され、標準出力に出力（画面に表示）されてプログラムは終了となる。

なお、新たな再帰が開始されるたびに、その中で使用する引数や変数を置くための変数が新たに作成される。上記のプログラムであれば、最初に再帰の引数として 5 が変数 $1 に代入され、$1 − 1 の結果である 4 が代入された変数 x を引数として新たな再帰が開始される。その際、新たな再帰で 4 が代入された変数 $1 は、呼び出し元の再帰で使用される、5 が代入された変数 $1 とは別のものとして作成される。

3.11 スライスくり返し処理と再帰について

スライスくり返し処理は、再帰でも表現できる。3.1 節で示した [1:5] を合計するプログラム [7]

```
[1:5] |: 0 ≫ sum ； #+>sum ； sum :|
```

は、スライスくり返し処理の代わりに再帰を用いると以下のようになる。

```
1. [1:5] ｛$1 ≫ Len <=1 ? $1 ≫ Head ；
2. $1 ≫ x !ys ； ys ｛｝ ； +>x x ｝
```

上記のプログラムでは、再帰中処理で受け取った引数 $1 を値 x とそれ以外のスライス ys に分割する。そして ys に対して同様の処理を再帰呼び出しでくり返すが、終了条件 $1 ≫ Len <= 1 を満たした際には $1 ≫ Head を実行して再帰を終了する。

終了条件は、最終的に処理するスライス ys が [5] だけとなり、引数として再帰呼び出しされた際に満たされる。$1 ≫ Head ではスライス [5] から先頭の値 5 を取り出して push する。

再帰の終了処理では、+>x でスタックにある値を pop して x と加算した値を新たな x とし、それを再帰の結果として push して終了する。

このように、再帰でスライスくり返し処理を表現できることから、構文が重複するスライスくり返し処理は不要にみえる。しかし、再帰での記述は複雑になることや、次章で述べるように再帰の内部でスライスくり返し処理を用いることがあるため、スライスくり返し処理は不可欠な命令となる。

[7] 3.1 節の内容とはスライスの記述を変更している。

4. プログラム例

提案するプログラミング言語によるプログラム例を示し、処理の流れを説明する。

4.1 スライスの反転

スライスの順番を逆にするプログラムは次のようになる。実行結果は、スライス [5 4 3 2 1] となる。

```
1. [1 :5] ｛$1 ≫ Len <= 1 ? $1 ；
2. $1 ≫ p !rs rs ；｛｝ ；≫ x x !p ｝
```

$1 ≫ Len?1 ? $1 では、再帰に引数として渡されるスライスの長さが 1 以下となると引数をそのまま返す。そうでなければ変数 $1 を変数 p（値）と変数 rs（スライス）に分割し、変数 rs を引数として新たな再帰を開始する。

終了処理 ≫ x x !p では、再帰の戻り値を pop して変数 x に代入し、その右側に変数 p を連結して push する。

全体の処理としては、元のスライスの値を左端から取り出し、それを右端に連結する処理がくり返され、元のスライスを反転したものが出力される。

4.2 探索

スライス内に指定するデータが含まれるかどうかを探すアルゴリズムが探索である。データの先頭から順に探す線形探索のプログラムは以下のようになる。

```
1. 7 [1:10] |: ≫ p − 1 ≫ r ； # = p ? # ≫ r. ；
2. r :|
```

最初に探したい値 7、続いて探す値を含む int 型のスライス [1:10] をスタックに push した後に、スライスくり返し処理を開始する。

スライスくり返し処理 ≫ p − 1 ≫ r では、≫ p で探したい値をスタックから pop して変数 p に代入する。また、探した値が存在するかどうかを保存する変数 r を作成し、初期値を -1 とする。

くり返し処理 #=p ? # ≫ r. では、スライスから値を 1 つずつ取り出して変数 # に代入し、その値が探したい値を保存する変数 p に等しいか比較する。値が見つかると条件が成立し、その値を変数 r に代入する。そして終了処理 r では変数 r をスタックに push する。

スライスくり返し処理が終了すると、スタックに置かれた変数 r の値が pop される。

4.3 可変引数の関数

関数 Flatten は、スタックにある複数のスライスを自動的に pop して 1 つのスライスに結合する。

以下のプログラムでは、→ の右側に実行結果を

示す。1 つ目の例は Flatten の結果、2 つ目の例は Flatten の結果を関数 Sum で合計した結果となる。

```
[1 2] [3 4 5] ≫ Flatten → [1 2 3 4 5]
[1 2] [3 4 5] ≫ Flatten ≫ Sum → 15
```

4.4 クイックソート

クイックソートは、並べ替えを行う値の一つを選択し（ピボットという）、残りの値をピボットよりも大きい値のスライスと小さい値のスライスに分割した後にその間にピボットを配置することを、再帰を用いて 2 つに分割した各スライスでも同様に処理する。実行結果はスライス [1 2 3 4 5] となる。

```
1. [3 1 5 2 4] ¦ $1 ≫ Len <= 1 ? $1 ; $1 ≫ p !rs
2. rs ¦: [ ] ≫ ls [ ] ≫ gs ; p>= # ? ls !# ≫ ls .
3. p<# ? gs !# ≫ gs .; ls gs :¦ ;
4. ¦ ¦ ≫ gs ¦ ¦ ≫ ls ; ls !p !gs ¦
```

最初に、並べ替えをおこなうスライス [3 1 5 2 4] をスタックに push し、それを引数として再帰処理を開始する。

終了条件の $1 ≫ Len <= 1 ? $1 では、引数となるスライスの要素数が 1 以下となると、引数 $1 をそのまま push する。

再帰中処理では、$1 ≫ p !rs で引数のスライスをピボット p（スライスの左端の値）とそれ以外 rs に分け、rs をスタックに push してスライスくり返し処理を実行する。

スライスくり返し処理の初期化 [ ] ≫ ls [ ] ≫ gs では、p 以下の値を格納するスライス ls と、p より大きい値を格納するスライス gs を用意する。

くり返し処理 p >= # ? ls !# ≫ ls . p<# ? gs !# ≫ gs . では、スライスから取り出した値 # と p の大小を比較し、結果に応じて # を ls または gs に振り分ける。

終了処理 ls, gs では、続く再帰呼び出しの引数に使用する ls と gs をスタックに push する。

再帰呼び出しでは、先に push された gs をスタックから pop し、それを引数として 1 つ目の再帰呼び出しを行う。この処理により、p よりも大きい値を並べ替えた結果がスタックに push され、それを ≫ gs で gs に代入する。同様にして、p 以下の値の並べ替えを再帰呼び出しで行い、その結果を ls に代入する。

終了処理の ls !p !gs では ls , p , gs を 1 つに結合し、再帰の結果としてスタックに push して終了する。

5. おわりに

本研究では、コンピュータを利用した問題解決手法であるアルゴリズムの学習と、さまざまなアルゴリズムのプログラム作成を同時に学ぶことを容易にすることを目的とした新たなプログラミング言語を開発した。

一般的なプログラミング言語では、プログラムを複数行にわけて記述する必要がある。それに対して提案するプログラミング言語は、プログラムを日本語の文章と同様に左から右に記述できる構文となっており、さまざまなアルゴリズムを一行で記述できる。また、構文が一定の長さごとに役割をもつため、大学における講義などで解説を行う際に、プログラムを途中まで記述しつつ、部分的に説明することが容易となっている。そして提案する言語を用いたプログラム例を示し、実際に一行でさまざまなアルゴリズムを実現できることを示した。

提案するプログラミング言語はプロトタイプであり、現在のところ基本データ型が限定されている。また、組み込み関数は使用できるが、関数を自作できない。文法のエラーチェックも行っていない。

これらの問題を解決することは今後の課題だが、さらにグラフ構造のような複雑なデータ構造を取り入れること、関数を自作するための文法を用意することが必要である。そして、提案するプログラミング言語を用いることで学習が有効に進むことについて、結果を示すことが必要と考える。

なお、提案するプログラミング言語の構文は、本論文で提案した内容で確定するわけではない。たとえば Python でも Version 2 から Version 3 になるにあたり、構文に変更が加えられている。提案言語でも同様にプログラミングの効率性を考慮し、文法のあいまいさを回避するため、今後構文に変更を加える可能性がある。

謝辞

本研究の一部は、獨協大学 情報科学研究所の研究助成によるものです。

参考文献

(1) 増井敏克、“プログラミング言語図鑑”、ソシム (2017)
(2) Dustin, Boswell, Trevor, Foucher（訳：角征典）“リーダブルコード”、オライリージャパン (2012)
(3) https://golang.org
(4) 松尾愛賀、“スターティング Go 言語”、翔泳社 (2016)
(5) Thorsten Ball（訳：設樂洋爾）、“Go 言語でつくるインタプリタ”、オライリージャパン (2018)
(6) 日向俊二、“やさしいコンパイラの作り方入門”、カットシステム (2009)
(7) 滝本宗宏、“実践コンパイラ構成法”、コロナ社 (2017)