プログラミング言語としての日本語

岡田 健[†1] 中鉢欣秀[†2] 鈴木 弘[†3] 大岩 元[†4]

†1慶應義塾大学政策・メディア研究科 †2慶應義塾大学SFC研究所
†3東京都立航空工業高等専門学校 †4慶應義塾大学環境情報学部

概要

正しく表記された日本語をプログラミング言語として用いることについて、語順と情報処理の効率の観点から議論する。日本語をプログラミング言語として用いると、文法教育を必要としないために、日本人に対するプログラミング入門教育ではプログラムの表現能力の育成に集中できる。Java の実行環境である仮想計算機のアセンブラは日本語に容易に変換可能である。このことを用いて、アセンブラから高級言語までを日本語を用いてシームレスに表現できるプログラミング言語「言霊」を開発した。この言語は、細かい処理を記述できるため、熟練プログラマのための言語としても、有効性が期待できる。

Japanese as a Programming Language

Ken Okada[†1] Yoshihide Chubachi[†2]
Hiroshi Suzuki[†3] Hajime Ohiwa[†4]

†1 Graduate School of Media and Governance, Keio University
†2 Keio Research Institute at SFC
†3 Tokyo Metropolitan College of Aeronautical Engineering
†4 Department of Environmental Information, Keio University

Summary

Using Japanese as a programming language is discussed from the viewpoint of word order and efficiency of processing. By using Japanese, programming may be taught without taking into account of the grammar of the language and the learner can concentrate on composing their idea to be realized. The assembler of Java virtual machine is compatible with the Japanese language and a programming language named "Kotodama" has been developed. It supports both assembler level description and higher-level description of the programs and expected to be useful not only for the beginners but also for the experienced programmers as well.

１．はじめに

プログラミング教育は、パソコンの導入とともにBASIC言語でプログラムを書く必要性から一般に広まったが、ソフトウェアの充実とともにその必要性は薄れていった。だがコンピュータとは何かを理解し、自分の考えたものをコンピュータ上で実現できるかどうかを判断できるようになるという教養教育の立場からは、プログラミング教育の必要性は依然として高い。

現在広く行われているプログラミング教育は実用教育に端を発しているものが多い為、教養教育の観点からは問題があるものが多い。実用教育ではコンピュータを使えるようになることが目的となる為、文法教育と定型パターンの利用だけが重視されてしまう。その結果パソコンやプログラミング言語が使えるようになった、という満足だけで終わってしまう。これでは定型パターンでは表せない自分のアイディアを表現する能力が育成されない。

既存のプログラミング言語はヨーロッパ言語を使う人間が設計していて、印欧語の影響を強く受けている。このため西欧人は一般的に印欧語を母語としない日本人より有利にプログラミング言語を学習できるものと予測される。しかしプログラミングで使われる用語が一般用語より限定された意味で使われるために、印欧語を母語とする初心者はプログラミング言語としての意味ではなく一般用語としての意味をとるものと考えて、誤ったプログラムを作ってしまう場合もある。従って単純に西欧人が有利というわけではない。

しかし表現能力を育成する立場からは、覚えなければならない知識はできるだけ少ないほうがよい。この点から、仮に自然言語で記述したものがそのままコンピュータ上で実行できるならば、プログラミング入門教育における文法教育を無くすことができる。そうした試みとして、水谷の「朱唇」がある[1]。しかし「朱唇」はマニュアルが文語で書かれ、プログラムの表記にカナ文字を使わなければならないなど、個性的すぎて一般に普及するには至っていない。初心者教育には、Logoを日本語化したもの[2]も

用いられているが、そもそもプログラミングを一般人に教える必要がないと考える人が多く、普及が進んでいない。

プログラムとして実行される日本語はコンピュータ上で意味が定義された用語に対してだけであるので、全ての日本語がプログラムになるわけではない。欧米の学習者は既存のプログラミング言語を用いたときに一般用語としての意味でプログラムを書いてしまう危険性がある。同じことが日本語プログラミング言語でも起こるかもしれないが、これは教育の過程における工夫で解決できるであろう。

日本語は日本人がソフトウェアを設計する際に最も重要な表現手段である。筆者達の行うプログラミング教育においては、日本語でプログラムを設計させることが、自分のしたいことをソフトウェアとして実現できるようにする教育の中核をなしている。このような考えから、開発環境で日本語を用いることがどんな影響があるかが、東京農工大で研究され[3]、論理型プログラミング言語の提案も行われている[4]。また、Cobol を基礎に分かち書きのない日本語でデータ処理を行う言語も提案されている[5]。

以下、第２章では日本語によるプログラムの設計について、第３章では情報処理に適した語順について、第４章では言語処理系の例として数式の翻訳について述べ、第５章では FORTH と教育用日本語プログラミング言語として最も広く知られた Mind について述べる。第６章では日本語プログラミング言語「言霊」の概要を、Java 言語と関連して述べる。第７章では具象文法設定機能について述べてこれにより「言霊」が自然な日本語で記述できることを述べる。第８章では「言霊」のプログラム記述を Mind や最近開発されたオブジェクト指向の教育言語「ドリトル」、また自然な日本語表現が可能な AppleScript と対比して示す。第９章では「言霊」が従来の言語処理系とは異なる手法で解析を行っていることを述べる。第１０章では「言霊」における低級表現と高級表現の混ぜ書きについて述べ、第１１章では「言霊」の今後の課題について述べる。

２．日本語を用いたプログラムの設計

次章で述べるように、日本語はプログラムを書くのに適した構造を持っている。従って日本語を使う日本人はプログラムを書きやすい環境にある可能性がある。

筆者らが行っているプログラミング入門教育では日本語で書くコメントを重視している。受講者に対して、実際にプログラムを書く前に日本語コメントだけを記述させてみるということを行う。例えばタートルグラフィックスを用いて家を書くプログラムを日本語コメントで表現すると、図１のようになる。日本語コメントが正確に記述できていれば、プログラムの構造を適切に設計できているということになる。

```
家を書くというのは、三角形を書いてから四角形を書く。
三角形を書くというのは、
  ・ 右に３０度曲がって角度を調整する
  ・ １００歩進み右に１２０度曲がる、ということを３回繰り返す
  ・ 左に３０度曲がって角度を調整する
四角形を書くというのは、
  ・１００歩進み右に９０度曲がる、ということを４回繰り返す
```

図 1　日本語コメント

この日本語コメントの妥当性はプログラミング言語へ翻訳して実行してみることで確認するのだが、この翻訳作業が受講者にとっては大きな障壁になる。 例えばこのコメントを **Java** 言語に翻訳すると図２のようになるのだが、これを記述する為には **Java** 言語の文法を理解する必要がある。現実の受講者は文法事項の理解が進まないために、この翻訳作業がスムーズに行えない。日本語でプログラムの論理的な構造を表現することはできるのに、それを現実のプログラミング言語として記述することができないのだ。

本論文が目的としているのは、日本語コメントがそのままプログラムとして記述できるようなプログラミング言語を作ることである。図３は「言霊」を用いて日本語コメントに近い形で表現したプログラムのコードである。こうしたプログラミング言語を作ることにより、日本語からプログラ

ミング言語への翻訳という無駄な作業を取り除くことで文法教育の必要性を無くし、プログラミング教育の効果をあげることができる。

```
public class MyTurtle extends Turtle{
  public static void main(String args[]){
    house(100);                              //大きさが100の家を描く
  }
  public static void house(int size){        //家を描くプログラム
    triangle(size);                          //三角形（屋根）を描く
    square(size);                            //四角形（本体）を描く
  }
  public static void triangle(int size){     // 長さがsizeの三角形を描く
    rt(30);
    for( int i=0 ; i<3 ; i++ ){ fd(size); rt(120); }
    lt(30);
  }
  public static void square(int size){       // 長さがsizeの四角形を描く
    for(int i=0 ; i<4 ; i++ ){
      rt(90); fd(size);
    }
  }
}
```

図　2　家を描く Java プログラム

```
メインとは {
  大きさが１００の家を描く。
} ことである。

大きさが「A（整数型)」の家を描くとは {
  長さがAの三角形を描く。//屋根を描く
  長さがAの四角形を描く。//本体を描く
} ことである。

長さが「A（整数型)」の三角形を描くとは {
  右に３０度曲がる。
   {A歩進む。右に１２０度曲がる} を３回繰り返す。
  左に３０度曲がる。
} ことである。

長さが「A（整数型)」の四角形を描くとは {
   {右に９０度曲がる。A歩進む。} を４回繰り返す。
} ことである。
```

図　3　家を描く日本語プログラム

3.「どうする、何を」から「何を、どうする」へ

最近のパソコンは、マウス操作でコマンドを制御する。従来のキーボードからコマンドを入力する場合とは、語順が変わった点が注目される。コマンド入力は印欧語の語順であったのに対し、マウス入力は日本語の語順になっている。

コマンド入力を行う場合は「どうする、なにを」という印欧語の語順になっている。例えば **UNIX** コマンドを用いてファイルの削除を行う場合は「**rm a.txt**」と入力する。最初に動詞が入力され、次に対象となる目的語が続く。

マウス入力を行う場合は「何を、どうする」という日本語の語順になっている。**Windows** の **GUI** においてマウス操作でファイルを削除する場合は、対象となるファイルを右クリックし、続いてメニューの中から「削除」を選択する。目的語が最初に来て、動詞が最後に来るのである。

操作体系がキーボードからマウスに変わるとともに語順も変化したのは、「何を、どうする」という語順のほうが情報を処理するのに便利だからである。

例えば世界で最初に商品化されたヒューレット・パッカード（ＨＰ）社の電卓は、数式どおりに入力するのではなく、日本語の語順と同じように演算される数を入力してから演算子を入力する方式である。　日本語では「**A** と **B** を足す」というが、「**A enter B +**」と入力するのがＨＰ社の電卓の計算方法である。ここで **enter** キーは **A,B** ２つの数値の間を区切る為に必要となる。

ＨＰ社の電卓では、数値をスタックと呼ばれる記憶機構に入力する。スタックには入力された数値が次々と積み上げられ、演算が指令されると積み上げられた数値のうち、上の２つの数値に対して演算が実行され、結果がこの２つの数値の代わりにスタックの上に積み上げられる。この方式は次章で述べるように括弧を用いる必要が無く、複雑な情報処理に適しているために、米国人にも愛用され現在に至るまで販売が続いている。

このように「何を、どうする」という語順は情報処理に適している。そ

して日本語がこのような語順であるという事もまた情報処理に適している証拠の一つといえる。日本人が暗算を得意とするのも、記数体系の合理性とあいまって日本語が情報処理に適した語順を持っているからであろう。

４．数式の翻訳

プログラミング言語の歴史の最初に現れるのは、アセンブラと呼ばれる言語である。「**ADD A B**」のようにニモニックと呼ばれる符号で表現された命令列を、コンピュータが実行できる２進符号の命令列に変換する。アセンブラにおけるプログラミングとは、電卓のキーをどの順番で叩くと目的の計算が達成されるかを書き出す作業といってもよい。

アセンブラによるプログラムは、それを実際に実行する状況全体を想像しないと何をやろうとしているのか分からないコードになってしまう。そこで処理の塊に対してコメントをつけて、何を実行するかが分かるようにする。

初期のプログラムは数値計算を行うものが大部分であったために、数式を書き出すことが人間にとって分かりやすい表現になる。数式を書けばコンピュータが必要な命令列を生成してくれるプログラムができないかと考えたＩＢＭの **John Backus** は、これに成功した。**FORTRAN** の誕生である。今日では数式をプログラムに書くことが当たり前になっているが、当時はこれが実用レベルで可能になるとは考えられなかった。**Backus** の仕事はコンピュータ使用のブレークスルーを成す画期的な仕事であった。

コンピュータは数式全体を見渡すことはできず、ひとつずつ情報を読み取りながら逐次に処理を進めていく。このときに開き括弧が現れると対応する閉じ括弧が現れるまで処理を確定することができない。このように大域的な性質を持つ括弧を扱うことは、コンピュータにとって困難な仕事なのである。しかしこれが可能なことが分かりそのメカニズムが整理されると、現在ではプログラミング入門を済ませた学生が次に学ぶ「データ構造とアルゴリズム」の授業の中で、例題として数式の翻訳が取り上げられるようになった。

数式を機械語に翻訳する為には、まず中置形式の数式をスタックを用いて後置形式に変換する。ここで後置形式とは「A B +」のように、被演算子の後に演算子を続ける表記法で、逆ポーランド記法とも呼ばれる。

数式演算の実行は、この後置形式で表現された数式に対してスタックを用いることで実現する。これは前述のＨＰ社の電卓を用いて計算するのと同じことである。実際に CPU がＨＰ社の電卓と同じくスタック機構を用いる方式を採用したスタック計算機は、バローズ社の B-5000 など過去にもいくつか存在した。 現在では Java 言語を実行するための仮想計算機（Virtual Machine, VM）がスタックを用いている。

５．FORTH と Mind

FORTRAN は、読みやすさのために中置記法で記述し、実行時にこれを後置記法に変換する仕組みである。これに対して初めから後置形式でプログラムを記述する言語には APL や FORTH がある。APL や FORTH では「１＋２」の演算を行う際には「１　２　＋」と後置記法で記述する。

FORTH が後置記法で記述できるのは、スタック操作が基本となっているからである。FORTH 処理系は入力されたものがデータであればスタックに積み、演算を実行する命令であればスタックに積まれたデータを用いて演算を実行する仕組みである。例えば「１　２　＋」という文字列が入力されると、「１」と「２」はデータなのでスタックに積まれ、「＋」は命令なので演算を実行する。

スタック操作に必要なのは、入力されたものがデータか命令であるかを区別することだけである。その為に FORTH 処理系は命令情報を保存する辞書を定義し、辞書に存在するなら命令であり、そうでないならデータとみなす。この単純な仕組みにより FORTH は高速で実用的な言語である。

FORTH は「どうする、何を」の語順ではなく、「何を、どうする」の語順を実現したプログラミング言語である。つまり FORTH は日本語の語順を満たしており、その語順であるがゆえに実用的な処理系を実現できた。そこで FORTH を日本語化して、日本語で記述できて実用的なプログラミング言

語を実現しようとしたのが片桐明の開発した Mind である[6]。Mind は「１＋２」の演算を行う際には「１と　２を　足す」と後置記法で記述する。

Mind は教育用言語として成果をあげている。 Mind は情報教育用言語として研究され[7]、BASIC と Mind を比較した授業が行われてた結果、日本語である為にプログラムが読みやすく、またプログラムに親しみがもてた為に学習意欲の持続にも効果がある事が報告されている[8][9]。Mind が教育用言語として成功している理由の一つは、語順と単語が日本語に則していて学習者は他の言語で学習したときと比べて違和感を覚えることが少なかったことである。

６．Java と「言霊」

現在広く利用されている Java 言語は、実行する際の仮想計算機（Java Virtual Machine, JavaVM）がスタックを計算機構として用いている。Java プログラムを記述すると、コンパイラは JavaVM が解釈できる機械語に変換する。その機械語を JavaVM が解釈して演算することで、Java プログラムは実行される。例えば図４のような Java 言語のコードがあったとする。Java コンパイラはこれを機械語に変換するが、その機械語をニモニックで表現したのが図５である。実行時には図５のニモニックで表された機械語命令に従ってスタック演算が行われる。

```
if( x==0 ) y=1;
else y=2;
```

図　４　Java 言語の記述

```
  iload_1
  ifne Label_0
  iconst_1
  istore_2
  goto Label_1
 Label_0:
  iconst_2
  istore_2
 Label_1:
```

図　５　ニモニックによる記述

我々が開発している「言霊」は、ニモニックを日本語化し、かつその表現を拡張する方向で設計された。例えば図５を「言霊」で表現すると、図６のような日本語表現になる。だが図６のような低級なスタック操作の命令があると表現が複雑になるので、図７のような高級な表現もできる。

「言霊」は **FORTH** や **Mind** と同じく、後置記法の語順を持ってスタック操作を行う言語である。そのためにこれらが持つ利点をそのまま受け継いでいる。特に「言霊」を用いてプログラミング教育を行う際にこの利点が強く生かされることが期待できる。つまり自然な日本語の語順や表現をしている為に学習者の抵抗が少なくなることが期待できる。

```
  ローカル変数１番目から整数を積む。
  ０以外ならば、ラベル０にジャンプする。
  整数１を積む。
  ローカル変数２番目に整数を格納する。
  ラベル１にジャンプする。
ラベル０:
  整数２を積む。
  ローカル変数２番目に整数を格納する。
ラベル１:
```

図　６　「言霊」による低級な表現

```
x が０ならば、１を y に代入する。
そうでなければ、２を y に代入する。
```

図　７　「言霊」による高級な表現

７.「言霊」の具象文法設定機能

「言霊」は自然な日本語表現を用いていると先に述べたが、どのような表現が自然な日本語表現なのかが問題になるだろう。この問題を解決する為に「言霊」では表現を自由に変更できるような機能がある。これを具象文法設定機能と呼ぶが、これによりユーザは各自が考える日本語表現を使ってプログラムを記述することができる。

例えば「**a=1**」という代入文を日本語で表現するとしたら、どのように表現するべきだろうか。「変数 **a** に１を代入する」「**a** に１を代入する」「**a** に１を入れる」「１を **a** とする」など、様々な日本語表現がありうる。

既存のプログラミング言語では、代入文とはこう記述するのだ、と定義して文法を作成してコンパイラを作成していた。だが我々は代入文の記述方法の定義はユーザに任せている。例えば図８のように設定ファイルの中で代入文の表現を定義する。「言霊」のコンパイラはこの設定ファイルを読み、こうした表現の代入文を受理するような処理機構を用意してからコンパイルを行う。その結果コードの中で「**a** に１を代入する」と記述すると、その表現は代入文として解釈される。

```
<代入先>に<代入値>を代入する
```

図 ８　代入文の表現の定義

この機能は **Meyer** が主張した抽象文法と具象文法の考え方に基づいている**[10]**。プログラムの文法は意味の文法と表現の文法に分離することが可能であり、それぞれを抽象文法と具象文法と呼んでいる。

例えば代入文の文法を **BNF** 記法（**Backus-Naur Form**）で表現すると図９のようになる。ところがこの記法には「代入文は変数と値を持つ」という代入文の構造と、「代入文は“ ～=～ ”と表現する」という代入文の外見が混ざっている。

```
<代入文> ::= <代入先>=<代入値>;
```

図 ９　ＢＮＦ記法を用いて定義した代入文

Meyer はプログラミング言語の外見ではなく構造を表す為に抽象文法を提案した。抽象文法を用いて代入文を定義すると、図１０のようになる。この定義の中で、代入文は代入先と代入値を持っており、それぞれの型が変数と値であることを示している。

```
代入文 ⏏ 代入先：変数　；　代入値：値
```

図 １０　抽象文法を用いて定義した代入文

プログラムの外見は具象文法により定義する。代入文の具象文法は図８に示したように定義する。その際に注意することは、具象文法は抽象文法

に拘束されているということだ。抽象文法により代入文は代入先と代入値のみを持つ概念であることが定義されているので、具象文法でそれ以外の要素を追加して定義することは許されない。具象文法で定義できるのは純粋にプログラムの外見のみである。

「言霊」は具象文法をユーザが設定することができる。これによりユーザは各自が理想とする文法表現を用いてプログラムを記述することができる。プログラムを始めたばかりの初心者であるなら、少々記述が長くても読んで理解しやすいような文法表現が適しているのだろうからそのように具象文法を設定すればいい。中級者はある程度略記した方が可読性・記述性の観点からよいだろうから、そのように具象文法を設定することができる。

８.「言霊」と他の日本語プログラミング言語との記述の比較

「言霊」が自然な日本語表記をしていることは、他の日本語プログラミング言語と比較することで分かる。

図１１は学校教育を目的として開発されたドリトル**[11]**であり、この例はタートルグラフィックスを使って四角形を描くプログラムである。全く同じプログラムを「言霊」で記述すると図１２のようになる。なおドリトルはオブジェクト指向言語なので、図１２の「言霊」の表記もオブジェクトを生成するプログラムになっている。「！」のような記号が無い為に「言霊」は自然な表現になっていることが分かる。

```
カメ太＝タートル！作る。
「カメ太！１００　歩く　９０　右回り。」！４かい　くりかえす。
```

図　１１　ドリトルによる四角形を描くタートルプログラム

```
カメ太（タートル型）を生成する。
｛カメ太が１００歩進む。カメ太が右に９０度曲がる。｝を４回繰り返す。
```

図　１２　「言霊」による四角形を描くタートルプログラム

また図１３は **Mind** のタートルグラフィックスを使って同様に四角形を

描くプログラムであり、図１４はそれを「言霊」で記述したものである。Mind は言葉と言葉の間に空白を入れる分かち書きが必要な言語である。例えば「９０度　右を向き」というコードは「９０度　右を　向き」や「９０度右を　向き」などのように空白を入れる場所を変えると文章の意味が変わってしまう。「言霊」では「右に９０度曲がる」と分かち書きをせずに記述することが可能である。このように「言霊」ではドリトルや Mind と異なり、自然な日本語表記を重視している。

```
４を　回数指定し
        １００歩　前進し
        ９０度　右を向き
繰り返す
```

図　１３　Mindによる四角形を描くタートルプログラム

```
｛
    １００歩進む。
    右に９０度曲がる。
｝を４回繰り返す。
```

図　１４　「言霊」による四角形を描くタートルプログラム

Macintosh で動作するスクリプト言語 AppleScript の古いバージョンでは、自然な日本語表記が可能だった。残念ながら MacOS8.5 以降はサポートされなくなってしまったので現在は使用できないのだが、図１５に見られるような自然な表記が可能だった。

```
アプリケーション“Finder”について
  アクティベート
  デスクトップのフォルダ“サンプル”を開く
  フォルダ“サンプル”のウィンドウの位置を｛０，５０｝にする
  10 回
    位置変数をフォルダ“サンプル”のウィンドウの位置にする
    横位置を位置変数の項目１にする
    横位置を横位置＋１０にする
    位置変数の項目１を横位置にする
    フォルダ“サンプル”のウィンドウの位置を位置変数にする
  以上
以上
```

図　１５　AppleScriptによるウィンドウを右に移動するプログラム

AppleScript の問題点の一つは、コンパイラによる解析上の都合から妙な文法規則を設定していることである。例えば AppleScript における代入文は以下のように定義されている。

```
<変数>を<代入値／文字列>にする
```

図 １６　AppleScript の代入文の定義

変数にはひらがな・カタカナ・漢字・アルファベットが使用できる。変数名がこれらの文字種のうち、単一のものを用いてかつ区切り文字になっている「を」と異なる文字種であれば、図１７のように問題なく記述できる。

```
A を 123 にする
文字変数を“あ”にする
```

図 １７　代入文が自然に表現できる例

ところが区切り文字の「を」と同じ文字種であるひらがなを用いて変数名を定義すると、図１８のように空白による分かち書きが必要になってしまう。

```
ことば　を“あいう”にする
へんすう　を 123＋456 にする
```

図 １８　空白による分かち書きが必要な代入文の例

また変数名に複数の文字種を用いている場合は、図１９のように括弧で変数名を括る必要がある。ここでは「変数１」は漢字と数字、「ＡとＢ」はアルファベットとひらがなという異なる文字種を用いている。

```
「変数１」を“ん”にする
「ＡとＢ」を 100 にする
```

図 １９　括弧で括る必要がある代入文の例

「言霊」にはこのようなコンパイラの都合による文法事項の制限は少ない。変数名にはほとんどの文字や記号が許される。また「言霊」には予約語も存在しないので、ユーザは自由に表現を行うことができる。

また AppleScript の日本語表現に対する批判も存在するだろう。筆者も

AppleScript の代入文が「～にする」という表現になっているのはいささか違和感を覚えた。筆者だったら「～に代入する」という表現を使ったほうが自然になると考える。

これまでの日本語プログラミング言語は、このように各個人の「自然な日本語表現」の感じ方が異なるという問題がある。だが「言霊」は具象文法を設定することができるので、ユーザがそれぞれ自然と思う表現を用いることが可能になっている。

９.「言霊」における解析

「言霊」におけるコードの解析は、従来のプログラミング言語の解析手法とは異なる点がある。従来では決定性有限オートマトンの考えに基づき字句解析を行い、プッシュダウンオートマトンの考えに基づき構文解析を行っていた。だが「言霊」は字句解析・構文解析とを分けることをせず、非決定性プッシュダウンオートマトンの考えに基づいて解析を行っている。

従来のプログラミング言語はコンパイルを効率よく行う為に文法に制限を課すという方向性で研究されていた。これはメモリが高価で CPU 速度も低速であったため、コンパイル処理で実効速度を得るためには文法を工夫する必要があったといえる。その為にあいまいでない文法を定義し、コードを１回走査するだけで解釈が可能になるよう様々な研究がなされた。

だが「言霊」は表現力を広げ、ユーザが望むようにプログラムを表現できることを目標にしている。コンパイラによる解析の都合で文法に制限を課すこともしていない。「言霊」は文脈自由言語のクラスに属し、非決定性プッシュダウンオートマトンの考え方に基づいて解析を行っている。

解析の例を挙げよう。図２０のようなコードを解析する場合は、以下のような手順を踏む。「言霊」には宣言文と代入文の２通りの文が存在したとしよう。宣言文と代入文の具象文法は図２１のように設定されているとする。コンパイラはまず宣言文の具象文法設定に基づき非決定性プッシュダウンオートマトン（以下、オートマトンと略記する）を生成し、図２０のコードがそのオートマトンで受理可能かを調べる。その結果受理しなか

ったので、図２０のコードは宣言文ではないことが分かる。続いて代入文であるかを調べる為に具象文法設定に基づきオートマトンを生成し、同じく受理可能かを調べる。その結果、図２２のように２通りの解釈ができることが分かる。どちらの解釈が正しいかは、代入文の代入先と代入値を解釈することで分かる。その解釈の為に、値を受理するオートマトンをさらに用意する。前者の代入値として解釈されている「１０と２０」という表現は値のオートマトンに受理しないため、これは代入値としての表現としては不適当であることが分かる。また「足したものを **a**」という変数は環境に存在しないことからも、前者の代入文の解釈は間違っていることが分かる。一方、後者の解釈はうまくいく。「１０と２０を足したもの」という表現は値のオートマトンに受理されるし、「**a**」という名の変数も環境に存在する。そこで初めて代入文の解釈は後者が正しいということが判明する。

１０と２０を足したものを **a** に代入する

図 ２０　代入文の例

宣言文	型を**<**型**>**、名前を**<**名前**>**として宣言する
代入文	**<**代入値**>**を**<**代入先**>**に代入する

図 ２１　宣言文と代入文の具象文法設定

[１０と２０**]　[**を**]　[**足したものを **a]　[**に代入する**]**
[１０と２０を足したもの**]　[**を**]　[a]　[**に代入する**]**

図 ２２　二通りの代入文解釈

「言霊」で特徴的なのは、あいまいな文法すらも設定可能にしていることである。「言霊」は自然言語に近いプログラミング言語なので多義性を持っている。

例えばユーザが「１０と２０を足したもの」という変数を宣言したとしよう。その場合は図２３のように２通りの解釈が可能になる。つまり「１０と２０を足したもの」という表現を、変数と解釈するか、足算の式と解

釈するかの２通りが出てくる。

このような場合、コンパイラは２通りの解釈が可能であるという警告をユーザに示し、あいまいでない表現に変更するようユーザに促すことで解決をする。この場合は変数名を変更するか、足算表現の具象文法設定を変更することで解決する。普通にプログラムを記述していればこのような場面に遭遇することはそんなに多くないと思われるので、この解決方法で十分対処が可能ではないだろうか。

プログラミング言語があいまいであってはいけない理由は､存在しない。あいまいな言語を解析するにはメモリと CPU パワーが必要なのだが、これまでは高性能なメモリと CPU が高価であったために解析が行えなかった。だからあいまいでない言語を定義しコンパイル速度を上げるような研究を積み上げてきたのだ。だが現代は高性能なメモリと CPU が安価に手に入る時代であり､「言霊」はそれを用いてあいまいにすらなりうる言語を解析しているのである。

```
[１０と２０を足したもの] [を] [a] [に代入する]
[ [１０][と][２０][を足したもの] ] [を] [a] [に代入する]
```

図 ２３ 二通りに解釈ができる代入文

１０．初心者と熟練者の双方に配慮した「言霊」

「言霊」はスタックを意識した低級表現と、意識しない高級表現の両方を許すことにより、初心者と上級者の両方にとって有効な言語になっている。例えば「**a=1+2**」という意味のコードを記述したいとする。「言霊」においてスタックを意識せずに高級な表現で記述すれば「１に２を足したものを **a** に代入する」などという表現になる。初心者にとってスタックの概念の理解は難しいが、このように高級な表現で記述が可能であれば学習が容易になる。

一方でスタックを意識した表現を許せば「１に２を足したものを積む。**a** に代入する。」あるいは「１を積む。２を積む。整数を足す。**a** に代入す

る。」のように低級な表現と高級な表現を混ぜることができる。

低級表現と高級表現の混在という仕組みは、上級者にとって教育と実用の観点から有用である。すでに基本的な学習を終えたプログラマでも、「言霊」処理系を高度に使いこなす為にはスタック操作を理解することが必要である。低級な記述が可能であれば、スタックを意識してプログラムを記述することで処理系の学習につながる。また効率的なプログラムを記述する為には、コンパイラによる最適化に期待するよりもユーザ自身が効率的なプログラムを自分で記述できることが望ましい。低級な記述が可能であれば、その状況において最適なコードをユーザが記述することができる。

１１.「言霊」の今後の課題

「言霊」を用いて自然な日本語表記を行うためには未解決な問題も数多くある。その一つが語順の問題である。「言霊」を用いてプログラミングの授業を行った際に最も多かった指摘が、語順を間違えて記述してしまうことだった。例えば「９０度右に曲がる」と記述するところを「右に９０度曲がる」と記述してしまう。解決策として一つあるのが「～に、～度、曲がる」と文節に区切って手続き宣言して、文節の順序が変わっても受理するようなコンパイラを実現することである。もう一つの解決策が形態素解析を行って得た情報を元に解析する方法である。

「言霊」の表記の問題として、同義語の問題も存在する。「言霊」を用いた授業の中で「亀が筆を下ろす」を「亀が筆を下げる」と記述してしまう例が多く見られた。これは第１章で述べた、一般用語としてプログラムを記述してしまう問題である。これを解決するには教育を工夫するほかに、開発環境の整備があるだろう。現在のソフトウェア開発環境のほぼ全てが実現しているコードインサイトの機能を使って、ユーザが宣言されている手続きの中から選択してプログラムを記述すれば問題は起きないはずである。

他にも「言霊」を記述する上での問題点は数多く存在するが、それらは今後の課題である。

また今後の課題として「言霊」を用いると本当に情報教育が改善されるのか、その教育効果を測定する必要がある。Mind には BASIC の授業と比較してどれほど効果が上がったかを測定した研究があり[8][9]、「言霊」においても同等の効果が期待できる。しかし「言霊」を用いて授業を行ったことはあるものの、その授業の結果について具体的なデータを取り、他の言語の授業と比較して教育効果が上がったのかを測定したわけではない。今後は実践事例を試行し「言霊」の教育効果について探求しなければならない。

１２．おわりに

完全な日本語として書かれたものが、プログラムとして Java の実行環境で動作するようなプログラミング言語「言霊」について述べた。「言霊」は日本人の初心者にプログラミング教育を行う場合、文法事項について教育する必要がないため、プログラム作成自体に集中できる。従来のプログラミング教育では、言語の文法を修得することにほとんどの努力が集中されるために、実現したいソフトウェアを制作できる能力をつけることが困難で、他人の作ったプログラムを理解し、それを改編する程度の教育しか行なわれてこなかった。「言霊」の完成と教育法の開発により、プログラミング教育が大きく変わるものと期待される。

日本語で表現することが苦手な情報処理技術者には、「言霊」のような日本語プログラミング言語は使いにくいという評価が与えられがちである。一方、日本語がうまく書けない情報処理技者の存在が問題になっている。プログラムを日本語で書くことで、日本語の苦手な技術者の意識が変わる可能性がある。

プログラムに無縁の文科系研究者には、「言霊」のようなプログラミング言語は歓迎される。このような言語が利用可能になるなら、自分でもプログラムを書いてみたいという希望を述べる人が多い。これに成功すれば、コンピュータと人間の関係を大きく変える可能性がある。

これまで、自然言語によるコンピュータへの指示は、データベースの問

合せ言語などで試みられてきたが､広く利用されるまでには至っていない。「言霊」の利用者が広がることによって、一般人のコンピュータ理解が深まり、情報社会の在り方が変わる可能性もある。

参考文献

[1] 水谷静夫**:**「朱唇の手引き」東京女子大学 日本文学科 **107pp. 1989** 年
[2] 芹沢浩**:**「日本語 **Logo** 入門」森北出版**(**株**)**、**183pp. 1994** 年
[3] 中川正樹、早川栄一、並木美太郎、高橋延匡**:**「母語プログラミングの理念、実現、実践とその効果」電子情報通信学会論文誌、 **Vol.J77-D-1 pp.364-374, 1994** 年
[4] 小谷善行**:**「日本語に基付く論理プログラム表現」情報処理学会論文誌、**Vol.34, pp.973-984, 1993** 年
[5] 今城哲二**:**「日本語プログラムの設計」、第 **40** 回プログラミング・シンポジウム報告集、情報処理学会、**pp.7-19, 1999** 年
[6] 木村明、片桐明:「日本語プログラミング言語『**Mind**』について　その概要と､日本語プログラミングの実用性」 プログラミング言語 **16-4**、情報処理学会　**pp.25-32 1988** 年
[7] 西之園晴夫:「情報教育用言語としての **Mind** の文法と記述法」**pp371-374**、**1989** 年、日本科学教育学会
[8] 林徳治、西之園晴夫:「教育用言語に望まれる文法と記述について」、日本教育工学会第６回講演論文集、**pp359-369**、**1990** 年
[9] 藤本光司、林徳治、西之園晴夫:「『情報基礎』領域における実証的取り組み　第４報:教育用 **Mind** を使ったプログラミング学習に関する授業実践について」､日本教育情報学会　第７回年会研究､**pp17-20**､**1991** 年
[10]Bertrand Meyer:「プログラミング言語理論への招待」アスキー出版局、**450pp**、**1995** 年
[11]兼宗進、中谷多哉子、御手洗理英、福井眞呉、久野靖:「オブジェクト指向言語「ドリトル」を利用した情報教育について」　情報処理学会「情報教育」シンポジウム **Vol.2001,No.9 pp.275-282 2001** 年