

修士論文 2002年度(平成14年度)

日本語プログラム言語「言霊」

慶應義塾大学 大学院

政策・メディア研究科 修士課程

岡田 健

日本語プログラム言語「言霊」

論文要旨

プログラム教育とは、自分の考えを論理的にプログラムで表現する能力を育成することである。それを通じて論理的な思考を鍛えてコンピュータに対する理解を深めて、自分のやりたいことをコンピュータ上で表現するのが目的の、教養教育である。ところが現実のプログラム教育ではプログラム言語の文法を理解して使いこなすことに終始してしまう。これでは自分の考えを表現できるようにはならない。

本研究では、正しく表記された日本語を用いて記述できるプログラム言語「言霊」を設計・実装した。日本語をプログラム言語として用いると文法教育を必要としない為、日本人に対するプログラム入門教育ではプログラムの表現能力育成に集中できる。

「言霊」は正しい日本語が使用できることや具象文法の設定を変更できる事により、可読性の高いプログラムを記述できるよう設計されている。その為初心者向けに読みやすいプログラムを記述することが可能であり、また分かりやすいプログラムを初心者が記述することも可能である。

また一方で「言霊」は Java の実行環境である仮想計算機のアセンブラを日本語化し、低級な機械語表現から高級表現まで日本語を用いてシームレスに表現できる。機械語レベルの細かい処理を記述できる為、コンパイラでは不可能な最適化をプログラマが行うことができる。その為、熟練プログラマの為の言語としても有効性が期待できる。

キーワード

1. 日本語プログラム
2. プログラム入門教育
3. 抽象文法
4. 具象文法
5. バイトコード

Japanese Programming Language “ Kotodama ”

Summary

A purpose of programming language teaching is to be able to express own thought logically by programming language. Through a education of programming language, One train logical thinking and deepen know a computer better, and One become to expression a thing which one want to do. It ' s general education. But actually programming education is only to learn a grammar of programming language and only to use it.

I produce Japanese programming language “ Kotodama ” which can describe in Japanese. If I use a Japanese programming language for programming language education for beginner, I don ' t need education for grammar and trainee concentrate to train expression skill.

“ Kotodama ” which can change configuration file of abstract syntax and concrete grammar is designed to describe readable program in Japanese. This language can describe readable program for beginner, and beginner can describe readable program.

On the other hand, “ Kotodama ” can describe program with low-class description and high-class description in seamless. Because a programmer can describe in low-class description, programmer can optimize a program. So, “ Kotodama ” is not only for beginner, but also for skilled programmer.

Key Word

1. Japanese programming language
2. A education of programming language for beginner
3. Abstract syntax
4. Concrete syntax
5. Java byte code

Keio University Graduate School of Media and Governance

Ken Okada

目次

第1章	はじめに	6
第2章	既存の日本語プログラミング言語の研究	9
第1節	プログラム言語の歴史	9
第2節	日本語COBOL・日本語LOGO・N ₈₈ -日本語BASIC	11
第3節	日本語AFL	12
第4節	小朱唇	14
第5節	Mind	16
第1項	背景	16
第2項	Mindの言語仕様	17
第3項	Mindの考察	19
第6節	AppleScript	21
第3章	日本語プログラミング言語「言霊」	22
第1節	概要	22
第2節	日本語バイトコード	25
第1項	Javaの仕組み	25
第2項	日本語バイトコードの概要	27
第3項	日本語バイトコードの表現設定	29
第4項	日本語バイトコードの効果	31
第3節	日本語プログラミング言語「言霊」	33
第1項	概要	33
第2項	「言霊」における文法の扱い	35
第3項	抽象文法	36
第4項	具象文法	42
第5項	具象文法の実装の仕組み	43
第6項	低級表現と高級表現の混在	53
第4章	「言霊」を用いた実験授業	55
第1節	授業概要	55
第2節	第一回目の授業	55
第3節	第二回目の授業	56
第1項	プログラム作成の方法	56
第2項	プログラムの概要	56
第3項	繰り返し文	58
第4項	場合分け	60
第4節	第三回目の授業	62

第5節	授業の感想について.....	66
第5章	今後の課題と展望について.....	68
第1節	語順の問題.....	68
第2節	動詞活用に対応した言語仕様.....	69
第3節	文脈を使った記述.....	71
第4節	構造エディタを用いた開発環境.....	71
第6章	終わりに	72
第7章	参考資料	73
第8章	謝辞	74

第1章 はじめに

大学において行われるプログラミング教育の目的は、二つある。一つは論理的思考能力を養うためであり、もう一つはコンピュータの動作原理を知ることによって主体的にコンピュータを使いこなせるようになるためである。

論理的思考能力とは、漠然とした目的を細かい手段に置き換えることを可能にする能力を指す。例えば以下のような能力のことである[1]。

- ・ 漠然としたアイデアを具体的な「目的」に置き換える能力
- ・ 「目的」を実現するために必要なプロセスを、正確に順序だてて組み立てる能力
- ・ 組み立てたプロセスを正確かつ簡潔に記述する能力

こうした能力があることにより、自分のアイデアを表現し、他人に伝え、そして実現することができる。多くの会社が新入社員教育の中でプログラミング教育を行うのは、こうした能力は仕事をする能力と同義であるからである。

プログラミング教育のもう一つの目的は、コンピュータの動作原理を学ぶことである。コンピュータの大部分はプログラムによって動作しており、プログラムを理解することで通常は表面に現れないコンピュータの性質を理解できるようになる。この先パソコンの普及率はさらに上がり様々な局面でパソコンを使いこなす必要が出てくるだろうが、その為にはコンピュータの動作原理に対する最低限の理解は必要である。

だが現在広く行われているプログラミング教育は、ソフトウェアを作れるようになる為の実用教育に近い。プログラミング言語の文法教育や記述パターンを利用する事で、自分で作ったプログラムを動作させるということが目的となってしまう。その為ソフトウェアが作れるという実用目的が達成されるだけで満足してしまう。

原因の一つにプログラミング言語の問題がある。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るため、特に英語の影響を強く受けている。その為日本人がプログラムを読み書きするとき、日本語の思考と英語の思考を相互に翻訳する必要がある。

英語の影響の一つとして語順の問題がある。英語の語順は、まず動詞が来て続いて目的語が来る順序になっている。例えば Lisp では図 1 - 1 のような記述をする。一行目は $4 + 5$ の計算をし、二行目は $4 \times (1 + 2)$ の計算を行っている。表記を見ると分かるが、まず最初に動詞が来て、続いて目的語を記述する語順になっている。また図 1 - 2 のように、Java などにおけるメソッド呼び出しも、まず最初に動詞を記述し、続いて目的語となる実引数を括弧で括って記述する。だが日本語では目的語が最初に来て、最後に動詞は来る。こうした語順の違いがあるため、日本人は日本語により頭の中で処理を描いた上で、それを英語に直してプログラムを記述する必要がある。

```
( + 4 5 )  
( * 4 ( + 1 2 ) )
```

図 1 - 1 Lisp プログラム例

```
drawLine(10,10,100,100);  
rectangle(100);
```

図 1 - 2 Java によるメソッド呼び出しの例

またプログラミング言語の表現がプロフェッショナル向けに洗練されているために、学生が日本語で思考するのに向かないという問題もある。例えば慶應義塾大学湘南藤沢キャンパスで Java 言語を用いて教育を行っているが、この言語は業務用ソフトウェアを製作する為の完全なプロフェッショナル用言語である。その為にプロフェッショナルが使う分には価値がある様々な表現が、学生には害になることが少なくない。例えばプログラムを学ぶ人が最初に見る HelloWorld プログラム (図 1 - 3) を記述するだけで、様々なキーワードを使わなければならない。もちろん全てのキーワードには意味がありプロフェッショナルの視点で見ればどれも重要な記述なのだが、初心者にとっては大変な負担になる。この HelloWorld プログラムには静的 (static) の概念が使われているが、これは 1 年以上プログラムを勉強してから学べばいい概念といって良い。また図 1 - 4 のプログラムのように “ % ” や “ == ” などの記号の意味は、学生には全く分からない。

```
public class MyApplet extends Applet{  
    public static void main( String args[] ){  
        System.out.println( “ Hello World! ” );  
    }  
}
```

図 1 - 3 Java による HelloWorld プログラムの例

```
if( num % 2 == 0 ){ rectangle(100); }  
else{ triangle(100); }
```

図 1 - 4 Java による、“ % ” や “ == ” を用いたプログラム

「分からなければ教えればよいではないか」という意見もあるが、プログラム表記の意味が分からないと、まず学生は学習する気を失ってしまう。学ぼうという意思が生じる前に「自分にはこんな複雑なものは扱えない」という諦観が先に出てしまうのだ。極端な例として、先のもので全く同じ意味のプログラムを三項演算子を用いて記述した例を図 1 - 5 に示す。これを見せられると、その記号性の高さからほぼ例外なく学生は理解しようという気をなくす。

```
num%2==0?rectangle(100):triangle(100);
```

図 1 - 5 Java による、三項演算子を用いたプログラム

だが学生にとっては図 1 - 5 のようなプログラムは難解だが、プロフェッショナルにと

っては記述したい事を簡潔に表現できるありがたい表記法である。プロフェッショナルにとってのプログラム言語の表記法は、ある程度の読みやすさ（もちろんプロフェッショナルにとっての読みやすさである）と簡潔な記述性があることが望ましい。記述性を最大限に追求した Perl [2]などは図 1 - 6 のような記述をする。

```
$_ = ' ab12cd ' ;  
s/¥d+/$&*2/e;
```

図 1 - 6 Perl による、正規表現を用いた複雑なプログラム

総じてプログラム言語の発展のベクトルには、読みやすさを多少犠牲にして簡潔な記述性を追及する傾向がある。プログラム言語は特殊な知識を有するプログラマだけが扱うから多少読みにくくても理解してくれるだろう、という前提があるのだろう。プログラマはプログラムを作成するのが仕事だから、むしろ可読性より記述性を優先してしまうという背景もあるのだろう。もちろん保守性を考えると可読性を疎かにはできないのだが、一般的には記述性を重んじている。

大学におけるプログラム教育においては、教育に適したプログラム言語が必要である。その為には学生にとって前提知識が少ない状況でもある程度読み下せるようなプログラミング言語であるべきだし、日本語の思考を妨げないようにプログラム記述ができるべきである。その為には、日本語でプログラムを記述できることが最良の解決策である。

本研究では日本語プログラミング言語「言霊」を開発した。過去にも筆者と同じ考えの下に日本語プログラミング言語が開発されてきたが、それらと特徴を異にしている点はいくつかある。一つは抽象文法を導入することにより、非常に強力な表現力を実現したことである。その為に解析手法が既存のプログラミング言語と異なり、非決定性オートマトンを用いて字句解析と構文解析を同時に進める必要がある。もう一つは、機械語などの低級表現と高級表現を、同じ日本語を用いてシームレスに表現ができる点である。これにより日本語プログラム言語でありながら Java 言語よりも高速で効率の良いプログラムを開発できる可能性がある。

以下、第 2 章ではこれまで行われてきた日本語プログラミング言語の研究について、第 3 章では日本語プログラミング言語「言霊」について、第 4 章では「言霊」を用いて行われた授業について報告する。第 5 章では「言霊」の問題点と展望について考察する。最後に第 6 章でまとめる。

第2章 既存の日本語プログラミング言語の研究

第1節 プログラム言語の歴史

プログラム言語の歴史とは、より良いプログラムを作ることができる言語を追求した歴史である[3]。良いプログラムとは、可読性・簡潔性・移植性・拡張性・堅牢性などの性質に優れたプログラムである。本論文の中で日本語プログラム言語に関係のあるのは可読性と簡潔性である。

可読性の高いプログラムとは、コードを読めばその記述意図が理解しやすいプログラムのことである。可読性の高いプログラム記述を可能にした印象的な言語といえば、FORTRAN である。FORTRAN が登場するまではプログラミング言語が存在せず、専らプログラムはコンピュータ実装よりの機械語による記述だった。例えば 1953 年に IBM701 コンピュータ用として開発された Speedcording (スピードコーディング) を用いて $2 \times 3 + 4$ の計算をするプログラムを記述すると、図 2 - 1 のようになる。そこで数列を記述すればコンピュータに必要な機械語に変換してくれるプログラムを作れないかと考え、その結果できたのが FORTRAN だった。FORTRAN を用いてまったく同じ内容のプログラムを記述すると図 2 - 2 のようになる。FORTRAN を使う事により、数式を機械語に変換するという作業を、人間からコンピュータに任せることが可能になった。

200 SET 100 2	(100 番地に 2 を入れる)
201 SET 101 3	(101 番地に 3 を入れる)
202 MUL 100 101 102	(100 番地と 101 番地を掛けて、102 番地に結果を入れる)
203 SET 103 4	(103 番地に 4 を入れる)
204 ADD 102 103 104	(104 番地に結果を入れる)

図 2 - 1 Speedcording による、 $2 \times 3 + 4$ を計算するプログラム

result=2*3+4

図 2 - 2 FORTRAN による、 $2 \times 3 + 4$ を計算するプログラム

簡潔性の高いプログラムとは、短い記述でやりたいことを表現したプログラムである。C 言語や Java や Perl などの代表的な言語はそうした特徴を備えているが、その最たるものは APL である。普通の言語だと 10 行ほどの長い記述が必要な部分でも、APL だと 1 行で表現することが可能である。例えば図 2 - 3 のようなコードを記述すると、APL はキーボード入力を待つ。ここで「10」と入力すると、続いて 10 個の数値入力を受け付ける。10 個の数値をスペースで区切って入力すると、その 10 個の平均値を出力する。APL 以外でも、C 言語や Java にも三項演算子のような簡潔な記述が可能な文法が存在する。こうした簡潔性は記述がしやすくなる反面、可読性を犠牲にする傾向がある。APL のプログラムは記述性が高いためプログラムの意図が分かり難く、覚えてから使いこなせるようになるまでが大変な言語だったようだ。

$$(+ / Y) \div Y$$

図 2-3 APL によるプログラム例

このようにプログラムの可読性・簡潔性を高め、より読みやすく書きやすいプログラム言語を作ろうという潮流はコンピュータの歴史の初期段階からあった。だがコンピュータの研究・開発に多く携わっていたのは欧米の人間であり、読みやすさ・書きやすさの基準もまた欧米を基準にしたものだった。英語と日本語はまったく異なる言語であるため、日本人は日本人にとって読みやすく書きやすいプログラム言語を作る必要があった。

日本語プログラム言語の研究の中で代表的なものは、日本語 COBOL・日本語 LOGO・N₈₈-日本語 BASIC や日本語 A F L や小朱唇や Mind などがある。これらを一つ一つ紹介していく。

第2節 日本語COBOL・日本語LOGO・N₈₈-日本語BASIC

初期の日本語プログラム言語の研究で中心を占めたのは、プログラム開発の中で日本語環境をどのように整えるかであった。1960年代はプログラム言語で使用できるのがアルファベットと一部の記号のみである為、日本語の片仮名や平仮名・漢字を用いて記述することがまず求められた。その為に文字コードや漢字変換などの Front End Processor などに関する研究が行われた。

プログラム言語を日本語化するという研究にもいくつかの段階が存在した。COBOL における日本語機能の現状と今後の動向[4]によると、COBOL における多バイト文字のサポート範囲を以下の4つのレベルに分けている。

- | | |
|-------|---|
| レベル1： | データとして多バイト文字が扱える |
| レベル2： | レベル1に加えて、データ名や手続き名などの利用者語（識別子）に多バイト文字が使用できる。 |
| レベル3： | レベル2に加えて、MOVE・ADDなどの予約語（構文）も多バイト文字とする。
（プログラムテキストを全て多バイト文字とする） |
| レベル4： | レベル3に加えてデータも全て多バイト文字とする。 |

図 2-4 COBOL における多バイト文字のサポート範囲

レベル1の段階としてDOD COBOL [5]が挙げられる。この言語の文法規則によると、入出力データ、定数、コメントなどの表現に仮名文字を入れることが可能になっている。

レベル2の段階として仮名COBOL [5]が挙げられる。DOD COBOLの文法規則をそのまま、データ名や手続き名などに仮名を用いることが可能になると、それだけで親しみやすく読みやすいプログラムを書けるようになる。以下にコード例を紹介する。

READ	オヤ - ファイル	AT	END	GO	TO	オワリ .
MOVE	ミダシ	TO	ミダシ - W .			
IF	キンガク = 0	GO	TO	ケイサンオワリ .		

図 2-5 仮名COBOLのプログラム例

第3節 日本語AFL

日本語COBOLなどのように既存のプログラム言語の日本語化ではなく、独自の日本語プログラム言語として日本語AFL [6] (A Fundamental Language) がある。1983年に松下技研(株)が発売し、数百本の販売実績を持つ。

面積 は [底辺 * 高さ / 2]。 底辺 は 20、高さ は 15。 面積 は 計算し、結果 を 表示する。	結果 : 150
--	----------

図 2-6 日本語AFLのプログラム例

日本語AFLの特徴は、単語の定義を重ねることでプログラムを作る点である。例えば図2-6の例では「面積」という単語を定義し、それを実行することでプログラムが進行していく。既存言語のメソッドやサブルーチンという概念に近いものがあるが、違うのは日本語AFLではデータとメソッド・サブルーチンの間に区別が存在しないという点である。例えば図2-7では「文章」という変数に入っているのは「こんにちは。」という文字列データである。だが図2-6では「面積」という変数に入っているのはサブルーチン処理である。

文章 は [こんにちは。]。 文章 を 表示する。

図 2-7 日本語AFLのプログラム例

また日本語AFLには豊富な基本構文が用意されている。その一部を図2-8に挙げる。全体で26種類の基本構文が用意されている。また基本構文は37種の述語、49種の修飾語、13種の助詞、11種の記号から成り立つ。これらの要素を組み合わせることでプログラムを作成する。

代入文	A は B。 A は B である。
実行文	A を 計算する。 A を 実行する。
比較文	A が B より 大きい か 判断する A が B と 等しい か 判断する A が B より 小さい か 判断する
算術演算文	A に B を 加える。 A から B を 引く。 A に B を 掛ける。 A を B で 割る。

図 2-8 日本語AFLの基本構文

日本語 A F L は日本語プログラミング言語として先駆的な役割は果たしたものの、構造化プログラミングという当時のプログラミング言語の潮流を無視した感が否めない。日本語 A F L は単語を増やしていくという形で部品化を行うが、それらの部品のモジュールの独立性を高める為の工夫が必要とされていたであろう。だが日本語 A F L ではローカル変数の概念が存在しない為、モジュールの独立性を保つことが難しかった。

代入述語	である / とする
実行述語	実行する / 計算する
比較述語	判断する
算術述語	加える / 足す / 引く / 掛ける / 割る / 変換する

図 2-9 日本語 A F L の述語

算術比較修飾語	大きい / 小さい / 等しい
文字比較修飾語	一致する / 含まれる / 空
選択修飾語	正しい / 間違い

図 2-10 日本語 A F L の修飾語

基本助詞	は / を / から / に / の / が / と / で
修飾助詞	まで / か / より / なら / へ

図 2-11 日本語 A F L の助詞

文記号	。 / , / / [] / { } / < >
算術記号	+ / - / * / / / ()

図 2-12 日本語 A F L の記号

第4節 小朱唇

小朱唇[7][8][9]は初心者が対象の日本語プログラミング言語である。開発者の水谷静夫氏は国文学者であり、学生がコンピュータと小朱唇を用いて以下を行うのを目的としている[7]。

- 1) 資料調製 a) 言語材料の整理・探索・編集
b) 上記の結果の統計解析など(数値計算)
- 2) 言語・文学の理論上の仮説の検証(立証又は反証)

一例を挙げる。図2-13は入力文字に応じて俳句を作成するプログラムである。その出力結果が図2-14である。

```
REI1 ハクドキ(夕夕夕 ニ):  
「 ¥「.¥」 ノ アト ニ ダイ (4モ-ラ) ヲ イテ クダサイ。」ヲ 拵。  
「 ヤメル ニハ ¥「/」 ヲ!」 ヲ 拵。  
メグリ  
∴ Xニヨメ。  
シカ  
∴ X=「/」 ナラ ヤメヨ;  
ホハ X_「ヤア」_X_「ヤ」_X_「ヤ」 ヲ 拵 ∴ ∴。  
初リ
```

図2-13 小朱唇によるプログラム例

```
「.」ノアトニダイ(4モ-ラ)ヲイテクダサイ。  
ヤメルニハ「/」ヲ!  
.メグツ  
メグツヤア メグツヤ メグツヤ  
.ツノヨ  
ツノヨヤア ツノヨヤ ツノヨヤ  
.ウメヲ  
ウメヲヤア ウメヲヤ ウメヲヤ
```

図2-14 図2-13のプログラムの出力結果

図2-14において下線部は端末からの入力であり、プロンプトが「.」である。このプログラムでは「/」が入力されれば実行が終わる。分かりにくいのは括弧の存在で、「∴」と「∴」がそれぞれ開き括弧と閉じ括弧になる。このプログラム例では繰り返し文の括弧があり、その中に仮定文が入っている。「_」は文字列連結記号である。入力文字とキーワードをつなげる事で俳句を作っている。

もう一つのプログラム例を挙げる。これは図 2 - 1 3 のプログラム例を改造したものである。

```
REI2 シュツヨク ニ トホバ`ンガ`ウヲフル :  
  0 ==> NR。  
  「メイ`ツ、ツユ ノ ヨ、ウメ`ル、」ヲ DAI ニ ウツ。  
めぐり  
∴ DAI ヲ #X_「、」_#DAI ニ ワカ。ナレバ`ヌダ`セ。  
  (NR 1 +) ==> NR。X_「ヤ」ヲ X ニ ウツ。  
  NR_「) 」_X_「ア」_X_「) 」_Xヲカ ∴。  
初り
```

図 2 - 1 5 小朱唇によるプログラム例

```
1) メイ`ツヤ アア メイ`ツヤ メイ`ツヤ  
2) ツユ ノ ヨヤ アア ツユ ノ ヨヤ ツユ ノ ヨヤ  
3) ウメ`ルヤ アア ウメ`ルヤ ウメ`ルヤ
```

図 2 - 1 6 図 2 - 1 5 の出力結果

このプログラム例では、扱うべき文字列がハードコードされている。「メイ`ツ、ツユ ノ ヨ、ウメ`ル、」という文字列が DAI 変数に保存される。そしてこの文字列を分割しながら俳句を作っていく。また何番目の俳句なのかを保存する変数が NR であり、これが繰り返し文の中で増加していく。

小朱唇は文字列処理を主目的としていて、自然な語法と豊富な機能を有していた。だが一方で開発が大学のメインフレームを対象に行われてしまい、使用できるのが半角カナだけであるのが問題であったし、解説書の類が全て文語体で記述されている事もあって一般に普及はしなかった。

第5節 Mind

最も実用性が高く広く受け入れられた日本語プログラミング言語が Mind[11][12]である。スタックマシン及び逆ポーランド記法と日本語との相性のよさに着目し、米国産でスタックマシンの機構を持つプログラム言語 FORTH[10]を日本語化して作られた。(株)リギーコーポレーションが開発し、NECのPC9800シリーズ・富士通FMシリーズ・日立2020などを対象に1985年12月に発表して以来、約2400本の販売実績がある。

第1項 背景

MindはFORTHに徹底した日本語化を行い、使い勝手をよくした言語として位置づけることができる。FORTHにはスタックという概念がその中心にあり、数値は全てスタックに積まれていくだけで変数を極力使用しないように設計されている。この「数値はすべてスタック上にある」という事実により、「A + B」という加算計算をFORTHで行おうとすると、以下のような実行手順になる。

1. 数値Aをスタックに積む
2. 数値Bをスタックに積む
3. (スタック上のデータを取り出し)加算する(その結果がスタックに積まれる)

これをそのまま素直に式に書いたものがFORTHの文法である。この加算計算は

A B +

と記述される。この間の処理とスタックの状態を図2-17に示す。

記述	初期状態	A	B	+
スタックの状態	空	A	B A	A + B

図 2-17 FORTHのスタックを用いた計算

この記述は逆ポーランド記法と呼ばれるもので、演算子(+や-など)等の指示語が一番最後に来る。そして「逆ポーランド記法は日本語との相性がいい」という事実があり、例えば

A B +

は

A と B を 足す

と読めば、そのまま日本語の文法に当てはまる。この発想に基づきFORTHの分かりにくさを改善するために日本語化を行い、その結果として生まれたのがMindである。

第2項 Mindの言語仕様

・送り仮名

Mindでは識別名として漢字とカタカナしか認識しないシステムになっているため、「・・・を」とか「・・・する」などの送り仮名は自然な日本語表記のために、自由に記述できるようになっている。送り仮名をどのように振ってもコンパイラは同一の単語として認識するので、ユーザは送り仮名によってその特徴を出すことができる。例えば、次の3つの表現は全て等価である。

- a. 「今日は」 表示
- b. 「今日は」を 表示する
- c. 「今日は」を 表示することである

aは識別名を並べただけであるのに対し、bとcは自然な日本語表現になっている。送り仮名の中には無条件に無視されるものの他に処理を行う特殊な送り仮名も用意されている。

から、を、より	データの転送元を示す送り仮名
に、へ	データの転送先を示す送り仮名
の	修飾を示す送り仮名
と	併記を示す送り仮名
は、とは	単語定義の開始を示す送り仮名

図 2-18 Mindで使用される特別な送り仮名

プログラムの構文の一部には、この特殊な送り仮名を強制されることがある。

3を 生徒数に 入れる。

においては、データの転送先を示す「に」がキーワードとして強制される。

またMindでは数値の判断にユニークな方法を取っており、「数字から始まる単語は全て数値である」と認識し、数字以外の後続文字は全て無視される。これによりユーザは自由に計数単位を記述することが可能になり、プログラムが見やすくなる。例えば

95点を 成績(5番目)に 入れる。

というプログラムでは、95点は95として、5番目は5として認識される。

・変数

Mindは先にも述べたように、スタックという概念が取り入れられており、処理単語間のデータの受け渡しはほとんどの場合はスタックを用いるようになっているが、変数の使用も可能である。スタックと変数の使い分けは、短期データはスタックで長期データは変数を使うことになる。

変数の種類は、数値用として「変数(32ビット変数)」、文字列用として「文字列」「文字列実体」が用意されている。

変数は次のように宣言する。

```
成績は 変数。
名前は 文字列。
```

図 2-19 Mind における変数宣言

Mind では無制限で変数を使うことができず、全てあらかじめ宣言を行う必要がある。またプログラム全体からアクセスするグローバル変数のほかに各処理単語内だけでアクセスされるローカル変数の使用も許されている。

数値に値を格納するには「入れる」という処理単語を使う。

```
50点を 成績に 入れる。
```

この他にも変数だけを操作する特別な処理単語として「クリア」、「一つ増加」、「一つ減少」、「増加」、「減少」の5つが用意されている。

・分岐

Mind での一般的な分岐の記述は以下の通りである。

```

. . . . .                条件判定処理
    ならば . . . . .      真の場合の処理
        . . . . .
    さもなければ . . . . . 偽の場合の処理
        . . . . .
    つぎに                二つのパスの合流地点を示す。
. . . . .                後続処理

```

図 2-20 Mind における分岐の一般的な形式

この構文で「つぎに」というやや不自然な表現が使われているが、これは他の言語の ENDIF に相当するものである。また「さもなければ」の項は ELSE に相当し、不要な場合は省略が可能である。

・繰り返し

繰り返しには、回数指定によるものと条件によるものの2種類が用意されている。回数指定による繰り返しの記述は以下の通りである。

```

. . . . .を 回数指定し      繰り返しのはじめ
    . . . . . . . . . . . し  繰り返す処理
    . . . . . . . . . . . し  "
繰り返す                  繰り返しの終わり
. . . . .                  後続処理

```

図 2-21 Mind における繰り返しの一般的な形式

このループの制御変数はシステムが自動的に管理しており、ユーザは「回数」、「現回数」を記述することによりその値を参照できる。

条件による繰り返しの記述は以下の通りである。

```
ここから
    . . . . .
        ならば 打ち切り          「でなければ」を使ってもよい
        つぎに
    . . . . . し
    . . . . . し
繰り返す
```

図 2-22 Mindにおける条件を用いた繰り返しの一般的な形式

ある条件を満足するまで繰り返し処理を続け、「打ち切り」によりループから脱出する。

ループの制御文としては「打ち切り」「終わり」「もう一度」の3種類が用意されており、その機能は以下の通りである。

```
打ち切り：「繰り返す」の直後の箇所までスキップする。
終わり   ：「。」のところにスキップする。つまり1つの単語の処理を終了する。
もう一度：ループの先頭に戻る。
```

図 2-23 Mindにおける繰り返しの制御文

・サブルーチンと関数

Mindではメインルーチンとサブルーチン、サブルーチンと関数、どちらの区別もなく「処理単語」として以下の形で定義される。

```
<処理単語>とは          「は」または「とは」の送り仮名は強制される。
    . . . . .            . . . . .            . . . . .            . . . . .            定義済みの単語を並べる。
    . . . . .            . . . . .            . . . . .            . . . . .            最後には必ず「。」をつける。
```

図 2-24 Mindにおける処理単語定義の一般的な形式

パラメータは全てスタックを通して受け渡しが行われ、処理結果がある場合はそれもスタックに戻される。

第3項 Mindの考察

Mindはスタックマシンを計算上の中心概念とし、送り仮名をある程度無視する事により処理系に負担を掛けずにかなり自由な表現を許すという特徴を持っている。これにより読みやすい日本語を使ってプログラムが記述でき、かつ実用的な処理速度を持つプログラム

を作成することができた。これは富士通がMindを試用してパーソナル統合ソフト（FM秘書）を開発したことから分かる。Mindは過去の日本語プログラム言語の中では最も実用的な機能を持ち、かつ評価された言語である。

第6節 AppleScript

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、以下のような自然なプログラム表記が可能だった。

```
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 , 5 0 } 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 + 1 0 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
```

AppleScript の問題点のひとつは、変数の表現に奇妙な拘束が存在していることだ。AppleScript において、変数を生成して代入するには以下のように記述する。この場合の変数の終わりは「を」という平仮名になるので、平仮名と異なる文字種で変数宣言を行うぶんには解釈はできる。だが変数名に平仮名を使ったり、平仮名と漢字を両方使った変数名などを用いると、スペースによる分かち書きを強いられたり、変数名を括弧で括る必要が出てくる。このように AppleScript では変数表現を自由に記述できないという問題が存在する。

```
A を 123 にする。
```

AppleScript の問題点のもうひとつは、プログラムの記述が人によって自然な日本語と感じられないということである。AppleScript は過去の日本語プログラム言語の中では自然な日本語表現を実現しているのだが、表現というものをどう捉えるかは人それぞれになってしまう。例えば AppleScript における代入文は「～にする」という表現になっているが、筆者はいささか違和感を覚える。筆者ならば「～に代入する」という表現のほうが分かりやすいのではないかと考える。こうした批判が AppleScript のあらゆる表現について存在するのである。

第3章 日本語プログラミング言語「言霊」

第1節 概要

本研究では日本語プログラミング言語「言霊」を試作した。「言霊」を用いると Java による記述よりも読みやすく、結果的に理解を助けるプログラムが記述できる。図3-1に Java と「言霊」のソースコードを示す。このプログラムは両方ともturtleグラフィックスを用いて家を描くプログラムである。家とは四角形の上に三角形が載っている形を指す。

<pre>public class MyTurtle extends Turtle{ public static void main(String args[]){ house(100); //大きさが100の家を描く } //大きさがsizeの家を描く public static void house(int size){ triangle(size); //屋根を描く square(size); //本体を描く } // 長さがsizeの三角形を描く public static void triangle(int size){ rt(30); fd(size); rt(120); fd(size); rt(120); fd(size); lt(30); } // 長さがsizeの四角形を描く public static void square(int size){ for(int i=0 ; i<4 ; i++){ rt(90); fd(size); } } }</pre>	<p>メインとは { 大きさが100の家を描く。 }</p> <p>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 }</p> <p>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。 A歩進む。 右に120度曲がる。 A歩進む。 左に30度曲がる。 }</p> <p>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。 }</p>
--	--

図3-1 Java と「言霊」を用いた、turtleによって家を描くプログラム

言霊の設計で特に注意したのは、以下の3点である。

- ・自然な日本語によるコード記述を可能にする

言霊で最も重視したのは、自然な日本語を使ってプログラムを作成することである。その為にパソコン側の都合による恣意的な文法事項を極力排除して、プログラマが自由に記述できるようにした。

例えばメソッドの記述方法はかなり自由度が高い。図3-1の家を描くメソッドの表現が気に入らなければ、以下のように書き換えることもできる。

呼び出し部	一辺の長さが100の家を描く。
宣言部	一辺の長さが「A (整数型)」の家を描くとは{・・・}

こうした表現が冗長だというのであれば、自分なりに考案した記述をすることも可能である。

呼び出し部	辺100家描画。
宣言部	辺「A (整数型)」家描画とは{・・・}

また Java 言語のような従来型の宣言の表現を使用することも可能だ。

呼び出し部	家描画(100)
宣言部	家描画(「A (整数型)」)とは{・・・}

日本語ではなく英語でプログラムが書きたいなら、このように宣言することで可能になる。

呼び出し部	draw house whose length is 100.
宣言部	draw house whose length is 「A (整数型)」とは{・・・}

また、英語以外の言語でも原則的には可能である。言霊は Java プラットホームで開発されているのでその環境上で動作する言語ならば扱うことができる。だから全く同じやり方で韓国語プログラミング、中国語プログラミングも可能なのである。

・文法の日本語表現の選択はプログラマに任せる

メソッド呼び出しは自然な日本語で記述することができ、その記述方法はプログラマが自由に考えて記述することができる。これで代入文や宣言文などのように最初から言語に備わっている文章の表現を自由にできないのでは片手落ちである。

言霊は、代入文や宣言文などの言語の基本的な文の表現を自由に変更することができる。筆者は、代入文をどのような日本語で表現するべきか結局分からなかった。人によって代入文を「～を～に代入する」と表現したり「～を～に入れる」や「～を～に複写する」など、様々な表現が考えられる。思うに、どの表現が正しいかという問いに答えはない。ならばその問いに対して答える権利をプログラマに委ねる事にしたのである。

例えば、代入文は現状では以下のように記述することになっている。この情報は文法設定ファイルとして存在しており、コンパイラはコンパイル前にこのファイルを読み込み文法設定に応じたコンパイラを作ってから、コンパイルを実行する。

<値>を<変数>に代入する。

だがこの表現が気に入らなければ

<値>を<変数>に入れる。

と変更することもできる。またどちらの表現も捨て切れなければ

<値>を<変数>に代入する。
<値>を<変数>に入れる。

と設定すれば良い。どちらも解釈できるコンパイラができる。また語順の問題もあるだろ

うから、以下のような設定にするのが良いかもしれない。こうすることで、日本語的に語順が変化しても問題なく解釈できるコンパイラになる。

<p><値>を<変数>に代入する。 <変数>に<値>を代入する。 <値>を<変数>に入れる。 <変数>に<値>を入れる。</p>
--

このように文法の設定を自由に変更できることにより、プログラマは自由に自分の考える正しい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。その結果、いくらでも読みやすいプログラムを作成することが可能だし、いくらでも書きやすい文法を設定してプログラムを記述することが可能になる。

・高いパフォーマンスのプログラムを作成できる

日本語プログラム言語は一般的に遅く、重いプログラムができやすい。だが「言霊」は速く、小さいプログラムを作成することが可能である。その仕組みが、高級表現と低級表現を混在して記述することが可能である事だ。主要な部分は高級表現で記述し、パフォーマンスに強く影響を及ぼす部分に関しては低級表現で記述して、高パフォーマンスのプログラムを作成することができる。結果、言霊は Java 言語の上級者にとっても利益のあるプログラム言語となりうる。

以上の3点を念頭に置いて、日本語プログラミング言語「言霊」を開発した。以下では「言霊」の詳細を述べる。第2節では「言霊」の基礎部分を成している日本語バイトコードについて述べる。第3節では「言霊」に関して述べる。

第2節 日本語バイトコード

日本語バイトコードを述べるには、Java の仕組みに関する理解が必須である。そこで第1項では Java の仕組みを述べ、続く第2項以降で日本語バイトコードに関して述べる。

第1項 Java の仕組み

Java の基本コンセプトを表す言葉として「Write Once, Run Anywhere」がある。Java でソフトウェアを作成すると、様々な PC や様々な組み込み機器において同じように動作するという考え方であり、この移植性により Java は広く評価されてきた。Java のプラットフォームを用いることで、プログラムをいったん書いてしまえば (Write Once)、どんな環境でも動作する (Run Anywhere) のである。

Java はインターネットの普及とともに広く使用されるようになった。Java で記述されたアプリケーションは PC のハードウェア・ソフトウェア・OS の互換性で悩まされることが無いため、Java アプリケーションはインターネット上で広く普及させることができる。例えば Java アプリケーションは Windows の PC や Macintosh の PC や Linux の PC だけではなく、最近では一部の会社の携帯電話などのようにネットワークに繋がった電子機器ならばどれも Java アプリケーションが動作する可能性を持っている。

この移植性を実現しているのが Java Virtual Machine[13] (以下、JavaVM) という考え方である。Java アプリケーションの動作の仕組みは以下になる。Java 言語で記述したソースプログラムはコンパイルすることにより「バイトコード」と呼ばれる機械語命令を記述したファイル (クラスファイル) を生成する。バイトコードの命令を JavaVM が解釈することでプログラムが実行される。

例えば図3 - 2のように左側の Java 言語で記述されたプログラムは、右側のバイトコードに変換される。JavaVM はスタックマシンとして実装されているため、バイトコードもそれに従った命令群になる。例えば図3 - 2の「int a=0;」は、スタックに定数値0を積み、続く命令ではスタックトップに積まれた0を読んでデータ領域に保存するという処理になる。また「a=b+c*d;」は、スタックに b,c,d のデータをデータ領域からロードして順に積んでいく。次の命令は掛算命令なのでスタックトップから2つの値を取り出して掛け算を行い、演算結果をまたスタックに積む。その次の命令が足し算命令なのでスタックトップから2つの値、つまり c と d を掛けた値と最初の b の値を取り出して、足し算を行って、演算結果をスタックに積む。最後の命令がデータ格納命令なので、スタックトップから1つの値を取り出し、それを a のデータ領域に書き込む。

Java の移植性は、様々なプラットフォームで動作する JavaVM を作ることにより実現される。例えば Windows 環境で動作する JavaVM と Macintosh 環境で動作する JavaVM、Linux 環境で動作する JavaVM が存在する。これらの JavaVM の役目は、バイトコードの命令を解釈して実行することである。前述の Java プログラムをコンパイルして生成されたバイトコードは、Windows 環境では Windows 環境の JavaVM がそれを解釈してプログラムを実行する。

Macintosh 環境では Macintosh 環境の JavaVM がそれを同じように解釈してプログラムを実行する。このように Java アプリケーションは、JavaVM を通してバイトコードを実行させることにより様々な環境で同じように動作させることができる。

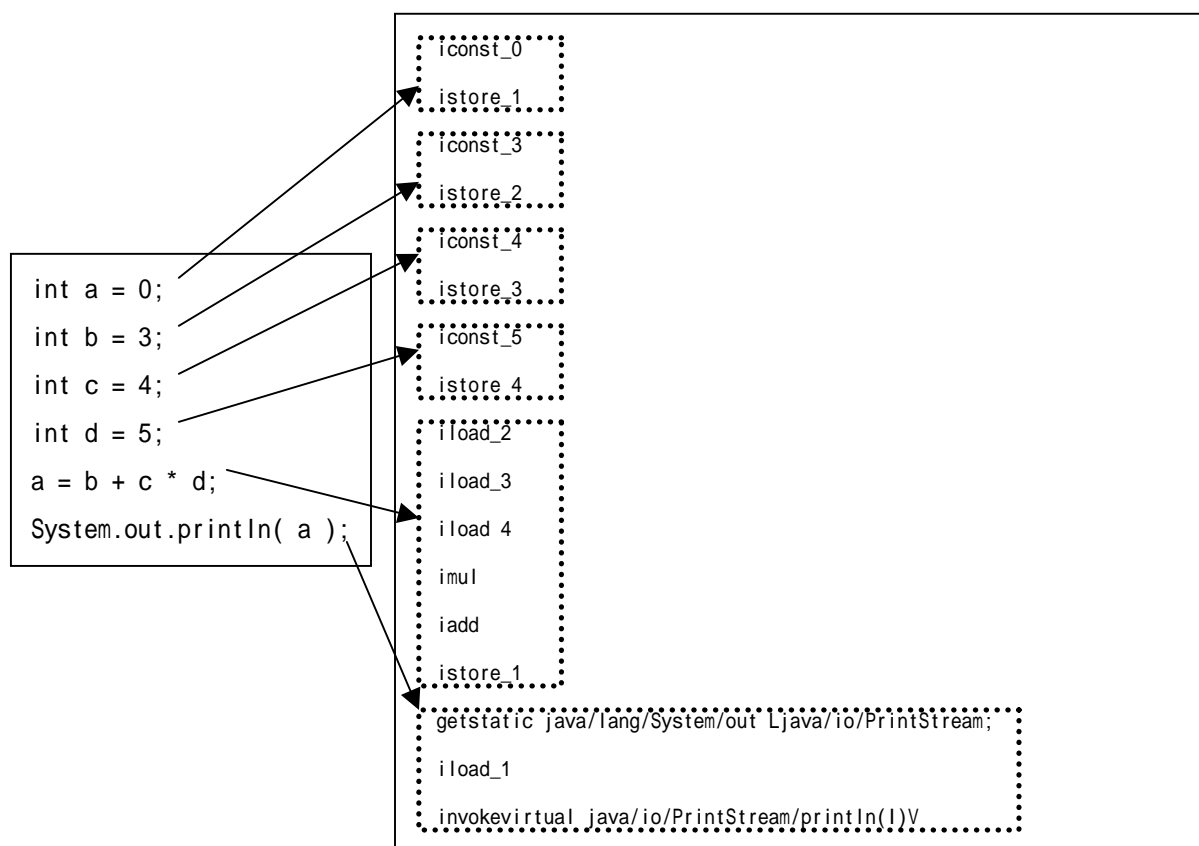


図 3 - 2 Java 言語からバイトコードへの変換

また Java の特徴の一つであるオブジェクト指向は、バイトコードレベルで実現されている。Java 言語で「System.out.println(a)」というメソッド呼び出しを行うところは、以下のようなバイトコード命令になる。まず java.lang.System というクラスの out 属性への参照を、java.io.PrintStream というクラスとして取り出してスタックに積む (getstatic ~)。次に変数 a のデータをデータ領域から取り出してスタックに積む (iload_1)。最後に java.io.PrintStream クラスの println というメソッドを呼び出す (invokevirtual ~)。その際に引数のデータと java.io.PrintStream クラスへの参照をスタックから取り出す。このようにバイトコード命令のレベルでオブジェクトに対する操作が可能になっているのが Java バイトコードの特徴である。

第2項 日本語バイトコードの概要

本研究では、日本語バイトコードを製作した。まずは日本語バイトコードの仕組みについて説明をする。次に何故 Java バイトコードの日本語化なのか、その理由について説明をする。最後にこの日本語化の特徴に関して説明をする。

日本語バイトコードは、Java バイトコードを日本語化したものである。図3-3のように、前述の Java バイトコードを、日本語で記述できるようにした。図3-3の右側のように日本語バイトコードで記述されたソースファイルを、本研究において製作した日本語バイトコードコンパイラでコンパイルすると、図3-3の左側のバイトコードのソースになる。また、日本語バイトコードコンパイラは、Java バイトコードから日本語バイトコードへの変換も可能である。これはバイトコードの意味が両方で1対1の対応をしているためである。その結果、バイトコードと日本語バイトコードは双方向に変換が可能である。

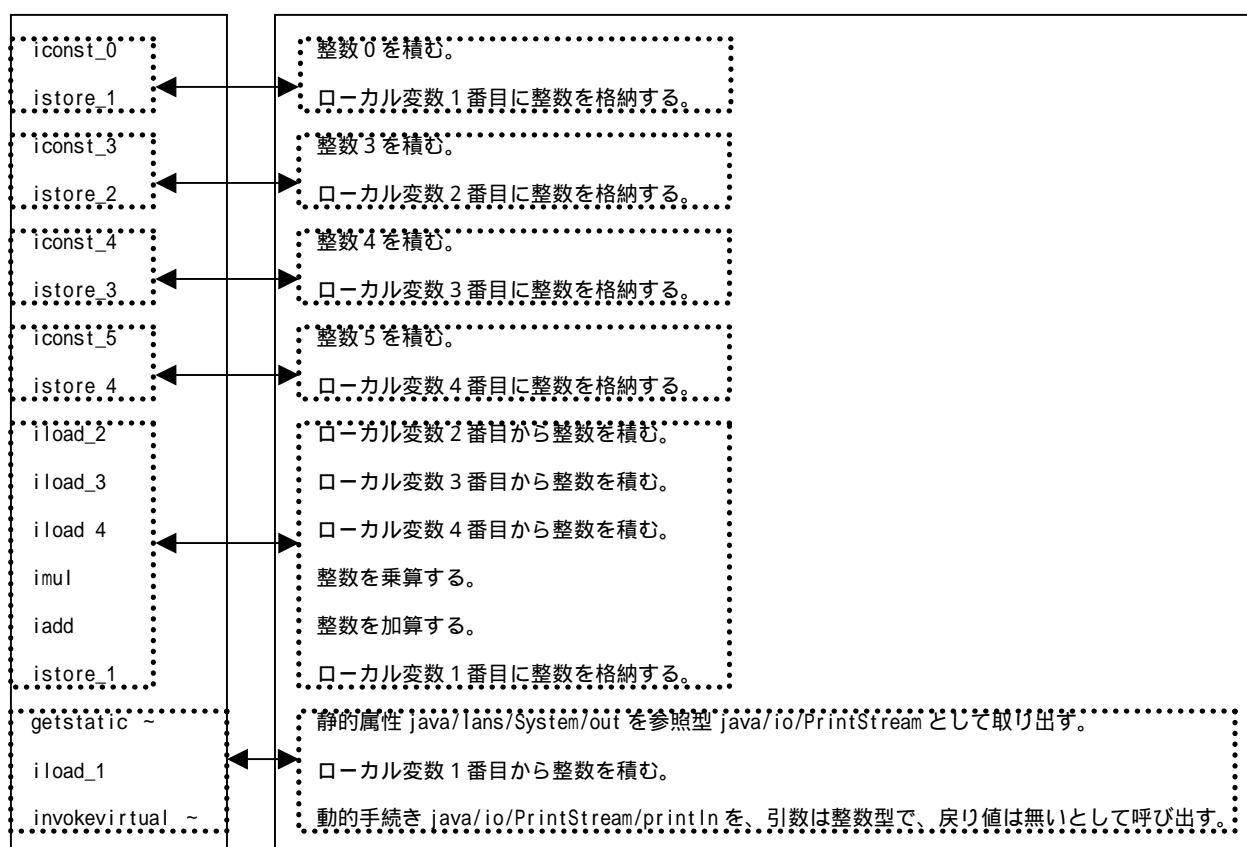


図 3-3 バイトコードから日本語バイトコードへの変換

バイトコードを日本語化した理由の一つ目は Java の持つ移植性を日本語プログラミング言語でも生かしたかった為であり、二つ目は Java バイトコードを日本語化するとオブジェクト指向の実現が容易だからである。

Java の移植性とは、各プラットフォーム上で動作する JavaVM は同じバイトコードには同じ挙動を行なうところから来ている。Java 言語で書かれたプログラムが一旦バイトコードに

変換されてしまえば、そのソフトウェアはどのプラットフォームでも同じように解釈される。日本語バイトコードでもそれはまったく同じである。日本語バイトコードで書かれたプログラムはバイトコードに変換され、それは Windows でも MacOS でも Linux でも同じように動作する。その為日本語バイトコードは Java の持つ移植性をそのまま有している。

二つ目の理由は、バイトコードはスタック操作を直接行う低級言語である一方、オブジェクトの扱いは高級な記述が可能になっているところから来ている。バイトコード上でオブジェクトを介してメソッドを呼び出す際には、その前にオブジェクトの参照と引数をスタックに積む必要はあるものの、基本的には「`invokevirtual`」などの単一命令で行う事ができる。その為 Java バイトコードをそのまま日本語化した日本語バイトコードは、そのままオブジェクト指向言語となる。

第3項 日本語バイトコードの表現設定

日本語バイトコードコンパイラには図3 - 4のような設定ファイルが存在し、それにより日本語表現を自由に変更することが可能になっている。例えば前述の日本語バイトコードプログラムは、下記のように日本語表現が設定されている。

<pre>iload <変数></pre> <p>ローカル変数<変数>番目から整数を積む。</p> <pre>getstatic <変数> <型></pre> <p>静的属性<変数>を<型>として取り出す。</p> <pre>invokevirtual <メソッド名>(<引数>)<戻り値></pre> <p>動的手続き<メソッド名>を、引数は<引数>で、戻り値は<戻り値>として呼び出す。</p>

図 3 - 4 日本語バイトコードの表現設定

ところが仮にこの設定を図3 - 5のように変更すると、プログラムの表現も変わる。

<pre>iload <変数></pre> <p><変数>番目の整数変数を積む。</p> <pre>getstatic <変数> <型></pre> <p>静的な属性<変数>を<型>として取り出して積む。</p> <pre>invokevirtual <メソッド名>(<引数>)<戻り値></pre> <p>引数が<引数>で、戻り値が<戻り値>として手続き<メソッド名>を呼び出す。</p> <hr/> <p>静的な属性 java/lang/System/out を参照型 java/io/PrintStream として取り出して積む。 1 番目の整数変数を積む。 引数が整数型で、戻り値は無いとして手続き java/io/PrintStream/printIn を呼び出す。</p>
--

図 3 - 5 図3 - 4の設定を変更したもの

この設定ファイルは、筆者が「正しい日本語表記」というものは人により変わるものであり言語設計者が表現を押し付けることが無いように配慮した結果である。「iload」という命令を日本語で表現するにはどのような表現が最適なのかは、コンパイラ製作前はもちろん現在でも筆者には分からない。ここで示している表現の設定は、私がこのコンパイラを用いて日本語バイトコードプログラムを行う過程で、どのような表現が高い可読性を持ちかつ記述しやすいかを試行錯誤した結果であり、それは人によって変わる可能性がある。

その為各プログラマに対して「i load」をどのように表現するかは、任せてしまう事にした。これが日本語バイトコードコンパイラの最も特徴的な点である。

そもそもプログラムを実行するコンピュータ側にとって表現は本質的な部分ではない。JavaVM にとっては、実行すべき命令が「i load」という命令であると区別できてその引数が何であるかが理解できれば、バイトコード命令を実行することは可能なのである。であれば「i load」命令であることが分かり引数を取り出すことができれば、命令をどのように記述しても良いはずである。このようにプログラムは「意味」と「表現」の二つに分離することが可能である。筆者は日本語バイトコードコンパイラで表現設定を機能として実装し、この事を実現した。この考え方は後述する日本語プログラミング言語「言霊」の中でも抽象文法と具象文法として生きているので、その中で詳述したい。

第4項 日本語バイトコードの効果

日本語でプログラムを記述すると、その可読性が高くなりプログラムに対する理解が進むと言うことの兆候がこの時点で見られている。日本語バイトコードコンパイラが完成して Java 言語をある程度理解している研究室のメンバーに見せたところ、彼らはあたかもおもちゃを与えられた子供のようにバイトコードの勉強に熱中した。彼らはオブジェクト指向を理解して仕事として Java のアプリケーションを製作する技術を持つほどのレベルは有している。だが Java バイトコードに関しては知識として存在を知っている程度であり、Java バイトコードを用いてプログラムをした経験はおろかバイトコードのニモニックすら見たことは無かった。彼らが日本語バイトコードコンパイラに触れるや様々な Java プログラムを日本語バイトコードに変換したりその逆を行うことで急速に Java バイトコードに関する知識を貪欲に吸収していった。

<pre>for(; ;){ int a=1; }</pre>	<p>Label0:</p> <pre>iconst_1 istore_1</pre>	<p>ラベル0:</p> <p>整数 1 を積む。</p> <p>ローカル変数 1 番目に整数を格納する。</p>
<pre>while(true){ int a=1; }</pre>	<p>Label2:</p> <pre>goto Label0</pre>	<p>ラベル2:</p> <p>ラベル0 にジャンプする。</p>

図 3 - 6 無限ループの for 文と while 文の差異を、日本語バイトコードで検証する

例えば Java 言語において無限ループは for 構文を用いるのと while 構文を用いる 2 通りの表現方法がある。これを日本語バイトコードコンパイラを用いて変換をすると、図 3 - 6 のようにバイトコードレベルでは両者のコードに違いが無いことが分かる。

彼らの話を聞くと、Java バイトコードを用いてはこの比較がやりにくいしそもそも比較して見ようと言う気にすらならないが、日本語バイトコードを用いると違いがあるか無いかが直ちに理解できると述べている。

また for 文と while 文がバイトコードレベルで違いが存在しないことは、図 3 - 7 の例でも分かる。図 3 - 7 のような Java プログラムを日本語バイトコードに変換すると全く同じバイトコードになる。for 文は初期化处理・継続条件・増加処理・繰り返し処理の 4 つから成り立つ。図 3 - 7 では初期化处理に当たる部分を赤字、継続条件に当たる部分を青字、増加処理に当たる部分を紫字、繰り返し処理に当たる部分を黒字で記した。その結果バイトコードレベルでは繰り返し処理がどのような順序で行なわれているかが明らかになる。

この例では特に Java バイトコードと日本語バイトコードの差異が顕著である。「i<10」のバイトコード命令は Java バイトコードで表現すると「if_cmplt」である。この表現が何を意味するかは Java 言語に精通している彼らでも理解はできず、せいぜい比較に関する命令であると言う事くらいしか読み取れ無い。だが日本語バイトコードの「前者が後者より小さければ、ラベル1にジャンプする」を読んで彼らは即座に意味を理解する事ができた。

<pre>for(int i=0 ; i<10 ; i++){ int a=1; }</pre>	<p>整数 0 を積む。</p> <p>ローカル変数 1 番目に整数を格納する。</p> <p>ラベル 0 にジャンプする。</p>
<pre>int i=0; while(i<10){ int a=1; i++; }</pre>	<p>ラベル 1 :</p> <p>整数 1 を積む。</p> <p>ローカル変数 2 番目に整数を格納する。</p> <p>ローカル変数 1 番目を 1 だけ増加させる。</p> <p>ラベル 0 :</p> <p>ローカル変数 1 番目から整数を積む。</p> <p>1 バイト整数 10 を積む。</p> <p>前者が後者より小さければ、ラベル 1 にジャンプする。</p>
<pre> iconst_0 istore_1 goto Label0 Label1: iconst_1 istore_2 inc 1 1 Label0: iload_1 bipush 10 if_icmplt Label1 </pre>	

図 3-7 for 文と while 文の差異を、日本語バイトコードで検証する

日本語バイトコードを使用することで生じた変化の中で重要な事の一つは、日本語の方が理解しやすい事である。そしてもう一つは、日本語で記述してあると理解しようとする姿勢が生じると言うことである。彼らは日本語により記述が理解しやすくなったことで初めてバイトコードと言うものに興味を抱いた。彼らはアルファベットによる表記ではバイトコードと言うものに対する興味はこれほど湧かなかっただろうと述べている。

第3節 日本語プログラミング言語「言霊」

第1項 概要

「言霊」は、日本語バイトコードを拡張する方向で開発された。日本語バイトコードは、日本語で Java アプリケーションを開発する環境を提供はしているが、言語レベルがスタック操作を意識させる低級言語である。図3 - 8右下の日本語バイトコードのように IF 文や代入文をバイトコードで記述すると表現が冗長になってしまう。これを日本語バイトコードはバイトコードの学習には役立つが、一般的なプログラムを書くという用途には向いていないのである。そこで図3 - 8右上のように高級な表現が可能なプログラミング言語とコンパイラを製作した。それが日本語プログラミング言語「言霊」である。

if(x==0) y=1; else y=2;	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。
iload_1 ifne Label_0 iconst_1 istore_2 goto Label_1 Label_0: iconst_2 istore_2 Label_1:	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 3 - 8 日本語バイトコードと日本語プログラム言語「言霊」

「言霊」は、日本語バイトコードを拡張する方向で設計された。例えば日本語バイトコードでは代入を行う際には、代入値をスタックに積んでから代入命令を実行する（整数 1 を積む。ローカル変数 2 番目に整数を格納する。）。「言霊」では代入文として「1 を y に代入する。」と一行で簡潔に表現できるようにした。

その為に「言霊」のソースプログラムをコンパイルするための言霊コンパイラを製作した。言霊コンパイラは「1 を y に代入する」という一文を、「整数 1 を積む。ローカル変数 2 番目に整数を格納する。」という日本語バイトコードの二文に変換する。

Java はプロフェッショナルが使用するプログラム言語であるが、初心者がプログラム学習に使う為に最初に覚える言語としては不向きである。プログラムを初めて学ぶ人は所謂 HelloWorld プログラムを読むことになるが、Java を用いると図3 - 9上のように初心者にとって意味が難解な記述を行う必要がある。ここで用いている public や static などの意

味を解説するのはずっと後かあるいは初心者教育の中では教えないような事柄である。これと同じプログラムを言霊で記述すると図 3 - 9 下のようなになる。言霊の場合は初期の段階で教える必要のない文法事項は極力記述しなくても良いよう設計した。図 3 - 9 下のプログラムの場合はデフォルトで public と static 設定が適用されるし、デフォルトで手続きの引数は無く返り値も無いことになる。

```
public class HelloWorld{
```

```
    public static void main( String args[] ){
```

```
        System.out.println( " Hello,World! " );
```

```
    }
```

```
}
```

```
メインとは {
```

```
    システムが「Hello,World!」を標準出力に出力する。
```

```
}
```

図 3 - 9 HelloWorld プログラムを、Java 言語と言霊で比較する

第2項 「言霊」における文法の扱い

「言霊」は、初心者が抵抗無くプログラム教育を行えるよう設計されたプログラム言語である。初心者教育で最も求められるのは、プログラム表現が自然な日本語になっていることである。初心者は最初に他人が記述したソースプログラムをお手本として見て、それを真似しながらプログラムを書きながら学習を進めていく。お手本のプログラムがアルファベットと記号だらけの Java のような既存のプログラム言語は、この点でプログラム教育に適さない。

だが何が自然な日本語表現なのかを決めることは容易では無い。個人の考え方の違いで「自然な日本語」は変わってくるし、そもそもソフトウェア工学の考え方を自然な日本語で表現できるのかという問題もある。例えばプログラムの基本的な要素として図3-10のような代入文がある。仮にこの代入表現を日本語表現を使って読みやすいようにしたとする。代入する値が1であれば特に問題が起こることは無い。ところが図3-11のように代入データに代入先変数自身が含まれると、一般の人から見ると奇異な日本語表現になる。

a=1;
a に 1 を代入する。

図 3 - 1 0 代入文の日本語表現

a=a+1;
a に a + 1 を代入する。

図 3 - 1 1 代入文の日本語表現

また、そもそも図3-10のような表現がそもそも自然なのかと言う議論も残る。代入と言う表現は硬いから「aを1とする」とか「aに1を入れる」など、様々な表現案が存在する。それぞれの表現は一長一短があり、筆者はどれか選択して「この表現が正しい」とみなして採用することはできなかった。

そこで「言霊」は抽象文法 of 思想を取り入れて意味と表現の分離を行うことで、自然な日本語表現の決定をユーザに委ねる事にした。代入文において本質なのは、代入先と代入値が含まれてかつそれが代入文であると識別できることである。代入文をどのように表現するかは、どこかに定義が存在すればあとはどのように表現しても良いはずである。ユーザが代入文の表現を定義できるような仕組みを用意すれば、各自の考える自然な日本語表記に従って定義をしてプログラムを書ける。

このような意味と表現の分離という考え方は、プログラムの構造を抽象文法と具象文法という考え方で捉える Meyer の文献[14]から来ている。以下では抽象文法と具象文法を解説し、それにより何が可能になり、「言霊」においてどのように実装されているかを解説する。

第3項 抽象文法

プログラミング言語の文法を記述するのに習慣的に使用される手法としてBNF (Backus-Naur Form) 記法がある。BNF記法では生成規則 (production) の集合から構成される文法 (grammar) によって言語の文法を記述する。生成規則は、文や式やメソッドなどの言語要素の形式を記述する。例えば図3-12は、標準的なBNF記法を用いてJava言語の繰り返し文の構文を定義している。

```
<Loop> ::= while( <boolean_expression> )  
           <instruction>
```

図 3-12 BNFを用いた、Javaの繰り返し文の定義

この生成規則は、Java言語における繰り返し文がどのように形成されるかを表している。まずキーワード `while` を書き、括弧“(”、“ ”)で囲まれた中に論理式、そしてその後文を記述する、と言ったことがここから分かる。

`<boolean_expression>`、`<instruction>`といった構文構成要素は `< >` の中に記述する。こうした構文要素は非終端構成要素と呼ばれ、文法全体の中のどれかの生成規則の中で定義される。生成規則の中で定義されない、必ず生成規則の右側に現れる構成要素を終端構成要素と呼ばれる。例えば `while` などのキーワードが終端構成要素である。

BNF記法は文法記述の手法として広く受け入れられているが、この記法にも限界がある。BNFによる仕様が実際に記述しているのはプログラマが見るプログラムの外見であり、意味では無い。BNFの生成規則にはキーワードなど、表面的な構文上の些細であり重要ではない事柄が含まれているのである。例えば前述の生成規則はJava言語における繰り返し文だったが、VisualBasicにおける繰り返し文の生成規則を記述すると以下のようになる。

```
<Loop> ::= Do While <boolean_expression>  
           <instruction>  
           Loop
```

図 3-13 BNFを用いた、VisualBasicの繰り返し文の定義

この二つの繰り返し文は表面的な記述方法が異なるだけで、繰り返し文の構造は全く同じものである。すなわち繰り返し文は`<boolean_expression>`と`<instruction>`の二つを要素として持っていて、それをどのように表現するかが異なるだけである。

抽象文法では表現上の詳細を省いて、言語の各構文要素の構造を記述する。例えば繰り返し文は `boolean_expression` と `instruction` の二つの要素をもっているということのみを記述する。一方でその抽象文法で定義された構文要素をどのように記述するかは具象文法で定義される。

「言霊」は、筆者が抽象文法を定義し、具象文法はユーザが自由に変更できるような仕様になっている。

抽象文法により定義されるプログラムの本質的な意味は、JavaVM のバイトコード命令を吐き出す源泉になる。変数 a に 1 を代入する代入文は、「iconst_1 istore_1」というバイトコード命令に変換される。逆にいえばバイトコード命令に変換できないものをプログラムの意味として記述してもしようがない。であるから筆者が JavaVM のバイトコード命令の仕様を元にして抽象文法を定義した。

抽象文法は Meyer の文献[14]では \triangleq という記号を用いて定義される。例えば先ほどの繰り返し文の抽象文法は以下の図 3 - 1 4 のように定義される。

Loop \triangleq body: <i>Instruction</i> ; test: <i>Expression</i>
--

図 3 - 1 4 繰り返し文の抽象文法

Loop (繰り返し文) は、body という名の Instruction と、test という名の Expression から成り立つという意味である。また、Expression (データ表現) は以下の図 3 - 1 5 のように定義される。

Expression \triangleq <i>Constant</i> <i>Variable</i> <i>Binary</i>

図 3 - 1 5 データ表現の抽象文法

Expression (データ表現) は、定数 (Constant) か変数 (Variable) か 2 項式 (Binary) のいずれかであると言う意味である。Meyer の文献[14]ではこうした規則を並べることで抽象文法を表現している。

だが筆者は Unified Modeling Language (統一モデリング言語。以下、UML) の表記法を用いた。UML を用いると抽象文法の構造をグラフィカルに伝えることができるし、筆者自身が Java を用いたソフトウェア開発で UML を用いたモデリングの経験がある為、UML を用いた。

UML を用いて先の繰り返し文と同じ意味のものを Rational 社の UML モデリングツール Rational Rose[15]を用いて表記すると、図 3 - 1 6 のようになる。UML における四角はクラスを表し、それを三分割した一番上にクラスの名前が記述される。図 3 - 1 6 の四角は Loop というクラスを表し、これは抽象文法の構文要素を表している。三分割した真ん中には、クラスの属性が記述される。Loop クラスは二つの属性を持っていることを表している。属性の詳細は「<名前>:<型>」と記述する。属性の一つの名前が body であり、その型が Instruction であることを表している。まとめると、Loop は body と test という名の属性を持ち、それぞれの型が Instruction と Expression であることを示している。

同様に Expression を UML で記述すると図 3 - 1 7 のようになる。Constant, Variable, Binary の三つのクラスから Expression に向いた矢印が記述されている。図 3 - 1 7 のような矢印は UML では継承関係を表している。三つのクラスは Expression

を継承しているということである。

継承関係とは、抽象 - 具体の関係にあるということである。Expression (データ表現) を具体化すると Constant (定数), Variable (変数), Binary (2項式) のどれかを指し、Constant (定数) を抽象化すると Expression (データ表現) になるということである。

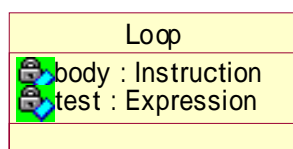


図 3 - 1 6 UML を用いて表現した繰り返し文の抽象文法

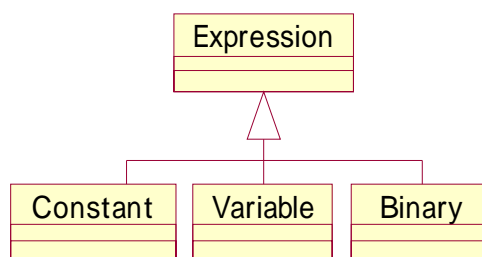


図 3 - 1 7 UML を用いて表現した値の抽象文法

「言霊」における抽象文法の構造を解説する。図 3 - 1 8 はクラス宣言文や手続き宣言文などを含めた部分の抽象文法をUMLで図示したものである。クラス宣言文には、クラス名と継承クラスと実装クラスとパッケージを持っている。また菱形の矢印があるが、これも属性をあらわす矢印である。だからクラス宣言文はさらに属性宣言文と手続き宣言文を持っていることを表している。手続き宣言文は実行文を多数持っていて、それは宣言文や代入文などのプログラムの要素を表す。

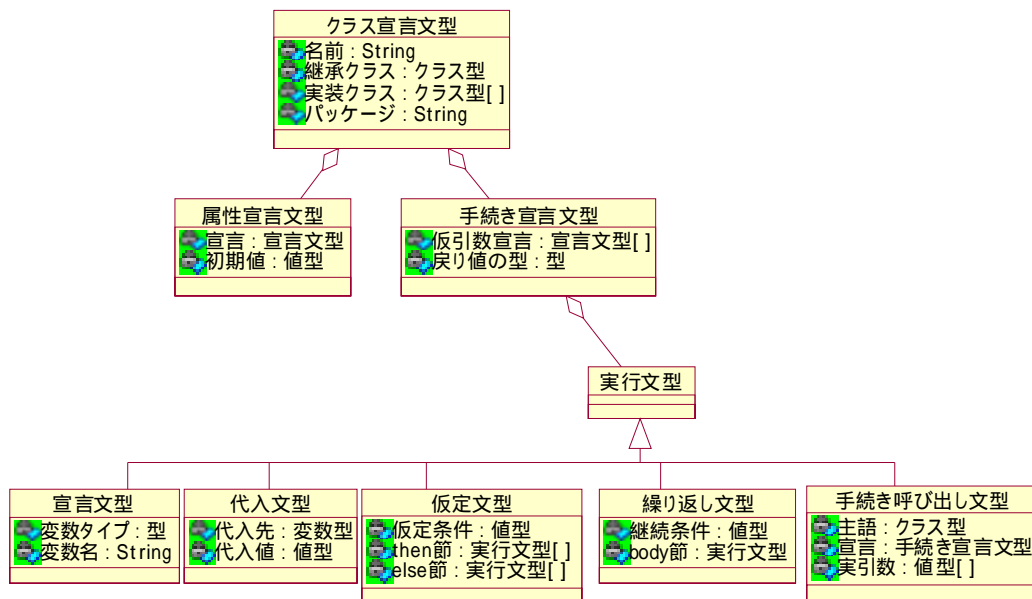


図 3 - 1 8 言霊における抽象文法

図3-19は「言霊」における値の表現を図示したものである。値は、定値か変数か2項式か手続き呼び出し文のどれかを表す。定値はPrimitive値か参照値か文字列参照かnull参照のどれかを表す。Primitive値とはBoolean値か数値を表し、数値は整数値か浮動小数点数値を表す。整数値はInt値かByte値かShort値かLong値かChar値を表し、浮動小数点数値はFloat値かDouble値を表す。その他の値も同様に読み取れる。

Boolean値などアルファベットで表現しているのは、それがJavaVMにおける用語を指すからである。「言霊」の抽象構文はJavaVMの実装に強く影響されるので、値表現の最も具体的な部分にはJavaVMの用語が登場する。もちろんBooleanは真偽値、Intは整数値、Shortが短整数でLongは長整数、Floatが実数、Doubleが倍実数、Charが文字など日本語で表現することも可能だが、ここではJavaVMの実装を強く意識させる用語を用いた。

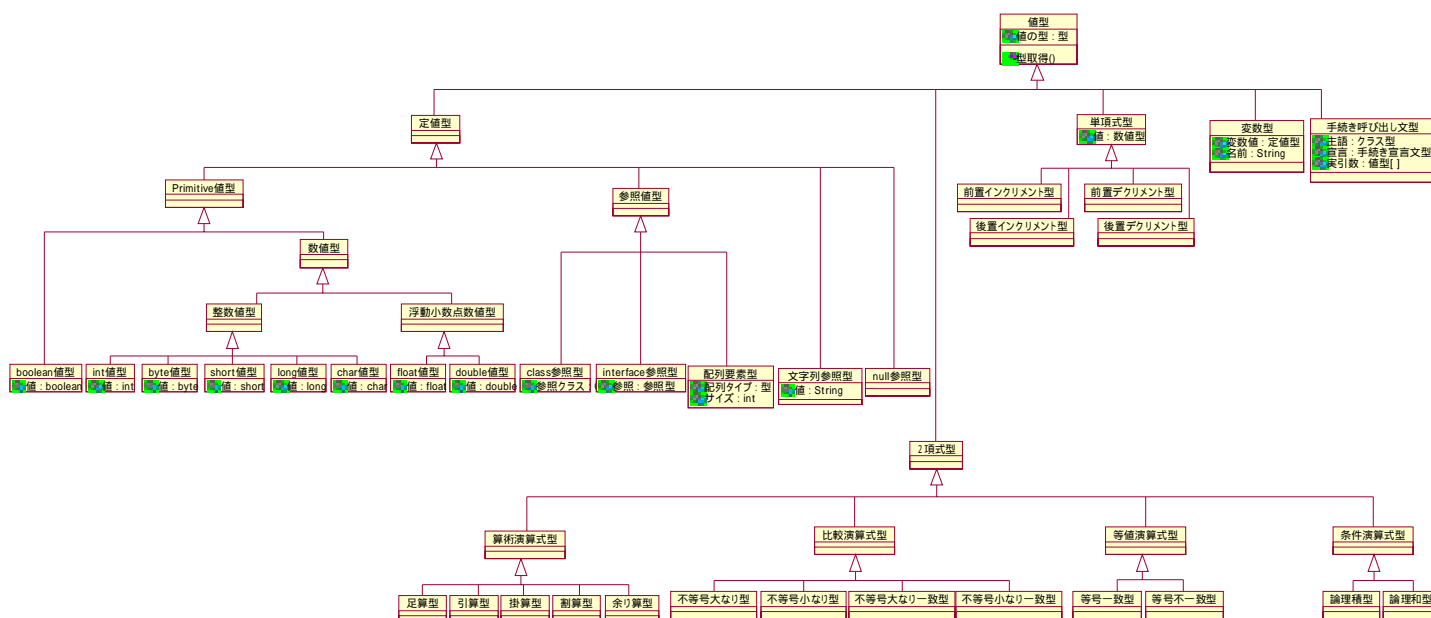


図 3 - 19 言霊における値の抽象文法

「言霊」における型の構造は、以下のような図 3 - 2 0 で表現される。

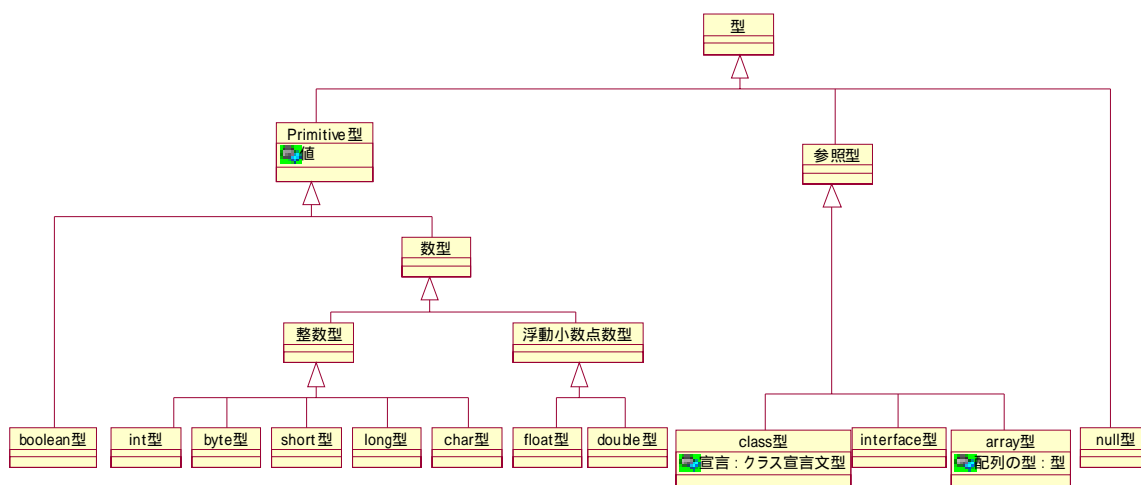


図 3 - 2 0 言霊における型の抽象文法

「言霊」は以上のような抽象文法で表される構造を持った言語である。プログラムをどのように表現しようが、この抽象文法で表現される構造を持っていれば JavaVM で動作可能なバイトコードを作成することができる。

次に説明する具象構文により表現形式を定義するが、どのような表現を用いようが最終的にはこの抽象文法の構造に変換される。だから表現が本質ではないのでどのような表現を行おうが問題では無い。日本語や英語や韓国語や中国語などの自然言語を用いても、最終的にこの構造へ変換が可能であれば実行可能なプログラムを作成できる。

第4項 具象文法

具象文法とは、抽象文法の表現に関するルールである。設定は図3 - 21のようになる。

図3 - 21では代入文の表現に関するルールを定義している。こう定義すると例えば代入文は例えば「1をaに代入する」「aに1を代入する」と表現することが可能である。

<代入値>を<代入先>に代入する。 <代入値>に<代入先>を代入する。
--

図 3 - 2 1 言葉における代入文の具象文法

これは日本語で無くても全く問題無い。例えばJava言語のような表現にするのであれば以下のように定義する。こう定義するとプログラムは「a=1」と表現することになる。

<代入先>=<代入値>

また、英語で表現することも可能である。英語での代入文は以下のような定義になるだろうか。こう定義するとプログラムは「substitute 1 for a」と表現することになる。

substitute <代入値> for <代入先>

具象構文の設定を使えば様々な表現でプログラムを記述することが可能になる。この例のように英語で表現することも可能であるし、韓国語・中国語・フランス語・ドイツ語などの自然言語でもそれぞれのネイティブが設定を行なえば各国言語によるプログラムが可能である。日本語でプログラムができるというのは、この機能の一部でしか無い。

第5項 具象文法の実装の仕組み

「言霊」のソースプログラムをコンパイルする仕組みは、既存のコンパイラの仕組みとは異なっている。既存のプログラム言語のコンパイラは決定性有限オートマトン (Deterministic Finite Automaton: 以下ではDFAと略記する) の考え方によって実装されているのに対して、「言霊」は非決定性有限オートマトン (Non-deterministic Finite Automaton: 以下ではNFAと略記する) を用いている。

そこで本項ではまずDFAを用いたコンパイラの仕組みを説明し、その問題点を指摘し、そしてNFAを用いた言霊コンパイラの仕組みを説明する。

・決定性オートマトンを用いた解析の仕組み

有限オートマトンとは、有限個の内部状態を持ち、与えられた記号列を読みながら状態遷移し、その記号列がある言語の文であるかどうかを判定するものである。これを用いてプログラム言語がコンパイル可能なものかを判定し、字句に分割する。

実際のプログラム言語の解析の例を取り上げ、DFAから字句解析プログラムを作成する過程を紹介する。取り上げる例はC言語などで使用される浮動小数点数の解析プログラムである。浮動小数点数は次の構文規則によって定義されるとする。

浮動小数点数	小数点数 (指数部) (数字列 指数部)
小数点数	(数字列) . 数字列 数字列 .
指数部	E (符号) 数字列
符号	+ -
数字列	数字 数字列 数字

この構文規則から正規表現を求めると、以下のようになる。なお、表現を簡単にするために「数字」という表現を「d」と書く。

浮動小数点数 $((| dd^*) . dd^* | dd^* .) (| E (| + | -) dd^*) | dd^* E (| + | -) dd^*$

この正規表現からNFAを求めると、以下の図3-22のようになる。

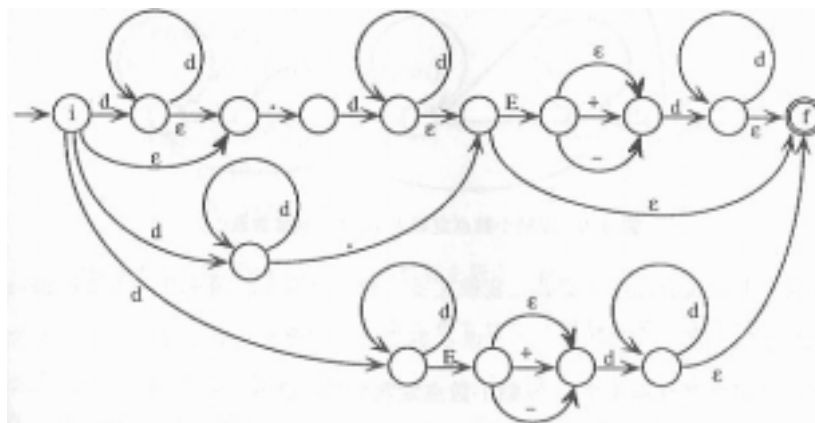


図 3 - 2 2 不動小数点数のNFA

さらにこのNFAをDFAに変換すると、図3-23のようになる。

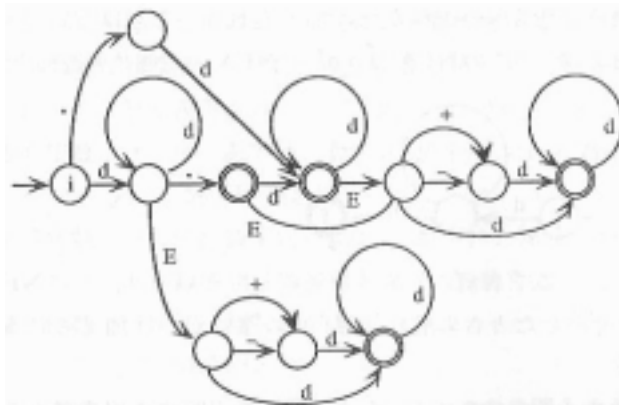


図 3 - 2 3 不動小数点数のNFA

このDFAの状態数を最小にすると、図3-24のようになる。

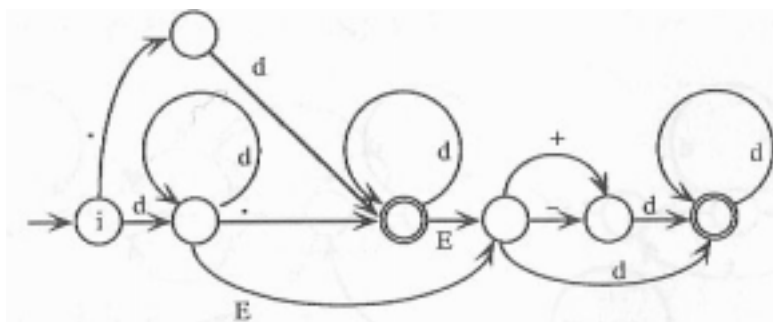


図 3 - 2 4 不動小数点数のNFA (状態数最小)

このDFAを元に読み取りプログラムを作成すると、図3-25のようになる。

```

/* x.yE±z の形の浮動少数定数を読み込む */
int b;          /* xy を整数と見た数 */
int e;          /* y の桁数 */
int i;          /* z の値 */
char sign;      /* 指数の符号 */
. . . . .
state_i: b=0; e=0; i=0;
    if( ch == ' . ' ){
        ch = nextChar();
        if( charClassT[ch] != digit ) error();
        b = ch - ' 0 ' ; e = 1;
    }else if( charClassT[ch] == digit ){
        do{
            b = 10 * b + ch - ' 0 ' ;
            ch = nextChar();
        }while( charClassT[ch] == digit );
        if( ch == ' E ' ) goto exp;
        if( ch != ' . ' ) error();
    }else error();
    ch = nextChar();
    while( charClassT[ch] == digit ){
        b = 10 * b + ch - ' 0 ' ;
        e++; ch = nextChar();
    }
    if( ch != ' E ' ) goto calc;
exp: sign = ' + ' ; ch = nextChar();
    if( ch == ' + ' || ch == ' - ' ){
        sign = ch; ch = nextChar();
    }
    while( charClassT[ch] == digit ){
        i = 10 * i + ch - ' 0 ' ;
        ch = nextChar();
    }
calc: /* ここで浮動小数点数の計算を行う */

```

図 3 - 2 5 浮動小数点数の読み込みプログラム

・ 決定性オートマトンを用いた解析の限界

既存のプログラム言語は、これまで紹介した決定性有限オートマトンを用いて字句解析を行った後に構文解析を行う手法を用いている。この手法を用いるためには、プログラム言語が字句解析によりトークン化しやすいような文法になっていることが前提条件である。

例えば以下の図 3 - 26 左の C 言語プログラムを字句解析する場合は、図 3 - 26 右のようにトークン化する。これを可能としているのは、制限された文法仕様である。例えばこの例では「abc」「e3」という変数が用いられているが、文法では変数に使用できるのは英数字のみである。それも変数の先頭は必ず英文字である必要がある。変数名に「=」や「+」などの演算子を用いる事は出来ない。

abc=e3*2.56+abc/e3;	abc	=	e3	*	2.56	+	abc	/	e3	;
---------------------	-----	---	----	---	------	---	-----	---	----	---

図 3 - 26 字句解析

決定性有限オートマトンを用いるのに都合が良い文法を定義すると、字句解析プログラムを作成するのが容易になる。この図 3 - 26 のプログラムを解析するための NFA を作成し、そこから DFA に変換した後に解析プログラムを作成する。そして字句解析によりトークン化を行った後に構文解析の処理を行う。

既存のプログラム言語が字句解析 構文解析というプロセスを経てコンパイルを行うのは、英語という言語の性質が影響していると筆者は感じている。英語は表記する際には単語と単語の間には必ず空白が存在する分かち書き言語であり、単語と単語の区切りが明白で token が存在するのが英語の前提である。プログラム言語の開発をリードしたのは、英語を母国語としているアメリカ人である。彼らはプログラム言語の解析のプロセスを考えた際に、構文解析という意味解釈の前に字句解析という単語区切りを行うべきだと判断したのは、分かち書き言語である英語の影響が強いのではないだろうか。また言語に token が存在すると考えれば、プログラムの解析を token を切り出す字句解析のプロセスと、token の列から意味を取り出す構文解析の二つから構成されると考えるのは、彼らにとって自然である。

ところが日本語を解析する「言霊」は、こうしたプロセスには馴染まない。英語のような分かち書き言語でもなく、token が存在するという前提がない日本語に、字句解析 構文解析というプロセスは適さない。

例えば先の例をそのまま日本語として記述し、それを解析するプロセスを図 3 - 27 に示す。ここで前提とする具象文法は、以下の 4 点である。もちろん具象文法は表面的な表現に過ぎずこの表現に問題があれば変更は可能であるが、ここではとりあえずこのように定義する。

- ・ 代入文は「～を～に代入する」と表現する。前者が代入値で後者に代入先が入る。
- ・ 足算式は「～と～を足したもの」と表現する。両者とも値が入る。

- ・ 掛算式は「～に～を掛けたもの」と表現する。両者とも値が入る。
- ・ 割算式は「～を～で割ったもの」と表現する。両者とも値が入る。

e3 に 2.56 を掛けたものと abc を e3 で割ったものを足したものを abc に代入する
e3 に 2.56 を掛けたものと abc を e3 で割ったものを足したものを [を] [abc] に代入する
e3 に 2.56 を掛けたもの と [abc を e3 で割ったもの] を足したもの
[e3] に [2.56] を掛けたもの [abc] を [e3] で割ったもの

図 3 - 2 7 日本語の解析の流れ

このプログラムの解析を行う際に、既存のプログラムのように字句解析と構文解析を分けて考える事は意味がない。英語を基礎にしたプログラムの場合は字句解析と構文解析を分離して token 切り出しと意味解析を別々の処理にしたほうがプロセスが単純になるが、日本語の場合はそもそも token というものが存在しない。

token を切り出す字句解析という処理は、意味解析とは全く無関係に token を取り出す処理である。英語は分かち書き言語であるから「pen what apple is go to Japan」などのように全く意味の分からない英文でも token を切り出す事は可能である。この場合は空白に区切られた英文字をそのまま取り出していけば token になる。C や Java のようなプログラム言語でも同じで「abc*3.4=(100,xyz)+3」という意味のない文でも、token 切り出しは原則的に可能である。この場合は「[abc] * [3.4] = ([100] , [xyz]) + [3]」のように token が分離する。もちろんこの文は意味がないので構文解析プロセスの中でエラーが出るのだが、字句解析プロセスの中では問題なく処理が進む。

日本語の場合は、意味解析と無関係に token を切り出す事は不可能だ。仮に日本語から token を取り出せるとしたら、それは意味を解釈できた時なのである。

先の図 3 - 2 7 を見てみよう。代入文は「～を～に代入する」という形をしている。文からこのような形をしているものを取り出すとすると、以下の 4 例が取り出せる。

- ・ e3 に 2.56 を掛けたものと abc を e3 で割ったものを足したものを [を] [abc] に代入する
- ・ e3 に 2.56 を掛けたものと abc を e3 で割ったもの [を] [足したものを abc] に代入する
- ・ e3 に 2.56 を掛けたものと abc [を] [e3 で割ったものを足したものを abc] に代入する
- ・ e3 に 2.56 [を] [掛けたものと abc を e3 で割ったものを足したものを abc] に代入する

この 4 例の中でどれが正しい区切り方なのかは、ここでは決定できない。どれが正しいかは、先の図 3 - 2 7 のようにさらに解析を続けて最終的に一番上の例が正しいと判断する。

日本人が具象文法の定義を理解したうえでこの 4 例を見た時に、正しい区切り方をしているのが一番上の例であり、それ以外の例が誤りである事が理解できるはずだ。これは、人間が以下のプロセスを踏んで解析している為であろう。

1. 代入文の文形とこの文を当てはめてみて、上記の4例を頭の中で作る。
2. 仮に上から二つ目の例が正しいのではないかとあたりをつけて解析を試みる。
3. e3 に 2.56 を掛けたものと abc を e3 で割ったものというのが代入文のその場所にふさわしいものかどうか、つまり代入値としてふさわしい表現なのかを考える。
4. 足したものを abcというのが代入文のその代入文のその場所にふさわしいものかどうか、つまり代入先としてふさわしい表現なのかを考える。
5. どちらもふさわしい表現ではないので、上から二つ目の例は誤りだと判断する。
6. その他の例を考える。(以下、2~5繰り返し)

(実際には値の表現のための文形(足算・掛算・割算の文形)があり、それらにふさわしい表現なのかの判定を行っている)

このように日本語の解析を行う場合は、まず意味のある文形を思い浮かべそれを元に仮に分割する。その分割が正しいのかはさらに部分を解析する必要があるが、その解析も意味のある文形を思い浮かべながら行っていく。日本語の解析をする場合は、意味のある文形がまずあり、それを元に単語を分割していく。

逆に言うと単語を分割できたと言う事は、その意味が確定しているという事である。先の図3-27の一番下の文では全ての単語が区切られている。これは文章の意味を理解できて初めて分割が可能になっているのである。

以上のことから、筆者は既存のプログラム言語のような字句解析 構文解析というプロセスは採用しなかった。このプロセスは、プログラムが意味解析とは無関係に token を切り出す事が出来ると言う前提があるからこそ成り立つプロセスであり、日本語を解析する場合にはこれは不適當である。日本語から token を切り出す場合は、意味の解析を抜きに考える事は出来ない。日本語の解析では意味の解析を行いながら token を切り出すという方法が適切である。

既存のプログラム言語は決定性オートマトン(DFA)の考え方をういて字句解析を行い、その後に構文解析を行うというプロセスだった。だが筆者は意味の解析を行いながら token を切り出す為に、非決定性オートマトン(NFA)の考え方をういて解析を行った。

・非決定性オートマトンの考え方

決定性有限オートマトン (D F A) では、ある状態においてある入力が行われた場合、次に遷移する状態は一意に決まっていた。だが非決定性有限オートマトン (N F A) の場合は、ある状態において入力が行われた場合、次に遷移する状態の数がいくつでも良いというモデルである。

N F A は、膨大な数の機械が同時に動いているという、一種の理想化された並列計算機のモデルとして考える事が出来る。ある非決定性有限オートマトン M が入力を受け取ったとき、次に取りうる状態が m 個、 q_1, q_2, \dots, q_m であったとする。このとき M は $m-1$ 個の自分と同じコピーを作り、 m 個の非決定性有限オートマトンはそれぞれ状態 q_1, q_2, \dots, q_m で計算を続行する。

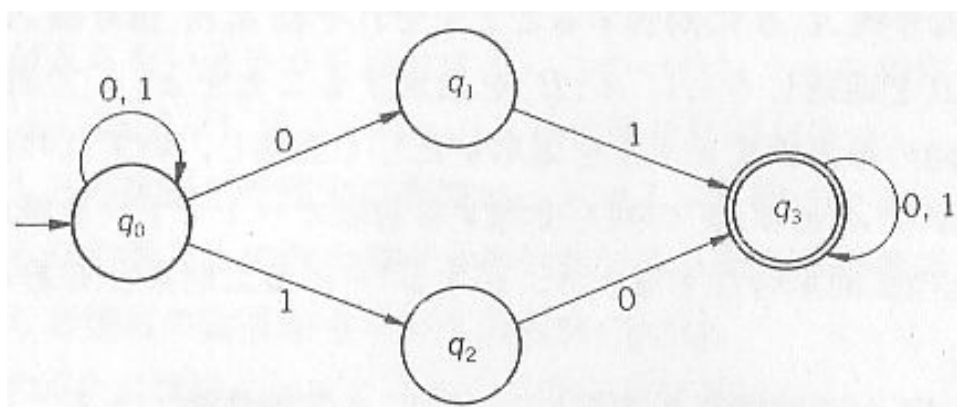


図 3 - 2 8 非決定性オートマトンの例

例を挙げる。図 3 - 2 8 はある N F A の状態遷移図である。 q_0 で 0 が入力されると q_0 と q_1 の両方に向かう矢印がある。その場合は q_0 での 0 入力に対し、N F A は自分のコピーを作り、片方は q_0 、もう片方は q_1 にいるような 2 個の N F A で計算を続けるのである。

このように N F A は次々に増殖を繰り返し、そのうちのどれか一つが受理状態に達したときに入力を受理するのである。

仮に入力 0 0 0 1 が与えられたときの M の様相の変化を、図 3 - 2 9 に示す。

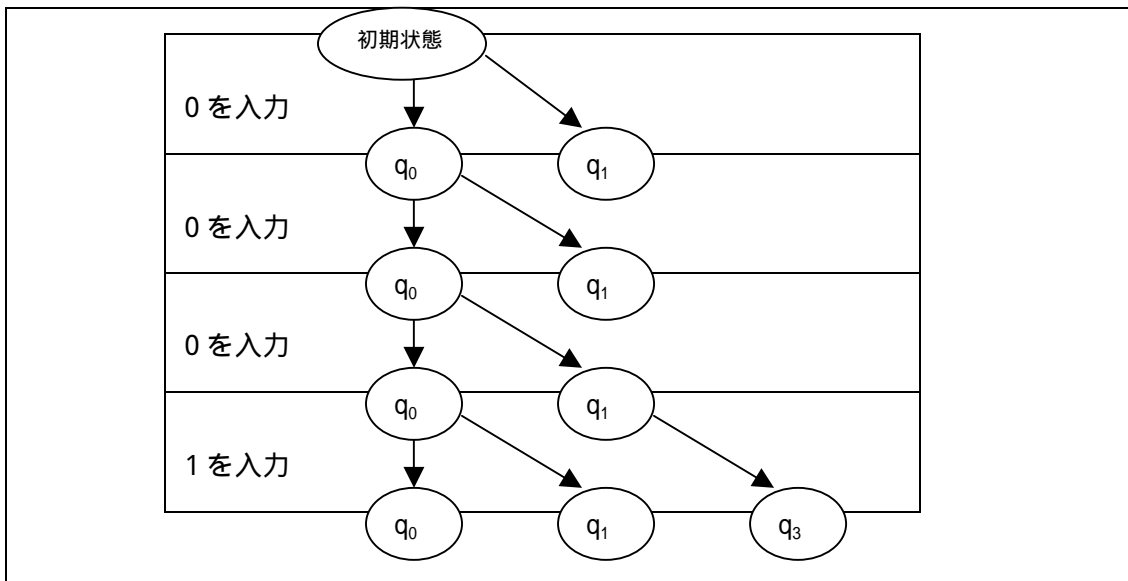


図 3 - 2 9 非決定性オートマトンの様相の変化

最初に 0 が入力されると、オートマトンは初期状態から q_0 と q_1 の二つの状態に遷移する可能性がある。その為、それぞれの状態に遷移したオートマトンが作成される。続いて 0 が入力される。 q_1 に遷移したオートマトンは、 q_1 から 0 で遷移する先を持たない為、このオートマトンは消滅する。だが q_0 に遷移したオートマトンは q_0 と q_1 へ遷移することが出来るため、それぞれのオートマトンが作成される。

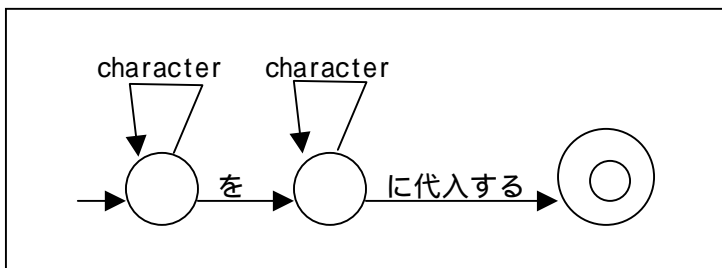
こうしたオートマトンの生成と消滅を繰り返すと、最後の入力文字 0 を受け取った時点で q_0, q_1, q_3 の三つの状態を持ったオートマトンが残っている。先のオートマトンの最終状態は q_3 なので、このオートマトンが受理状態となる。結果、このオートマトンが辿った道のり「 $q_0 \ q_0 \ q_1 \ q_3$ 」が、入力文字列 0 0 0 1 を受理するための正しい道のりである事が分かる。

・非決定性オートマトンを用いたコンパイラの実装

「言霊」における解析は、非決定性オートマトンの考え方をを用いている。先に紹介した、以下のプログラムを解析するプロセスを例に挙げる。

e3 に 2.56 を掛けたものと abc を e3 で割ったものを足したものを abc に代入する

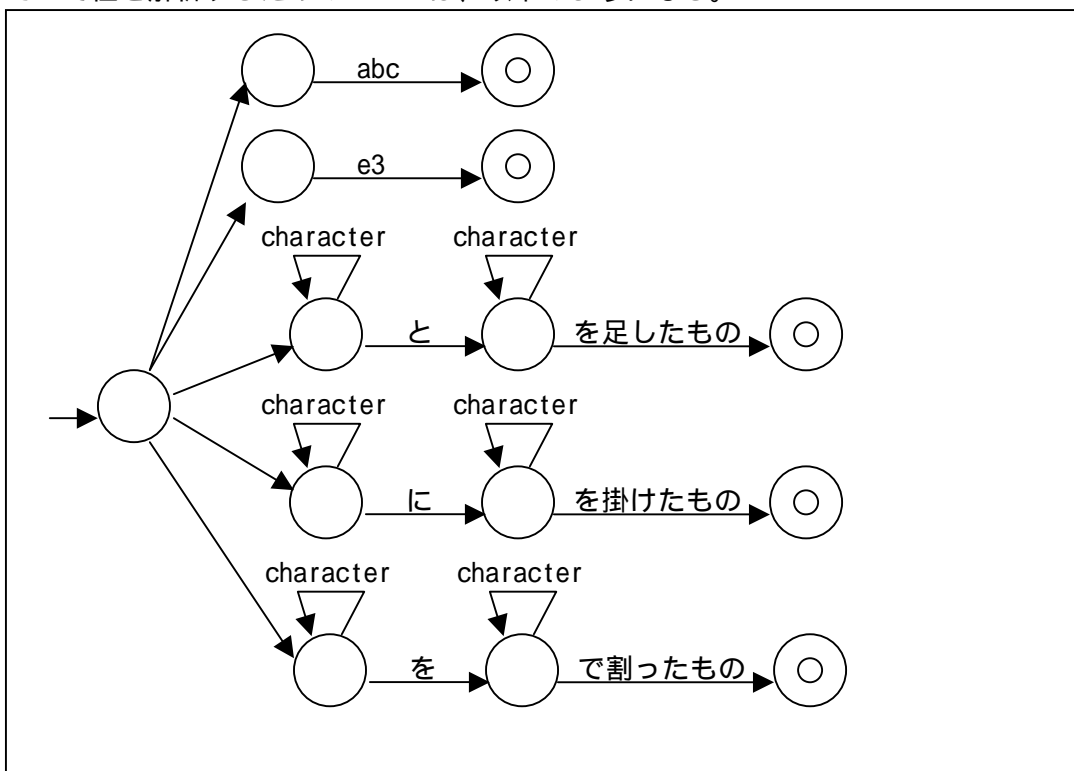
コンパイラはまず代入文であるかどうかを調べるために、代入文の文形を現す非決定性オートマトンを作成する。代入文の文形は「～を～に代入する」なので、NFAの形は以下のようなになる。



ただし character で示された矢印は、全ての文字を現している。このオートマトンを用いて先の文を解析すると、以下の四つの解析結果が得られる。

- e3 に 2.56 を 掛けたものと abc を e3 で割ったものを足したものを abc に代入する
- e3 に 2.56 を掛けたものと abc を e3 で割ったものを足したものを abc に代入する
- e3 に 2.56 を掛けたものと abc を e3 で割ったもの を 足したものを abc に代入する
- e3 に 2.56 を掛けたものと abc を e3 で割ったものを足したもの を abc に代入する

この解析結果を用いてさらに解析を進める。代入文の前半に来るべきものは代入値である。値は、変数(ここでは abc, e3 のどちらか)か加算式・掛算式・割算式のどれかである。そこで値を解析するためのNFAは、以下のようなになる。



このオートマトンで解析すると、 の「e3に2.56」は受理されない為、値ではない事が分かる。 、 においても同じく受理されないの、値ではない。その結果 、 、 はそれぞれ代入文ではない事が判明する。だが だけは「e3に2.56を掛けたものとabcをe3で割ったものを足したもの」は受理される。これは「e3に2.56を掛けたものとabcをe3で割ったものを足したもの」と解析された。そこでこの解析結果を用いてさらに解析を重ねる。足算式の「~と~を足したもの」において、~に入るのは値である。そこで「e3に2.56を掛けたもの」と「abcをe3で割ったもの」を先ほどのオートマトンで解析する。その結果「e3に2.56を掛けたもの」「abcをe3で割ったもの」と解析できる。この結果を元にさらに解析すると「e3」「abc」「2.56」も値である事が分かる。その結果この文の解析は終了し、代入文であることが分かると共に構文木を作成する事が可能になる。

実際のプログラムの解析の場合は、もう少し複雑な処理が必要になる。例えば文は代入文以外にも宣言文・仮定文・繰り返し文・手続き呼び出し文があるし、値の表現もより複雑になる。だが基本的にはここで解説した手法の拡張に過ぎない。

「言霊」のコンパイラは具象文法の設定によりNFAを動的に作成し、解析する。例えば先ほどの代入文の解析では「~を~に代入する」という具象文法を用い、それに合わせたNFAを作成した。だが「~を~に入れる」という具象文法を用いる場合は、これに合わせたNFAを作成すればいいだけである。コンパイラはあり得るどのような表現であろうと、それに合わせたNFAを作成してくれる。その結果コンパイラは代入文に必要な属性である代入値と代入先を抽出し、抽象文法に必要な情報を構築して、最終的にバイトコードを出力するのである。

第6項 低級表現と高級表現の混在

「言霊」の特色のひとつに、低級な日本語バイトコード表現と、高級な表現の混在を許しているという点がある。例えば図3-30のように言霊で記述したプログラムがある。この中で代入文に相当する部分を赤字で示す。

```
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 3-30 言霊による高級表現

図3-30のプログラムは、コンパイラによって図3-31のような日本語バイトコードに変換される。日本語バイトコードの中で、言霊における代入文に相当する部分を赤字で示す。

```
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 3-31 日本語バイトコードによる低級表現

だが言霊では高級な表現の中に日本語バイトコード表現を混ぜることが可能である。代入文に関する部分だけ日本語バイトコードで記述した例が図3-32である。同じく、代入文に相当する部分を赤字で示す。

```
xが0ならば{  
  整数1を積む。  
  ローカル変数2番目に整数を格納する。  
}  
そうでなければ{  
  整数2を積む。  
  ローカル変数2番目に整数を格納する。  
}
```

図 3-32 低級と高級を混ぜた表現

このように言霊と日本語バイトコードの混在が可能になると、熟練プログラマは自分で最適化されたプログラムを記述することが可能になる。例えば図 3 - 3 2 のプログラムを良く見ると重複している部分がある。これを修正すると図 3 - 3 3 のようなプログラムになり、プログラムサイズが小さくなる。こうした最適化は、高級な表現しか記述できない Java 言語では不可能である。

```
x が 0 ならば {
    整数 1 を積む。
} そうでなければ {
    整数 2 を積む。
}
ローカル変数 2 番目に整数を格納する。
```

図 3 - 3 3 最適化されたプログラム

低級な日本語バイトコードの表現を言霊の中で記述できることによるメリットは、2つある。1つ目はプログラムの最適化を行うことで、プログラマの力量次第で JavaVM の性能を引き出すようなプログラムを記述することが可能である。これは主に JavaVM とバイトコードの仕組みに精通した熟練プログラマが対象である。

もう1つが JavaVM とバイトコードの学習に役立てることが可能な点である。言霊コンパイラの機能には、言霊で記述されたプログラムを日本語バイトコードに変換する機能がある。その為、言霊の高級な表現がどのような低級表現に変化するのかが確かめることができる。この機能を生かして日本語バイトコードを読んだ上で、自分で言霊の一部を日本語バイトコードに書き換えながら勉強することが可能である。日本語バイトコードを読む機能と、書いて実行する機能の2つがあれば学習が可能になる。これは主に言霊を使って普通にプログラムができるようになった中級プログラマが対象である。

第4章 「言霊」を用いた実験授業

第1節 授業概要

日本語プログラム言語「言霊」を用いてタートルグラフィックのプログラムができるような環境を整えて、大岩元教授の授業「対話システム論」の中で3回に渡り言霊を用いた授業を行い、レポートを提出させて反応を調べた。その結果を報告する。

授業は2002年10月31日、11月28日、12月5日の三回にわたり行われた。第一回目は「言霊」の背景となるプログラム言語の歴史や他言語との比較を行い、第二回目は「言霊」を用いてタートルグラフィックスを使ったプログラムを記述できるよう、細かい文法を解説した。第三回目は演習で、課題を説明しただけで講義は行わず「言霊」のインストール作業やアドバイスなどを行った。

第2節 第一回目の授業

この授業では主にプログラム言語の歴史を講義し、日本語プログラム言語の位置づけを示した。その中で大昔の機械語 (SpeedCoding)、FORTRAN、FORTH、Mind などの言語や仕組みを紹介した。主な内容は第2章に含まれるので、ここでは割愛する。

第3節 第二回目の授業

この授業では言霊を使って具体的にタートルグラフィックスを用いてプログラムを作成する方法について講義した。プログラムの順次実行の性質や変数の概念、宣言文・代入文・繰り返し文・仮定文に関して説明をした。この内容は学部1年生がプログラミング入門Bコースで3週間使って学ぶ内容である。だがJavaを教材として使用しているプログラミング入門と比較して、言霊ならば教育時間が短縮できると考えてこの範囲を選択した。

教えた内容は、以下の4点である。以下、第1項～第4項でそれぞれを解説する。

1. プログラム作成の方法
2. プログラムの概要
3. 繰り返し文
4. 場合分け

第1項 プログラム作成の方法

プログラムは、最初にソースコードを記述した上でコンパイルを行う事により、実行可能なクラスファイルを作成するということを講義した。その中で最も単純なソースコード(図4-1を参照) 具体的にコンパイルするために必要なコマンドの使用例、そして作成したクラスファイルの実行方法を講義した。

```
メインとは {  
    亀(亀型)を作成する。  
    亀が100歩進む。  
}
```

図 4-1 単純な言霊ソースコード

第2項 プログラムの概要

プログラムを書くために必要な基本概念を講義した。メインメソッドの概念を教え、プログラムは図4-2のような構造をしていることを講義した。

```
メインとは { ... }
```

図 4-2 メインメソッドの構造

続いて亀オブジェクトを作成することの意味を講義した。亀オブジェクトを作成する際には図4-3のような表記をし、<>部分に亀オブジェクトの名前と型を入力することを講義した。プログラムの授業をする際、このような初期段階でオブジェクトの存在を示すのは妥当ではないのでここは簡単に説明するだけだった。

```
<名前> (<型>) を作成する。
```

図 4-3 宣言文の構造

さらに亀オブジェクトを用いて命令を実行するやり方を講義した。命令を実行する際の記述方法は図4 - 4のように行うこと、また亀型のオブジェクトが使用できる命令を図4 - 5のように示した。

<主語>が<命令>。

図 4 - 4 オブジェクトに対する命令の構造

~ 歩進む
~ 度右に曲がる
~ 度左に曲がる
筆を上げる
筆を下げる

図 4 - 5 亀型が使用できる命令一覧

ここまでの説明により単純なタートルグラフィックスのプログラムを作成するのに必要な予備知識を講義した。そこでプログラムの順序実行の概念を教えながら、基本的なプログラム例を図4 - 6のように示した。ここでは四角形を記述するプログラムを例に示した。

```
メインとは{  
  亀（亀型）を作成する。  
  亀が 100 歩進む。  
  亀が 90 度右に曲がる。  
  亀が 100 歩進む。  
  亀が 90 度右に曲がる。  
  亀が 100 歩進む。  
  亀が 90 度右に曲がる。  
  亀が 100 歩進む。  
  亀が 90 度右に曲がる。  
}
```

図 4 - 6 言霊のプログラム例

第3項 繰り返し文

先ほどのプログラムが図4 - 7に示す部分の作業の繰り返しになっていることを指摘した。

```
亀が 100 歩進む。  
亀が 90 度右に曲がる。
```

図 4 - 7 繰り返しになっている部分のコード

このプログラムの場合はこの作業を4回繰り返しているということを指摘し、図4 - 8のように繰り返し文を用いて書き直したプログラムを示した。

```
メインとは {  
    亀 ( 亀型 ) を作成する。  
    {  
        亀が 90 度右に曲がる。  
        亀が 100 歩進む。  
    } を 4 回繰り返す。  
}
```

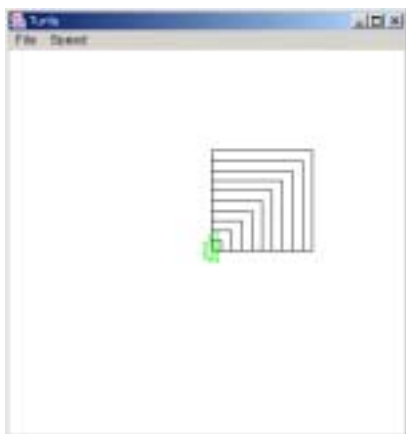
図 4 - 8 繰り返し文を用いたソースコード

その上で以下の図4 - 9を示して繰り返し文の文法を解説した。<回数>の場所に数値などを入れるとその回数だけ繰り返すという、繰り返し文の概念を講義した。

```
{ . . . } を <回数> 回繰り返す。
```

図 4 - 9 繰り返し文の構造

さらに以下のような絵を記述するためのプログラムを図4 - 10のように示した。この中で描画する四角形の大きさを保存するために変数を使っていることを指摘し、変数の概念を講義した。



```

メインとは {
    亀 ( 亀型 ) を生成する。
    A ( 整数型 ) を生成する。
    10 を A に代入する。
    {
        {
            亀が A 歩進む。
            亀が 90 度右に曲がる。
        } を 4 回繰り返す。
        A + 10 を A に代入する。
    } を 10 回繰り返す。
}

```

図 4 - 1 0 変数を用いたソースコード

また、変数宣言のための宣言文と、値を代入するための代入文の文法を以下の図 4 - 1 1 のように示して解説した。

```

<名前> ( <型> ) を作成する
<値> を <変数名> に代入する

```

図 4 - 1 1 宣言文と代入文の構造

また代入文で用いる値の表現として記述できるものを講義した。それを以下に示す。

```

算術表現 . . . . . 1 + 2、5 × A、( 3 + A ) × A
日本語算術表現 . . . 1 に 2 を足したもの、5 に A を掛けたもの
                      3 に A を足したものに A を掛けたもの

```

第4項 場合分け

以下のような絵を書くためのプログラムとして、図4 - 12のような例を示した。



```
メインとは {
    亀 (亀型) を作成する。
    回数 (整数型) を作成する。
    0 を回数に代入する。
    {
        もし 2 で回数を割った余りが 0 と等しいならば {
            { 亀が 20 歩進む。亀が 90 度右に曲がる。} を 4 回繰り返す。
        }
        もし 2 で回数を割った余りが 0 と等しいならば {
            亀が 30 度右に曲がる。
            { 亀が 20 歩進む。亀が 120 度右に曲がる。} を 3 回繰り返す。
            亀が 30 度左に曲がる。
        }
        回数 + 1 を回数に代入する。

        亀が筆を上げる。
        亀が 90 度右に曲がる。 亀が 30 歩進む。 亀が 90 度左に曲がる。
        亀が筆を下げる。
    } を 4 回繰り返す。
}
```

図 4 - 12 場合分けを用いたソースコード

また、このプログラムの中で使われている仮定文の文法を示した。



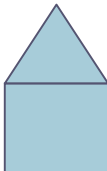
```
もし<仮定条件>ならば { . . . . }  
もし<仮定条件>なら { . . . . }  
もしも<仮定条件>ならば { . . . . }  
もしも<仮定条件>なら { . . . . }
```

第二回の授業では、以上の4点を講義して終わった。自分でメソッドを定義して呼び出す方法は教えていないものの、簡単なタートルグラフィックスのプログラムを記述するのに必要な情報を講義した。

第4節 第三回目の授業

第三回目の授業は課題演習である。演習問題をいくつか提示して、授業時間内にできる範囲で取り組んでもらった。そして2週間後の授業時間内に演習問題の答えと「言霊」に関する感想などをレポートとして提出してもらった。課題を出すにあたり、プログラムをちゃんと書けることが評価に直結する訳ではない事を強調した。この授業はプログラミング入門の授業ではなく授業目的もプログラムが書けるようになることではない。こちらが授業をさせてもらって、どのようなレポートが返ってくるかの反応を見ただけである。だからプログラムの課題が分からなければ分からないと書いてレポートを提出しても構わないということを強調した。

演習問題は5問あり、さらに自由課題を設定した。自由課題とは、言霊のタートルグラフィックスを用いて自由に絵を描いてみなさいという課題である。以下に演習問題を示す。

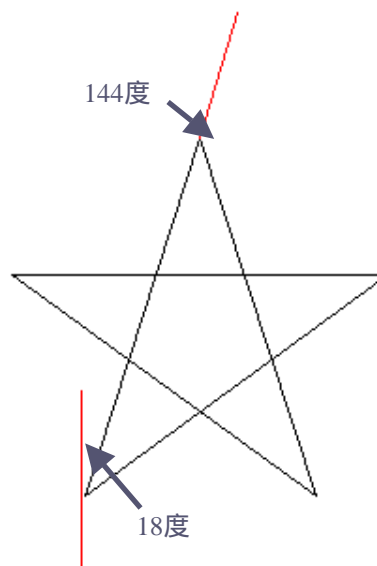
演習問題 1 三角形を描くタートルをプログラムしなさい。	
演習問題 2 四角形を描くタートルをプログラムしなさい。	
演習問題 3 家を描くタートルをプログラムしなさい。	

演習問題 1 ~ 3 は、練習問題である。最も単純なプログラムを書いてそれを動かすだけの練習問題として設定した。

これらの問題で間違っている生徒は全くいなかった。美しくないコードであったり、将来メソッドを使って記述をしないおすと問題が起こりそうなコードは散見されたが、今回の授業ではそこまで教えていない。今回の授業ではタートルを用いて描きたいことが描けるか、言霊であれば描けるのかを見るのが主目的である。

演習問題 4

右図のような星を描くタートルをプログラム
しなさい。



演習問題 4 は、角度を操作して絵を描くというタートルグラフィックスを練習して欲しくて設定した。また星を描きたいという初心者は多いので、星を描くために角度をヒントとして与えた上で描画してもらうことにした。

この問題の正答率もほぼ 100%であった。

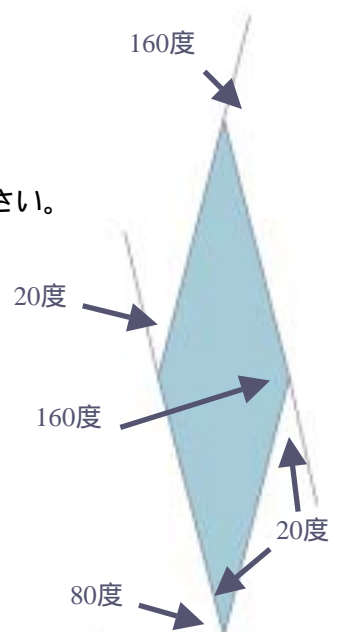
演習問題 5

これから、右のような花をタートルを用いて描こう
と思います。続く問題に教えてください。



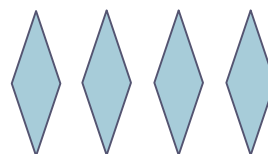
演習問題 5 - 1

右図のようなひし形を描くタートルをプログラムしなさい。
ただし一辺の長さを 50 とする。



演習問題 5 - 2

右図のようにひし形を並べて描くタートルをプログラムしなさい。ただし、プログラムは 5 - 1 のものを改造し、下のプログラムの「ひし形を描く部分」の穴を埋める形で作りなさい。



メインとは {

亀（亀型）を作成する。

{

ひし形を描く部分

亀が筆を上げる。

亀が 90 度右に曲がる。

亀が 30 歩進む。

亀が 90 度左に曲がる。

亀が筆を下げる。

} を 4 回繰り返す。

右に 30 歩
移動している

}

演習問題 5 - 2

花を描くタートルをプログラムしなさい。ただし、5 - 1、5 - 2 で書いたプログラムを良く理解したうえで、下のプログラムの「ひし形を描く部分」の穴を埋める形で作りなさい。



メインとは {

亀（亀型）を作成する。

{

ひし形を描く部分

亀が 20 度左に曲がる。

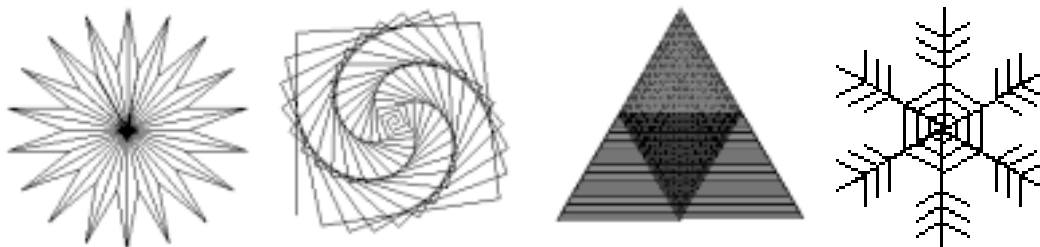
} を 18 回繰り返す。

演習問題 5 は、花を描くという課題を解く際にどのようなプロセスで考えればいいのかを教えるために出題している。いきなり花を描きなさいと出題したらおそらく解けない。どのように考えたらいいのか分からず、人によってはただ力技で全座標をハードコーディングする人もいるだろう。だから花を描く為には、プロセスを分解して 1 つのひし形を描いてみて、それを積み上げることで花を描くのだということをこの問題を通して理解して欲しかった。

この問題もほとんどの学生が回答して正解している。

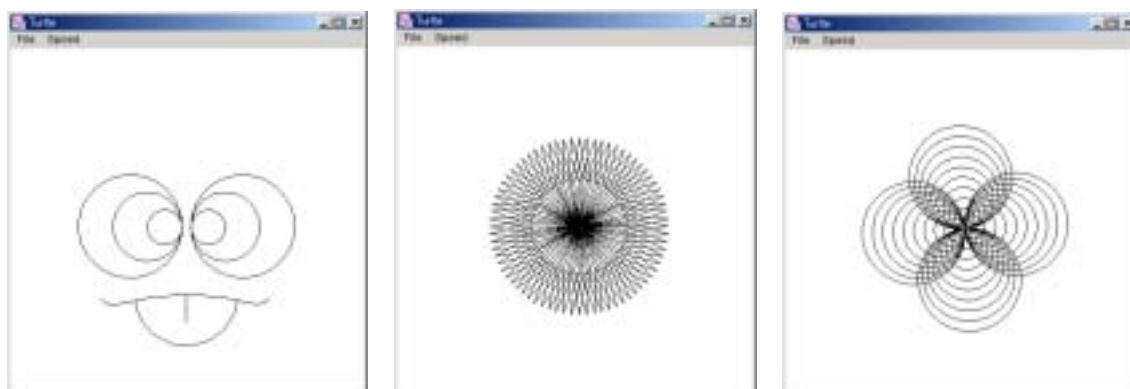
応用演習

以下のような絵を描くタートルをプログラムしてください。また、好きな絵を描いてみてください。



応用演習はヒントを与えずに、自分で考えてプログラムをするとどのような反応が返ってくるかを見るための問題である。筆者は応用演習に関して、どうせできないだろうと余り期待はしていなかった。だが受講者の半分はこのうちのどれか、あるいは全ての図形についてプログラムを作成してきた。特に一番難しいと思っていた左から2番目の渦巻き四角形の正答率が非常に高かったのが印象的である。また、感想のところ「プログラムが苦手なプログラム入門を受講するのは憂鬱だったが、言霊は楽しかった」と述べていた学生のプログラムを見ると、全ての演習問題と全ての応用演習についてプログラムがちゃんと記述されていた。

また、「自由に絵を描いてください」という課題に対しては以下のような作品が提出された。



第5節 授業の感想について

レポートでは、言霊を使ってプログラムを書くことでどのような感想を持ったかを書いてもらった。その代表的な感想を紹介する。

・楽しかった

プログラムの表記がアルファベットではなく平易な日本語になっただけで、プログラム言語に対するストレスが減り楽しく課題に取り組めた人が多かった。

まず思ったのはプログラミングってこんなに簡単だったのか？ということです。

1年のプログラミング入門の授業を通して最後のほうは訳がわからなくなって苦痛に感じていたプログラミングを実際にまたやってみて、こんなに楽しいと考えるようになるとは思いませんでした（環境情報学部2年、男）

・プログラムの可読性が高い

プログラムが読みやすい為に、プログラムの理解がスムーズに行くという意見が多かった。

プログラムが分からない友達に日本語プログラムを教えていた時、何をやるメソッドなのか、この一行はどんな動作をするか、が一目で分かるので教えやすく、また教えられているほうの理解も早かった。初めてプログラムを勉強するときには、まず簡単なコードを読むことから入るわけであるが、読んだときにそのコードが何をしているのかが分かると自分でもすらすら書けるようになる、という基本的な部分の重要性を改めて実感した（環境情報学部3年、女）

後になってプログラムを見直したときに、すぐに理解できるのは良いことだと思う（環境情報学部2年、男）

・論理的思考がしやすい

日本語で記述できるということは、日本語で思考したものがそのままプログラムで動かせるということである。その為思考がスムーズになったという指摘が多かった。

思考を途中で断ち切られることが少なかった。（中略）言霊ではプログラムを日本語で考えることができるので、思考と論理的表現の乖離が小さく、負担が少なかった。（環境情報学部2年、男）

以前は、「日本語で考えて、プログラム言語化して、書いてみて、実行してだめだったら最初に戻る」という手順だったので、1つ減っただけでもすごく気持ちがいい。母語で考えたことをそのまま書き下ろすことができるのは快適。（環境情報学部2年、男）

Java などを使った教育により分かりにくい文法に四苦八苦してしまうという経験を持っていて、言霊だとそういった表面的な文法ではなく論理的思考のトレーニングになるとい

う意見もあった。

言霊を使って「プログラミングとは論理的思考を行うことである」ということを改めて意識した（環境情報学部3年、男）

使い慣れない表記法にとられることなく、直にアルゴリズムの流れのみを考えながら日本語で記述できるので、初期段階の「プログラムの思考」の教育に対して有用であると思われる（政策・メディア研究科1年、男）

一方で、日本語プログラミング言語は書きにくいという指摘もあった。

初めてプログラミングを習う人にとっての親しみやすさとしては日本語プログラミングに分があるかもしれないが、書きやすさとしてはPrologのほうが優れていると私は考える。（政策・メディア研究科1年、男）

言霊は、現在見やすく理解しやすいが、決して書きやすい言語ではないと考えた（政策・メディア研究科1年、男）

日本語にしたために書きにくいという問題について、具体的な指摘が数多くされた。その中で最も多かったのは語順の問題である。タートルを使って曲がるときに、現在の仕様では「亀が90度右に曲がる」などと記述されることになっている。だがこれを「亀が右に90度曲がる」と記述した人が非常に多く、レポート提出者の半分はどこかでこの問題に言及していた。

同義語の問題も指摘された。「亀が筆を下げる」を「亀が筆を下ろす」と書いてしまったり、「亀が90度右に曲がる」を「亀が90度右に回転する」と書いてしまった例があった。

同じような意味を持つ助詞の問題も指摘された。「亀が90度右に曲がる」を「亀が90度右へ曲がる」と書いてしまった例があった。

また漢字の送り仮名の問題も指摘された。「繰り返す」というキーワードを「繰り返えす」と記述してしまった例があった。

第5章 今後の課題と展望について

本研究において制作した日本語プログラミング言語「言霊」は、読みやすいプログラムを記述できることを最大の目的とした。授業の結果を見ても分かる通り、受講者にはソースコードが読みやすく親しみやすかったという感想が多数出ているし、受講者のやる気と理解を示すように課題の正答率も高かった。

だが「言霊」は、現状では読みやすく書きにくいプログラミング言語である。日本語であるがゆえにプログラムの記入ミスをしてしまう例が多かった。存在している課題と今後の展望についてまとめる。

第1節 語順の問題

例えば以下のような言霊における手続きがあったとする。上が手続き呼び出し文で下が手続き宣言文である。

亀が右に100歩進む。

「亀(亀型)」が「方向(方向型)」に「距離(整数型)」歩進むとは{・・・}

この手続きは普通に日本語で思考すると、以下のようにも表現できる。

- ・ 亀が100歩右に進む。
- ・ 右に亀が100歩進む。
- ・ 右に100歩亀が進む。
- ・ 100歩右に亀が進む。
- ・ 100歩亀が右に進む。

これは日本語が語順の変化を許す性質があるからだ。最後の動詞部分の位置は変わらないが、それ以外の全ての要素の順番はいくらでも変化しうる。現状のコンパイラでは、こうした語順変化には対応しない。上記の語順で宣言したらその通りに記述しない限りコンパイラは解釈しない。

だが書きやすさを求めるなら語順変化に対応するべきである。例えば以下のように節を「、」で区切って宣言して、節の順番が変化しても対応できるようにするべきであろう。

亀が右に100歩進む。

亀が100歩右に進む。

右に亀が100歩進む。

右に100歩亀が進む。

100歩右に亀が進む。

100歩亀が右に進む。

「亀(亀型)」が、「方向(方向型)」に、「距離(整数型)」歩、進むとは{・・・}
--

第2節 動詞活用に対応した言語仕様

「言霊」は語尾表現において問題が存在する。図5-1は例として「2乗」に関する手続きを宣言し呼び出している。この「2乗」手続きのように副作用を起こさず戻り値を返す手続きは、名詞表現で宣言をすると正常な表現になる。

```
「A (整数型)」の2乗とは { A × Aを返す。 }  
3の2乗を出力する。
```

図 5 - 1 正常な手続き呼び出し表現

だが副作用を起こしかつ戻り値を返す「出力する」手続きの場合は、正しい日本語表現にならない場合がある。ここで「出力する」手続きは、引数を標準出力に出力し、戻り値に実行結果を真偽値として返す仕様とする。

図5-2のように「出力する」手続きを動詞表現で宣言すると、6行目のように手続き実行は正しく表現できる。だが7行目のように実行結果を判定に使うときは、不自然な日本語になってしまう。

```
1: 「A (整数型)」を出力するとは {  
2:   (略) //出力処理  
3:   結果を返す。  
4: }  
5:  
6: 3を出力する。  
7: もし3の2乗を出力するが真ならば {  
8:   (略)  
9: }
```

図 5 - 2 副作用手続きの動詞表現

一方、図5-3のように名詞表現を用いて宣言して「出力」手続きを作成すると、7行目は正しい表現になるが、逆に6行目の手続き実行文は不自然になってしまう。

```
1: 「A (整数型)」の出力とは {  
2:   (略) //出力処理  
3:   結果を返す。  
4: } 事である。  
5:  
6: 3の出力。  
7: もし3の出力が真ならば {  
8:   (略)  
9: }
```

図 5 - 3 副作用手続きの名詞表現

この問題を解決する為には、コンパイラが動詞活用に対応する必要がある。日本語の動詞活用は論理的で例外が少ない為、与えられた動詞がどの活用形なのかを判定することは可能である。

仮に動詞活用に対応したコンパイラが実現できていたと仮定する。例えば図 5 - 4 では、手続きは「出力する」と宣言されている。手続き宣言は終止形で行われるとすると、この手続きはさ行変格活用の動詞であることが分かる。そして返り値を表す表現を「<動詞表現の連用形>結果」と文法で決めると、7行目の「もし3の2乗を出力した結果が真ならば」のような柔軟な表現が可能になる。また一方で6行目の手続き呼び出しも終止形を用いて「3を出力する」と正常な日本語表現が可能になる。

```
1: 「A (整数型)」を出力するとは{
2:   (略)//出力処理
3:   結果を返す。
4: }事である。
5:
6: 3を出力する。
7: もし3の2乗を出力した結果が真ならば{
8:   (略)
9: }
```

図 5 - 4 副作用手続きの動詞表現

また、このように動詞活用に対応していれば図 5 - 5 のように手続き呼び出しを命令形で表現してより直感的なコードを記述することも可能である。もちろん手続き呼び出しは命令形が良いのか終止形が良いのかは議論の分かれるところかもしれないが。

```
計算するとは{ (略) }
演算するとは{ (略) }
描くとは{ (略) }

計算しろ。
演算しろ。
描け。
```

図 5 - 5 柔軟な手続き表現

第3節 文脈を使った記述

日本語は、主語を省略する事が多い言語である。これは文脈において主語が自明であるときには省略する文化があるからだ。逆に英語は必ず主語を記述する必要がある言語である。この英語の特徴はオブジェクト指向においても反映されており、Java 言語における手続き実行文は必ず主語と動詞と目的語を明確に記述する必要がある。例えば描画設定を赤に変更する手続きは Java 言語では図 5 - 6 のように記述するが、これは主語と動詞と目的語が明確に記述されている。

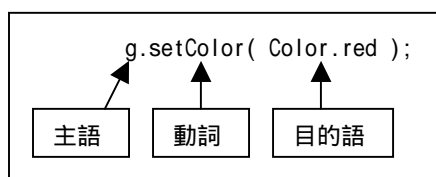


図 5 - 6 Java 言語の表現

だが日本語はより簡潔に「描画色を赤に設定する」と記述したいと思うのが自然である。この表現では主語が省略されているが、主語となる対象は、“赤”を引数にとる“設定する”手続きを持つオブジェクトである事が分かる。この情報だけで主語を特定できる状況は多いと思われる。よって一見曖昧に見える表現も文脈によって解決は可能であると考えられる。

第4節 構造エディタを用いた開発環境

プログラムを効率的に記述する為には効果的な開発環境が必要であり、言霊にも専用の開発環境があることが望ましい。既存の開発環境にはコードインサイトという機能がある。状況に応じてクラスの手続きや属性名などを表示し、選択する事で記述を進める事ができる仕組みなのだが、それにより開発者が全クラスの手続き名や変数名などを記憶しなくてもプログラムをスムーズに記述する事が可能になる。だが Java 開発環境などにあるコードインサイトは、Java 言語が英語の語順であるから実現できているのだが、「言霊」は日本語の語順である為に違うやり方が必要になるだろう。

開発環境には構造エディタを用いるのが適当である。構造エディタは文法知識を持っていて、ユーザに対して文法的に正しいプログラムの記述を強制することができる。初心者にとって正しいプログラム記述を開発環境によって強制されることは、余計な文法上の記述の失敗を避けることができプログラミング学習に効果的だと考えられる。

第6章 終わりに

第2章で紹介した小朱唇を開発した水谷静夫氏の論文[7]は、このような一節から始まる。

計算機が使えることが偉かった時代は過ぎたはずである。文科系それも特に語文系では、本気のアマグラマが育って欲しい、つまりマニア学生やパッケージの消費者的学生より、自分の問題を有ちそれを解く道具として計算機を使う学生が出てもらいたいのに、なかなかそうはいかない。

全く同感である。プログラム言語は一部の人間のみが小難しい記号をいじりながら扱うものではなく、全ての人間が扱えるべき道具である。プログラム言語を理解することで、人は真にコンピュータを道具として扱えるようになるのであり、その為に日本語プログラミング言語を研究した。

またプログラム言語やコンピュータは欧米人が開発したものであり、その設計思想に彼らの思考の多くが入っている。これらをわれわれ日本人がそのまま扱うのは難しいため、日本人にあわせた言語を作る為に多くの先人が研究を重ね、私も研究している。その精神は「和魂洋才」の心である。日本人としての精神を堅持しつつ、欧米人が発明したコンピュータを受け入れるのである。

日本語プログラミング言語「言霊」は、以上の精神の元に研究され、今後も続いていく。「言霊」により日本人がコンピュータを真に理解し、コンピュータを道具として使いこなせるようになったときに、情報化社会のあり方が変わる可能性がある。

印刷術の普及により従来ラテン語で記述された聖書が、日常語であるフランス語やドイツ語で書かれる様になった。この事によって学問が大衆のものとなった歴史がある。日本語プログラミング言語「言霊」も、一部の技術者や欧米人のものであったプログラミング言語を日常語に翻訳したことで、コンピュータを日本人の大衆のものとしたいのである。

第7章 参考資料

- [1]<http://www.sfc.keio.ac.jp/~tsaito/1bc/>
- [2]Randal L. Schwartz :「初めてのPerl」オライリー・ジャパン、305pp、1995年
- [3]長谷川浩行 :「ソフトウェアの20世紀」翔泳社、319pp、2000年
- [4]床分眞一、今城哲二 :「COBOLにおける日本語機能の現状と今後の動向」プログラミング言語 16-9、情報処理学会、pp53-56、1988年
- [5]西村恕彦 :「日本語とCOBOL」情報処理学会論文誌、Vol.8 No.3 pp.157-160、1967年
- [6]那野比古 :「要説 日本語AFL」東京ブック、189pp、1983年
- [7]水谷静夫 :「次第立て言語《小朱唇》の設計思想」情報処理学会 プログラミング言語 5-3、情報処理学会、1986年
- [8]水谷静夫 :「小朱唇 言語仕様」東京女子大学 日本文学科、69pp、1986年
- [9]水谷静夫 :「国語風次第立て言語 朱唇の手引き」東京女子大学 日本文学科、107pp、1989年
- [10]石田晴久、片桐明 :「パソコン言語学」アスキー、pp257-288、1984年
- [11]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、pp25-32、1988年
- [12]片桐明 :「日本語プログラミング言語 Mind 基本文法」(株)リギーコーポレーション、279pp、1988年
- [13]Jon Meyer, Troy Downing :「Java バーチャルマシン」オライリー・ジャパン、480pp、1997年
- [14]Bertrand Meyer :「プログラミング言語理論への招待 正しいソフトウェアを書くために」アスキー出版局、450pp、1995年
- [15]<http://www.rational.com/>
- [16]岩田茂樹、笠井琢美 :「有限オートマトン入門」森北出版株式会社、91pp、1986年
- [17]中田育男 :「コンパイラ」オーム社出版局、193pp、1995年
- [18]ジェラルド・ジェイ・サスマン、和田英一 :「計算機プログラムの構造と解釈」ピアソン、409pp、2000年

第 8 章 謝辞

はじめに、環境情報学部の大岩元教授に深く感謝いたします。教授には論文執筆中のみならず、学部在籍時から 4 年近くご指導を頂きました。その中で数々の貴重な意見及び知識を学ばせていただきました。特にオートマトン勉強会を大岩先生の指示のもと実施していなかったら本研究は成り立ちませんでした。また学会での発表を促して尻を叩いてくれたおかげで本論文を書く際にネタ不足で困るということが全くありませんでした。博士課程でも変わらぬ御指導御鞭撻の程をよろしく願いいたします。

次に、SFC 研究所の中鉢欣秀さんに深く感謝いたします。共に日本語プログラミングという夢を共有して、貴重な意見と知識を学ばせていただきました。特に抽象文法に関する Meyer の本との出会いがなければ本研究も現在のような形にはならなかったでしょう。また現段階では反映できていませんがサスマンの SICP の本も貴重な知識となりました。いつかこれを言霊に反映させたいです。

さらに、CreW プロジェクトメンバーの全員に感謝いたします。修士 2 年の後半に研究に没頭できたのはみなさんの支援の賜物です。Boxed Economy プロジェクトを頑張っている松澤さん・海保さん、enTrance プロジェクトを頑張っている小林君・杉浦君、研究会を切り盛りしてくれている青山君・川村君、いつも掃除や事務処理などの縁の下の力持ちの福田さんに深く感謝します。また研究を超えていつも興味深い示唆を与えてくれる斉藤俊則さんにも深く感謝いたします。修士論文に関して貴重な助言をくれた秋山君にも感謝いたします。

最後になりますが、温かく見守ってくれた両親と兄に感謝いたします。この研究を続けられたのは家族の支えがあったからです。本当にありがとう。

