コンピュータと教育 47－5
（1998. 3. 20）

プログラミング学習を支援する言語処理系「NB2」の設計

橋本裕† 早川栄一† 並木美太郎† 高橋延匡‡
†東京農工大学 ‡拓殖大学

開発用の処理系をプログラミング教育に使用すると、(1)機能が多すぎる、(2)概念や文法は既知であるとして設計させている、(3)評価のためのデータ収集が難しい、(4)ターンラウンドタイムは問題にされていない、という問題がある。本稿では、その問題を解決するためのプログラミング教育用の言語処理系「NB2」について述べる。この処理系は、プログラミングに必要な基礎概念、すなわち制御構造やデータ構造、言語構文、計算量の学習支援機構を持っている。特にプログラミングをしている時は実行時よりも編集時に学習支援機能が必要になることがある。そこで、編集時と実行時に統一的にそれらの学習支援を行えるように設計した。

NB2: Language Processor to Support Programming Education

Yutaka Hashimoto†, Eiichi Hayakawa†, Mitarou Namiki† and Nobumasa Takahashi‡
†Tokyo University of Agriculture and Technology
‡Takushoku University

Using language processor for development to educate programming, following problems appear: (1)it has too many functions, (2)it have been designed as concepts and grammar have been already understood, (3)it is difficult to measure data for evaluation and (4)it doesn't care turn around time. This paper describes a language processor to solve the problems for programming education. This processor has mechanisms to support learning of basic concepts, such as control flow, data structure, syntax and computational order, that required for programming. Especially when programming, the functions of learning support may be required in the editing phase rather than the execution phase. Thereby, we've designed to support the learning in both the editing phase and the execution phase as same way.

1. はじめに

コンピュータサイエンスの学習では、プログラムの作成能力が必要とされる。このプログラムの作成能力とは、単純にプログラムを記述することができる能力だけではなく、より広い、例えばプログラミングに必要となる基礎概念の理解という内容も含まれている。

このプログラム作成能力を養うため、学習者はプログラミング言語の文法やプログラミングの基礎概念を学習し、プログラミング技術を磨かなくてはならない。しかし、彼らは初めは初心者である。そのため、例えば、制御構造の概念もまだ理解していない。そこで、その概念を実際に理解させるために、プログラムを作成・実行させると、理解に大きな影響を与える。なぜなら、その座学で得た抽象的、概念的知識を実際に適用することで、その知識が具体化され、理解が促進されるからである。ただし、抽象的・概念的知識はそのまま実際のプログラムに適用できず、プログラミング言語に変換して記述しなくてはならない。しかし初心者にはこの変換作業は難しい。

この難しさをやわらげるために、あらかじめ変換作業が終了した(つまり理解している人が作成した)サンプルプログラムを用意し、そのサンプルプログラムを少しずつ修正していくことで知識を身につけていくという方法がよく取られる。これは、難しさをやわらげるための一つの方法であるが、もっと直接的にその概念と実際のプログラムとの対応を明確に示すことができれば、学習者の理解を促進することができると考えられる。本稿では、そのようなプログラムの作成能力を養うために必要な機能を持つ言語処理系「NB2」について述べる。

2. 開発用と教育用の処理系の違い

本章では、開発用の処理系を教育に適用したときの問題点と、教育用の処理系に対する要求事項について述べる。

2.1 開発用の処理系の問題

プログラミングの学習では、他に適当な処理系がないというような理由やその他の理由から、開発用として販売されている処理系をプログラミングの学習に利用していることが多い。しかし、開発用の処理系のプログラミング教育へ適用には、次の問題がある。

・機能が多すぎる

開発用の処理系は開発を目的としているので、製品を作成しやすいよう数多くの選択肢を設けたり、数多くの最適化オプションを提供している。そのため、何の機能かよくわからないようなメニューが大量に表示されたり、操作が複雑になったりする傾向がある。このような傾向は、初心者の使い方を覚える手間を増えさせたり、多すぎるメニューを見てプログラミングに対し心理的な壁を作ってしまう原因にもなる。操作の学習は、プログラムの学習にとって、本質的ではなく、あまり意味を持たないものである。

・概念や文法は既知であるという前提で作られている

一般的にいって、開発用の処理系は、そのベースとなる言語やプログラミングの基礎概念は既知であるという前提をもって開発されている。例えば、初心者がすぐには理解できない、値渡しと参照渡しの違いに対し、特にその理解のための支援を行うということはたいていの場合ない。これは、プログラムを開発するための処理系であるので、そのような概念はプログラマは知っているとの前提があるからである。しかし、初心者は座学で学習した後でも、そのような概念を本当に理解しているかは疑わしい。したがって、その違いを意識してプログラムを書けるとは限らない。

・アルゴリズム評価のためのデータ収集が難しい

アルゴリズムの学習時には、その時間的空間的な計算量の評価を行う必要がある。そのため論理的考察以外にも、実際の実行データの収集が必

要になる。しかし、このデータを単純に収集しようとすると非常に多くの労力を必要とする。開発用の処理系では、プロファイル機能として提供されることもあるが、別パッケージになっているのでそれ用の操作を別に学習する必要があったり、機能が大量にありすぎて混乱したり、と初心者が使うには問題がある。

・ターンラウンドタイムは特に問題とされていない

開発用の処理系では、プログラム作成のターンラウンドタイムも、短い方がよいが、特に短くなければならないということもなく、あまり問題にされていない。

2.2 教育用の処理系への要求

前節での問題点を踏まえると、教育用の処理系には次のことが要求される。

・機能は過剰にせず必要最小限にする

最適化の機能やカスタマイズの機能、詳細すぎる機能などは初心者にとって混乱の元である。教育用の処理系では、むしろ最小限の必要な機能だけを提供した方がよい。

・概念や文法を理解していないという前提の元に機能を提供する

教育用でも、プログラムを作成するという観点からは、開発用の処理系と同じプロセスを踏まなくてはならない。しかし、プログラミング時に必要となる文法やプログラミングの基礎概念は初心者はまだ理解していない。そこで教育用の処理系ではこの点の支援を行う必要がある。

・評価のためのデータ収集機構を用意する

前述したようにアルゴリズムの学習には評価のためのデータを取得する必要がある。そこでそのためのデータ収集機構が必要である。

・ターンラウンドタイムは短い方がよい

教育用の環境ではターンラウンドタイムは短い方がよい。これは自分の変更した部分のプログラムの動作をすぐに確認することができるようにするためである。すぐに確認することができると、プログラムを少しずつ確認しながら修正していくことができ、その結果一度に大量かつ理解不能な事態を招く危険性が少なくなる。

3. 編集・実行機構の設計

教育には、開発用の処理系を使用するよりも、教育用として設計された処理系を使用する方が初心者の学習の役に立つ可能性が高い。本学でもプログラミング入門講座で市販の環境を使用しているが、単純に実行できる環境が提供されているだけで、実行時の状態がわからないという問題があったり、日本語を使用すると問題が発生したりしていた。もちろん、概念の学習支援のための機能は存在していなかった。そのようなことから、新たに教育用の処理系を設計する。

プログラミングの学習には言語処理系が欠かせない。しかし、そのために学習者はソースプログラムファイルや学習のメモファイルといった資源を管理しなくてはならない。学習はステップアップで行われるため、この管理は重要である。文献[1]でこの管理方法が述べられている。ここでは、その中の一つのページである編集・実行のページについて述べる。

編集・実行のページで学習者はプログラムを編集したり実行したりすることができる。また、同時にこのページで基礎概念の学習支援が行われる。本章ではそれらの学習支援の方法について説明する。

3.1 設計方針

編集・実行機構の設計にあたっての設計方針は次のとおりである。

・編集時にも実行時とほぼ同等の学習支援を提供する

[2]ではもともとBNFの学習支援だけではなく、他の概念の教育も念頭におかれていた。例えば、編集系、つまりエディタでは、トークンの種類ごとに色分けすることで識別名の種類の違いを認識さ

せ、変換系、つまりコンパイラの部分では BNF の学習支援を、そして実行系では、変数を箱として表示することで、それが何かを置くための場所であることを示すということを考えていた。しかし、多くの情報はソースプログラムを解析した後、つまり実行時にしか得られない。例えば、参照渡しの引数に式を書いたときにはどうなるのかといったことは、実行時にしかわからない。

ここで、実際にプログラムを組むときのことを考えると、それらの情報が必要なのは実行時というよりむしろ編集時である。例えば、先ほどの参照渡しの例では、プログラムを書いているときに、向こうから「参照」されるとはいったいどういうことか、もしそこに式を書いたら何が起こるのかということを実行時ではなく、プログラムを書いているときに知りたいのである。そのように考えると、編集時にほしい学習支援も存在することわかる。

そのようなことから、編集時と実行時のほぼ同様な学習支援機構を提供することでプログラムを実行しなくても、その場でその概念の学習支援ができるようにする。

また編集時に解析した情報を元に、すばやく実行状態に移行することができるようになるので、ターンラウンドタイムの短縮にもなるという利点がある。

・概念を説明することができるように可視化をする

概念というものは、抽象的なのでわかりづらい。それを説明するためには、具体的に示した方がわかりやすい。そこで、実際にそれを、説明しやすいように可視化をする。

3.2 全体構成の設計

プログラム作成には、エディタが必要であることから、編集のための領域が必要である。また、変数は、ある値を持った箱ということを示すため、あるいは、参照引数とは何かということを示すために、変数の表示をするための領域が必要である。そして、プログラムの構文エラーを理解するためには、構文のどこで誤っているのかを示す部分が必要である。そのようなことを考え、全体の構成を図 3.1 に示す表示にすることにした。

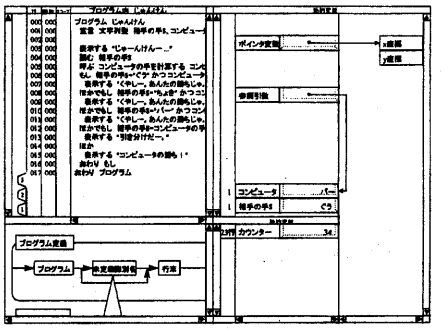

図 3.1 編集・実行機構

この図では、編集部分とは、左上の区画で、構文表示部分とは左下の部分であり、変数表示部分とは、右側の部分である。これが同一のページに表示される。それぞれは密接な関係を持っているので分割させることはできないようにする。なお、エディタには、そのソースプログラムについて他の人と話がしやすいように、行番号を表示する。

3.3 編集カーソルとプログラムカウンタ

編集部分にはエディタがある。したがって、挿入点を示す編集カーソルが必要となる。これは、Windows 上で実現することを考慮して、カーソルを縦棒で、また範囲選択を黒の反転色であらわすことにする。また、実行時には、編集カーソルは、その編集モジュール内の学習者が着目したい点をあらわすカーソルとして機能する。

プログラム実行時はコンテキストがある。これにはプログラムカウンタが含まれ、現在の実行部分をあらわしている。これがどのように動くかでプログラムの流れが変わるので、プログラムの学習では特に注目すべきものである。そのようなことから、わかりやすくするため可視化をする。具体的には、現在の実行点を意味しているトークンの部分をある色で反転することによりそれを意味する(図 3.4 の現在のプログラムカウンタを参照)。

プログラムカウンタは実際にプログラムコードが出現する部分にしか現れない。この性質を考えると、実行文と非実行文の概念を教育することができ

る。そこで編集時にもプログラムカウンタを出現させる。編集時のプログラムカウンタは編集カーソルと同じ位置に出現する。つまり、編集カーソル位置＝現在の実行位置である。もちろん、このプログラムカウンタは、宣言や構造体の宣言といった非実行文に編集カーソルが移動した場合には、出現しない。この表現により、実行文と非実行文の教育をすることができる。

3.4 変数と定数

次に、変数と定数について考える。手続き型プログラミング言語でプログラムをするときには、変数と定数という考えがある。変数は値を入れることのできる「箱」であり、定数は変更することのできない値である。

しかし、彼らが今まで学習してきた数学や算数では、変数は漠然と何かの値に結び付けられたものであり、代入という考えも存在していなかった。この点が初めて学習するときの問題点になる。

そのようなことから、変数を表示するときには、それが値を格納することができる箱であることを明確に示すため、箱の表示を行う。そして、定数は値であることを示すため、プレートとして表示をする。

また、変数や定数に名前が結び付けられている場合、それを示すためにラベルを左に付加する(図3.2)。

変数名	3
定数名	2

図 3.2 変数と定数の表示

3.5 制御構造の学習支援

制御構造には、大きく分けて三つのタイプがある。それは、基礎構造、ルーチン、モジュールである。基礎構造とは、連接や、選択、反復、飛越しといった概念である。これらは、関数や手続きといったルーチンの中に現れる。モジュールはその名のとおりモジュールである。

3.5.1 基礎構造

まず、連接の概念に関しては、特に支援を行わない。これは、プログラムは原則的に上から下へ流れるということを教えれば、容易に理解することができるからであり、可視化する必要がないと考えたからである。

次に、選択や反復といった概念は、それ自体で構造を持っており、条件により、その中の区を実行したり実行しなかったりする。したがって、構造を持っているということと、内部の区を単位にして実行するということを理解することが重要となる。そこで、これらの概念が構造を持っていることを示し、どの部分が区になるのかを示す。その様子を図 3.3 に示す。図 3.3 では構造の部分の背景色を変更している。この背景色は構造ごとによって、色を変えることで、それぞれが違う概念であることを認識させる。また、その構造の色の先頭は必ずその構造をあらわす予約語が現れることから、その構造がどういう名前かも知ることができる。また、この構造に挟まれた部分が区であることも明確にわかる。

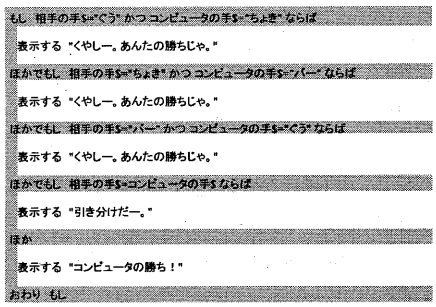

図 3.3 構造対応の表示

そして、飛越しやループから抜ける特殊構文(例えば、この言語の場合「ぬける　ループ」など)、区をぬける場合は、次に実行する位置は次の行ではない。この場合、実行すべき位置が次の行でないことを示すことが重要となる。そこで、プログラムカウンタがその部分にきたとき次に実行すべき位置が次の行でないことを示すために、その位置から次の実行位置まで矢印を表示する(図 3.4)。

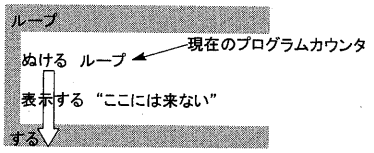

図 3.4 非通常遷移時の矢印表示

3.5.2 ルーチン

ルーチンとは、手続きや関数のことである。手続きや関数の重要な性質は、呼び出した後元の位置に戻ってくる、引数を渡すことができる、関数は結果を返すということである。そのことを考慮に入れて、まず手続きや関数も、構造を持っている。そこで図 3.3 に示した表示を手続きや関数に対しても行う。

ここで呼出しレベルという言葉を定義しておく。呼出しレベルは、現在の関数や手続きまでのスタックフレームの存在個数をあらわす。つまりある手続きのレベルを 1 とすると、そこから何か関数か手続きを呼出すとその関数や手続きの呼出しレベルは 2 になり、さらにそこから他を呼出すと 3 になるというような値である。もちろんこの値は実行時にしかわからない。

これからは、とりあえず実行時の動作について考える。関数を呼び出すと値を得ることができる。今回は理解の単純さを考慮して、実行をスタックマシーンで実現することを考える。すると値を返す場所をあらかじめスタック上に確保する必要がある。そこで、変数のスタック表示の部分に現在の呼出しレベルで「～の戻り値」という名前の未初期化変数を作成する。

次に、各引数を評価していく。

そして、実際に呼出す段階で、「戻り先」という名前の変数を作成しそこに戻りソース行を記述する。呼ばれた関数や手続きは現在の呼出しレベル+1として動作する(図 3.5)。

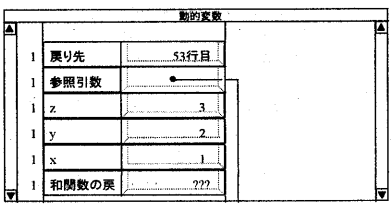

図 3.5 戻り値・引数渡しの表示

このように実際に実行時に行われている動作をそのまま見せることにより、手続きや関数の理解を深める。これは、特に値渡しや参照渡しを理解するときに重要である。なぜなら、ソース文面上だけからではこれらの内部状態はまったく理解できず、これが理解を難しくしている原因となっているからである。

しかし、これだけでは、元の位置に戻ってくるという機構は理解できるかもしれないが、イメージ的には理解しづらい。そこで、編集の部分の最も左側に呼出しレベルをつけたタブを表示して、呼出しレベルが増えるごとにそのタブをスタック上に積み上げてゆく。もちろん下の方にあるタブをクリックすると、それを呼出す前のソースプログラムが表示されるようになっている。このような表示方法で元の位置に戻ってくるという性質を表現する。

また呼出しレベルをつけることにより、どの部分のタブにどの変数が対応しているかを簡単に知ることができる。

ここまでは実行時の話であった。この手続きや関数は理解するまで(特に引数の渡し)が大変であり、実行時よりもプログラム作成時に支援がほしいところである。方針に基づき、編集時にもなるべく同じ機能を提供する。

プログラムカウンタ(つまり編集カーソル)が呼び出す関数名の上にきたときには、戻り値の箱をスタック上に表示する。そして引数の上をプログラムカウンタが(編集カーソルが)移動するにつれて、値が不定の引数の箱をスタック上に表示する。もし、関数や手続き名上で呼び出すというコマンドを選択した場合には、実行時に呼び出すように、新しい呼出しレベルが作成される。もし呼出し先でローカル変数が宣言してあれば、それらがスタック上に配置される。このような動作をすることで、編集時に引数渡しがどのようなものかを説明することができる。

3.5.3 モジュール

本処理系の提供する言語では、論理的なモジュールは存在せず、物理的なファイルモジュールしか存在しない。そのようなことから、物理的なモジ

ュールを一つの編集単位として提供することでモジュールの概念認識させる。

3.6 データ構造の学習支援

データ構造は、基本データ型、構造データ、配列、ポインタといった概念を含んでいる。本節では、これらの概念の学習支援方法について説明する。

3.6.1 基本データ型

言語には最も基本となるデータ型が存在する。それが基本データ型である。本処理系の対象とする言語でも、基本データ型として、数値型、短数値型、整数型、バイト型、論理型、文字列型、何でも型が存在する。ちなみに数値型とは他の言語でいうところの実数型のことである。これらのすべての基本型を理解する必要はないが、最低限数値型と文字列型は理解する必要がある。

ここで、理解とは、それが何をあらわしているのかということを理解するということである。つまり、どうして 0.0001 を 10000 回足した値と 1 を比較すると等しくないのかといったことや、文字列をソートするには文字コードの話が出てくるが、文字コードとは何か、文字列とはどのような関係があるのかということを理解するということである。

そこでこの理解を助けるため、各データ型の内部表現を表示できるようにする。具体的には、変数の内部に値として表示されている通常表記のほかに、2 進数での表記法や、数値型であるならば、浮動小数点の内部形式の見やすい表示方法を提供する。

また、型の違いを認識させるため、それぞれのデータ型の箱の色を変えることでそれを表現する。

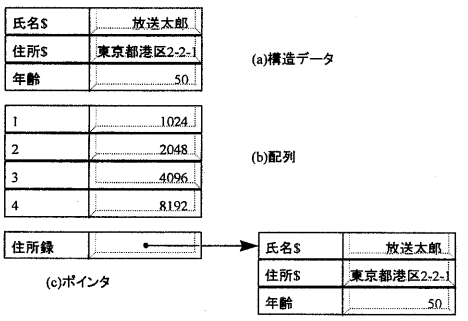

図 3.6 構造・配列・ポインタの表示

3.6.2 構造データ・配列・ポインタ

構造データ(他の言語でいう、構造体やレコード)・配列・ポインタはデータとしてある種の構造を持っている。そこで、それらの特徴をそのまま表示し、理解を図る(図 3.6)。

3.6.3 表示場所

前節までで、どのように各データを表示するのかについて述べた。ここではどこにそれを表示するのかについて述べる。

以前に述べたように、関数や手続きの引数渡しの理解にはスタック上に存在する動的変数の考え方が必要である。また、ポインタの学習では動的に作成される変数を理解する必要がある。これらは動的変数と呼ばれる。さらに、静的な変数も存在する。

学習者は結局のところこれらの変数の性質について理解しなくてはいけない。

そのことを踏まえ、スタックやヒープ領域、静的変数領域をそのまま表示することにした。それが図 3.1 の右半分である。スタック上の変数は、動的変数と示された部分の左側に、ヒープに割り当てられる変数は右側に表示する。また、静的変数はスタックの下にわけて表示する。このように表示することで、動的変数と静的変数の概念と存続範囲の概念を説明しやすくする。

なお、スタック上には関数や手続きの引数だけではなく、普通に宣言された変数もスタック上に表示される。

またそのまま表示するといっても、実際の CPU のアドレスを表示したり、[3]のように完全なスタックの使用状況を見せるわけではなく、関数呼出しの戻り値を表示するというようなある程度簡単化されたものを表示するという意味である。

ここで、今まで述べた表示はいずれも間接的な表示である。つまり、表示される位置は、ソースプ

ログラムシンボル上から多少距離があり、直感性にかける。そこで、エディタ上で変数や定数の上でマウスカーソルをしばらく停止させるとその部分にも一時的にその図が表示されるようにする。

もちろんこれらの機能は編集時にも実行時にも提供される。

3.7 構文の学習支援

[2]と同様に、構文エラーの原因の理解、構文の理解のために機能を付加する。ただし、[2]とはその表現が異なる。[2]では BNF とその概念の学習を中心に置いたため、BNF を学習するように構文を BNF で表現するようにした。しかし、BNF は表現が記号的なため、わかりづらい面もあった。

今回はより直感的な構文図式の記法を使用することで、構文を理解しやすくする。

また、今回は編集カーソルのある位置に連動して構文が表示されるようにする。

3.8 計算量の学習支援

計算量には、時間計算量と空間計算量がある。どちらもアルゴリズムによりある程度見積もることができる。しかし、それを実測して確かめることが工学である。そこでそのための機能を提供する。これは、またプロファイルの機能を兼ねている。

時間計算量に関しては、1行を時間単位としたときの行の実行時間として提供する。図 3.1 ではそのためにエディタの左側に各行の実行回数の表示のための欄があいており、実行した回数が表示される。編集時にはプログラムカウンタが行き来した回数が記録される。

空間計算量に関しては、ヒープとスタックそれぞれの使用メモリサイズが時間単位ごとに計測される。これは、実行履歴のページでグラフとしてみることが可能である。また、時間計算量も実行履歴のページでグラフとして見ることが可能である。

4. おわりに

本稿では、プログラミング教育を行う上での、教育用処理系に対する要求を明確にし、編集時と実行時の両方で統一的に概念や構文の学習支援をすることができる言語処理系の設計について述べた。今後はこの処理系の実装を進めていく予定である。

謝辞

本研究は、科学研究費基盤研究(A)(2)09358004により行われた。

参考文献

[1]橋本裕,早川栄一,並木美太郎,高橋延匡: プログラミング教育用言語処理系 NB2 の設計,情報処理学会第 56 回全国大会 4L-5, 1998.3

[2]橋本裕, 早川栄一, 並木美太郎, 高橋延匡: BNF の学習を支援する Full BASIC 言語処理系, 情報処理学会コンピュータと教育研究会, Vol.39, No.3, pp.17-24, 1996.1

[3]Thomas L. Naps and Jeremy Stenglein: TOOLS FOR VISUAL EXPLORATION OF SCOPE AND PARAMETER PASSING IN A PROGRAMMING LANGUAGES COURCE, SIGCSE BULLETIN, Vol.28, No.1, pp.305-309, 1996.3