教育用プログラミング言語の設計と試作

93B548 小林 順一 93B551 小島 章嗣

93B552 駒田 高一

指導教員　　野呂昌満

1　はじめに

本稿では、教育用プログラミング言語の設計と試作について述べる。本研究の目的は、初心者プログラマを教育するために必要なプログラミング言語の機能を考察し、新しい教育用プログラミング言語を設計することである。

本学では、現在、教育用プログラミング言語として Pascal が使われている。しかし、 Pascal には以下のような問題点がある。

1. 大域変数の宣言ができる

   大域変数はプログラム中のどこからでも参照、変更することができてしまうので、副作用が原因となった誤りのあるプログラムを記述することがある。

2. プログラムの意味的な正しさを確認する手段がない

   プログラマはプログラムが正しく動くことを確認しなければならない。しかし、 Pascal はプログラミング言語レベルで、プログラムが正しく動くことを検証できる機能がない。

我々は、上記の問題点を解決するために、データ抽象の概念と形式的手法を教育用プログラミング言語に導入することを考え、初心者プログラマを教育するために必要な言語を設計した。

以下の要求を満たす教育用プログラミング言語を設計した。

- 手続き指向言語である。
- 強い型付けの言語である。
- データ抽象を支援する。
- 形式的手法を支援する。

我々が定義した構文に曖昧さがないことを確認するために、 UNIX で用意されているコンパイラ生成系である lex と yacc を使用して字句解析部、構文解析部の試作を行なった。その結果、構文の曖昧さがないことが確認できた。

2　教育用プログラミング言語としての Pascal

教育用プログラミング言語への要求を考えるにあたり、教育用プログラミング言語として使われている Pascal を調査した。

手続き指向パラダイム　フォンノイマン型の計算機上で、プログラミングをすることは、データとそれを処理するアルゴリズムを記述することである。手続き指向パラダイムに基づく言語はアルゴリズミックにプログラムを記述するための言語である。この観点からすると、 Pascal は手続き指向に基づく言語なので、初心者のプログラミングの教育に適した言語であるといえる。しかし、データアクセスの本質を考えたとき、データを特徴づけるのは、データ構造ではなく、データを扱う処理と考えられる。この考えに基づけば、データ構造とそれに対する処理がそなわっているデータ抽象の概念が必要であるが、 Pascal にはその概念がない。

制御構造　Pascal の制御構造には、連接、選択、繰り返しがある。 C のように、制御を中断する return 文や break 文を記述を認めていないので、各構文の出入口が一つになり、プログラムの実行の流れが容易に追跡できる。しかし、欠点として、 goto 文が記述できるので、プログラムの流れが容易に追跡できないプログラムを記述する危険がある。

データ構造　Pascal では、データ構造として、配列、列挙、レコード、ポインタを定義できる。これらを使って、プログラマは任意のデータ構造を構築できる利点がある。しかし、 Pascal では、データ構造を定義することがデータ型を定義することになり、前述した、データアクセスの本質に反する。

手続きと関数　Pascal は手続き・関数によって、プログラムを機能ごとに分割し、作成することができる。手続き・関数のブロック内で、変数や手続き・関数の宣言を局所化できる利点がある。しかし、複数の機能が共通のデータを取り扱う場合、それぞれの機能と共通のデータを一つのモジュールとしてひとまとめにすることができない欠点がある。

手続き・関数の呼び出しの際の仮引数の種類は、値引数と変数引数がある。値引数では、実引数の値を渡され、変数引数では、実引数のアドレスが渡される。値引数が基本である C では、変数引数を疑似的に行なう知識が必要となる。 Pascal では、そのような負担を初心者プログラマに与えない。

強い型付け　Pascal は、型が異なるデータどうしの代入や演算を認めない、強い型付け言語である。もし、異なる型どうしの代入や演算が行なわれた場合、コンパイル実行時に誤りとして報告されるので、実行時まで誤りをもちこさない。 Pascal には以上の利点があるが、整数と実数の混在する式を許す。可変レコードのタグの変更が実行時にできるので、コンパイル時に型検査できないという欠点がある。

3　教育用プログラミング言語に対する要求

既存の教育用プログラミング言語の調査結果をふまえ、教育用プログラミング言語としてどのような言語であればよいか、どのような要求があるかを考察した。教育用プログラミング言語を使用するのは、おもに初心者プログラマなので、以下の指針に留意して言語仕様を決定しなければならない。

- 初心者プログラマに対し、何を教育すべきかを考える
- どういう言語仕様であれば計算機やプログラミングの基礎的な知識が身につくかを考える
- プログラミングに関する知識の曖昧さや経験の未熟から生じる誤りをいかに正しい記述に導くかを考える

我々が考察した教育用プログラミング言語の特徴を以下に記述する。

1. 手続き指向パラダイムにもとづく言語である

   手続き指向パラダイムは、フォンノイマン型の計算機上で、プログラムを記述することに適している。
2. データ抽象を支援する

   データ抽象の概念は、データとそれに対する処理をもつというデータアクセスの本質に基づいている。
3. 強い型付けの言語である

   プログラムの誤りで多いのは、型の不一致によるものである。そのような誤りを実行時にもちこさないために、強い型付けが必要である。 Pascalでは、整数と実数の混在する式を認めたが、本言語では認めない。

以上で述べたことに加え、形式的手法（真野芳久ゼミナール担当）を教育用プログラミング言語には必要な要素であるとし教育用プログラミング言語に取り入れる。

4　言語仕様

本言語では、より少ない概念・機能を持つ言語である方が、初心者にとって理解が容易になると考え、必要最小限の概念・機能を用意する。

4.1　型

型とは、データとそれに付随する演算の集合と定義される。本言語では、処理系で用意する型と、型生成子によって生成される型を扱う。
基本型として、 **integer** 型、 **real** 型、を処理系で用意する。また、型生成子として列挙 **(enum)**、部分範囲 **(subrange)**、配列 **(array)**、ポインタ **(pointer)**、クラス **(class)** を用意する。型生成子によって生成された型は、構文要素 **type** によって型名をつける。さらに、 **character** 型、 **boolean** 型を列挙型で定義し、ライブラリとして用意する。

列挙型で character 型、 boolean 型を定義する理由
character 型、 boolean 型はとりうる要素の範囲がすでに決まっているので、列挙型の概念を用いればそれを表現することは可能である。列挙型の概念を用いることで表現できるのであれば、より少ない、共通の概念で異なる型を記述できることになるからである。

型生成子　型生成子とは、必要な要素を指定することでその要素に対応した型を生成する構文要素のことである。我々は以下のように考え、列挙、部分範囲、配列、ポインタ、クラスを型生成子とした。

- **列挙** 列挙型を生成する場合、その型がとりうるすべての要素 (列挙定数) を、プログラマが指定しなければならない。列挙型に関する演算は、あらかじめ以下のものを用意することにした。
  - **order()**(値の順序数を返す)
  - **successor()**(一つ後の値を返す)
  - **predecessor()**(一つ前の値を返す)
  - 関係演算子 (<, <=, =, <>, >=, >)

  要素を指定するだけで型が生成されるので、列挙型は型生成子によって生成される型であると考えた。
- **部分範囲** 部分範囲型を生成する場合、親の型と要素範囲をプログラマが指定しなければならない。部分範囲型に関する演算は、親の型に用意された演算を引き継ぐ。親の型と要素範囲を指定することで型が生成されるので、部分範囲型は型生成子によって生成される型であると考えた。
- **配列** 配列型を生成する場合、配列要素の型と部分範囲型をプログラマが指定しなければならない。配列型に関する演算は、添字による要素の参照 '[ ]' をあらかじめ用意することにした。
  要素の型と要素数を指定することで型が生成されるので、配列型は型生成子によって生成される型であると考えた。
- **ポインタ** ポインタ型を生成する場合、それが指す先のデータの型 (被指示型) をプログラマが指定しなければならない。ポインタ型に関する演算は、あらかじめ以下のものを用意することにした。
  - 間接参照 '^'
  - 等 (=)、不等 (<>) の関係演算子
  - **new**

被指示型を指定することで型が生成されるので、ポインタ型は型生成子によって生成される型であると考えた。

- クラス クラスを生成する場合、クラスの内部データとそれに対する操作 (メソッド) をプログラマが指定しなければならない。必要な内部データと操作を指定することで型が生成されるので、クラスは型生成子によって生成される型であると考えた。

記述方法 本言語では、構文要素 **type** を用いて、型生成子によって生成された型に名前 (識別子) をつけるものとする。列挙型、部分範囲型、配列型、ポインタ型、クラスはそれぞれ **enum**、**subrange**、**array**、**pointer**、**class** という型生成子によって生成される。以下にそれぞれの記述方法を示す。

- 列挙型の記述方法
  type 列挙名 **is enum**< 列挙定数, 列挙定数, ...>
- 部分範囲型の記述方法
  type 部分範囲名 **is subrange**< 親の型名, 要素範囲 >
  要素範囲は、下限 **..** 上限と記述する。
- 配列型の記述方法
  type 配列名 **is array**< 要素の型名, 部分範囲型名 >;
- ポインタ型の記述方法
  type ポインタ名 **is pointer**< 型名 >;
- クラスの記述方法

```
/* 仕様部 */
type クラス名 is class
  private :
    データ宣言欄
  public :
    操作宣言欄
end クラス名;
/* 実装部 */
type implementation クラス名 is class
  操作実装欄
end クラス名;
```

型変換 本言語は強い型付け言語であるので、異なる型との混合演算は認めない。しかし、次の場合に限り変数の前に'(' 型名 ')'と記述し、その型への明示的な型変換を行なわなければならない。

- integer 型 ↔ real 型
- 部分範囲型 ↔ 親の型
- (列挙定数が多重定義された) 列挙型 ↔ 列挙型

4.2 制御構造

本言語では、制御構造として、連接、選択、繰り返しを持つ。できる限り必要な構文要素の数を減らし、初心者プログラマが容易に理解できることを優先してるので、選択として if 文、繰り返しとして loop 文のみを用意する。以下に記述方法を示す。

- if 文の記述方法
  if 論理式 **then** 文並び **end if**;
  if 論理式 **then** 文並び **else** 文並び **end if**;
- loop 文の記述方法
  loop ループ不変式 **when** 条件式 **exit**
  文並び
  end loop;

4.3 データ抽象の支援

本言語では、クラスと呼ばれる抽象データ型を用いてデータ抽象を支援する。抽象データ型の記述には、Ada に代表される手続き指向言語、C++ に代表されるオブジェクト指向言語による記述を考えた。我々は、一つの構文要素 **class** を用いて抽象化・型定義を同時に表すことができ、抽象データ型の簡潔な記述が可能となる後者を選択した。
以下に、参考とした Ada と C++ における抽象データ型の記述方法の例を示す。

- Ada

```
package パッケージ名 is
  type 型名 is
    .
    .
end パッケージ名;
```

  構文要素 **package** を用いて抽象化を表し、構文要素 **type** を用いて型定義を表している。

- C++

```
class クラス名 {
  .
  .
};
```

  一つの構文要素 **class** を用いて抽象化・型定義を同時に表している。

クラスの記述方法 クラスは仕様部と実装部にわけて記述する。仕様部では、**private** 部に内部データを、**public** 部に特定の操作の宣言を行なう (図 1参照)。実装部では操作の実装を行なう。特定の操作とはメソッド (手続き・関数) であり、内部データのないクラス、メソッドのないクラスは記述できない。

データ宣言欄には、 **use** 指定、変数、定数、型の宣言が含まれる。クラスのメソッドは、そのクラスの初期化手続き、またはクラスに局所的な手続き・関数である。

```
/* 仕様部 */
type Word is class
 private :
  constant Table_Size = 257;
  type Str is array<character,1..Str_Size>;
  variable s is Str;
           hash_val is integer;
 public :
  procedure init(in d is data);
  function get_hash() : integer;
end Word;
/* 実装部 */
type implementation Word is class
  procedure init(in d is data) is
                              /* 初期化手続き */
    variable i,j,x is integer;
  begin
    i := 1; x := 0;
    loop when i > Str_size exit
      s[i] := d[i];
      x := (Table_Size-1)*x + (integer)d[i];
      i := i + 1;
    end loop;
   ...
  end init;

  function get_hash() : integer is
  begin
    return hash_val;
  end get_hash;
end Word;
```

図 1: class の記述例

アクセス制御　クラスの内部データへのアクセスは、同じクラスに記述されたメソッドに限定される。メソッドを介してのみ内部データへアクセスすることで情報隠蔽を実現する (図 2参照)。

クラスオブジェクトの領域確保・初期設定　クラスオブジェクトは、それが宣言 (領域確保) された後に、その使用に先行して初期設定されなければならない。 C++ では、コンストラクタを定義することにより、クラスオブジェクトの初期設定をクラスオブジェクトの宣言時に自動的に行なっている。

```
variable W is Word;
          /* クラス Word 型の変数 W の宣言 */
           h is integer;
 ...
h:=W.get_hash();
/* クラス Word の関数 get_hash() の呼び出し */
```

図 2: アクセス制御

本言語においても、**初期化手続き init()**(名前は固定) を記述できるようにし、 C++ におけるコンストラクタの初期化機能と同等の機能を提供する。
C++ のコンストラクタには、利点として以下の点が挙げられる。

- 動的 (**new** 演算子を使用)・静的なクラスオブジェクトの生成に共通して、コンストラクタが自動的に呼び出される。コンストラクタの自動呼び出しによって、クラスオブジェクトの初期化を忘れることがなくなる。

この利点を考慮し、静的・静的なクラスオブジェクトの生成にさいし、初期化手続きの**自動的な呼び出し**を行なう。

クラスオブジェクトの領域解放　本言語では、独立したクラス間のポインタの受渡しを禁止しているので、複数のクラスオブジェクトから同じ領域を指すポインタが存在しない。よって、複数のクラスオブジェクトによる領域の共有がなくなり、クラスオブジェクトが必要なくなった時点でその領域を解放することが安全に行なえる。

必要なくなったクラスオブジェクトの領域を解放することは、メモリの節約のために必要である。しかし、我々は、メモリ管理をプログラマにさせる必要はないと考えた。クラスオブジェクトが必要なくなる場合を認識させること、メモリ管理を教育するような言語仕様にすることは、本研究の目的ではないからである。

以上のことから、本言語では、使わなくなった領域を自動的に解放する**ゴミ集め機能**を処理系で提供することにした。

入れ子クラス　入れ子クラスの記述を可能とする。入れ子クラスとは、クラス内部に定義されたクラスのことである。なお、入れ子クラスに対するアクセス制御は、前述と同様である。

入れ子クラスの記述は、クラス構造の複雑化という欠点を招く可能性がある。しかし、あるクラスに局所的なクラス

を定義することで、他のクラスから見えないようにできるので、入れ子クラスの記述を可能とすることにした。

4.4 手続き・関数

手続き・関数はクラスのメソッド、およびメソッドに局所的な手続き・関数として記述できる。

記述方法 以下に、手続き・関数の記述方法を示す (図3、4参照)。

```
procedure 手続き名 (仮引数宣言) is
  宣言部
begin
 ...
end 手続き名;
```

図 3: 手続きの記述方法

```
function 関数名 (仮引数宣言) : 戻り値の型名 is
  宣言部
begin
 ...
end 関数名;
```

図 4: 関数の記述方法

宣言部は、 **begin** の前に記述しなければならない。宣言部では、 **use** 指定、定数宣言、変数宣言、手続き・関数宣言を行なうことができる。これらは宣言された手続き・関数内に局所的である。

仮引数の種類 仮引数は、 **in**、 **out**、 **inout** の三つの指定子のうちのいずれかを持つものとする。

- **in** 指定の仮引数には、実引数の値が与えられる。仮引数の値の変更は、実引数の値に影響しない。
- **out** 指定の仮引数には、実引数から値が与えられない。手続き内で与えられた仮引数の値が、実引数に与えられる。
- **inout** 指定の仮引数には、実引数の値が与えられる。仮引数の値の変更は、実引数の値に影響する。

in 指定は、値渡しに相当する。 **in** 指定のみでもプログラムを記述できるが、 C のように値渡しのみとしなかったのは、以下のように考えたからである。

- C では、 C++ などの参照渡しと同等の処理を行ないたい場合、アドレス、ポインタを用いて疑似的な参照渡しをしなければならない。この操作は、プログラミングの本質とは関係ないことであると考え、参照渡しに相当する **inout** 指定が必要であると考えた。
- 配列の要素を取り出すような手続きの記述を想定した場合、仮引数を **out** 指定した手続きの記述が考えられる。配列の要素をを返すような関数を記述することで、同等の処理が可能となるが、これは手続き指向的な考えに基づいた記述ではないと考え、 **out** 指定が必要であると考えた。

手続き・関数の多重定義 我々は、つぎのような理由から、類似した処理を行なう手続き・関数を多重定義できることにした。
手続き・関数に同じ名前をつけることで、類似した処理を行なうということが明確になり、プログラムが読みやすくなる場合があると考えた。

多重定義された手続き・関数の判別は、引数の数や型、種類 (指定子) の違いによって行なう。

大域的宣言の禁止 どのクラスにも属さない大域的手続き・関数による不正なデータの使用が発生し、プログラムの保守性が低下する可能性があるので、手続き・関数の大域的宣言を禁止する。

4.5 分割コンパイル

本言語では、つぎのような理由から、分割コンパイル機能を取り入れる。

プログラムを作成するさい、必要な機能をあらかじめコンパイルし、それを取り込むだけでその機能が使用できれば、プログラマの労力は軽減される。分割コンパイルは、大規模言語プログラミングにおいての作業分担のさいに必要なので、分割コンパイルについての知識を教育することは必要であると考えた。

コンパイル方法 本言語では、誤りをできる限り多くコンパイル時に発見することが必要であると考え、分離翻訳で分割コンパイルを行なう。

本言語における分割コンパイルの概要 以下に、分割コンパイル機能の概要を記述する。

- コンパイル単位は、 **program** と **type** である。
- 異なるファイルに実体のある型を使用するには、

  ```
  use type 宣言された型名;
  ```

  と記述する。 **use** 指定する場所は、主プログラムと手続き・関数の宣言部の先頭、およびクラスの内部データ宣言欄の先頭とする。

- クラスは仕様部、実装部の順にコンパイルし、 **use** 指定を伴うコンパイル単位は **use** 文に記述された型からコンパイルしなければならない。

  クラスの仕様部、 **use** 指定された型を先にコンパイルすることで、コンパイラは依存関係にある型情報の整合性を検査することができる。
  また、クラスの仕様部に変更がない限り、実装部に変更を加えても実装部のみを再コンパイルすれば良い。この場合、依存関係にあるすべてのコンパイル単位を再コンパイルする必要はない。

- クラスは仕様部と実装部にわけてコンパイルするので、相互参照が可能となる。

- クラスの仕様部には、クラスのサイズに関するすべての情報を記述してあるので、実装部をコンパイルするさいに必要に応じてクラスの静的・動的な領域確保が可能となる。

4.6 入出力

本言語は、標準入出力とファイル入出力を行なえるように設計する。その実現方法は、入出力機能を持つ **Stream** クラスをライブラリとして用意し、使用者は、 Stream クラスの変数を宣言した後、 **standard**(標準入出力)、あるいはファイル名 (ファイル入出力) を指定して標準、あるいはファイル入出力に切替える。図 5は、標準入出力の場合の例である。

```
variable St is Stream;
        /* Stream クラスの実体を生成 */
  ...
St.open(standard);  /* 標準入出力に切替え */
St.read(...);   /* 入力 */
St.write(...);  /* 出力 */
St.close(standard);
```

図 5: 標準入出力

標準入出力であっても **open**、 **close** というメッセージを送らなければならない。ファイル入出力の場合は、 **standard** の代わりにファイル名を指定する。

5 字句解析部と構文解析部の試作

現段階において定義できている構文規則をもとに、字句解析部、構文解析部の試作までを行なった。そして、構文の曖昧さがないことを確認した。

構文解析部を yacc で記述した結果、構文規則が 150、終端記号が 62、非終端記号が 60、衝突の報告が 0 となった。この結果、構文の曖昧さがないことが確認できた。

6 おわりに

本研究では、 Pascal に代わる新しい教育用プログラミング言語の設計、および構文定義を行なった。

本年度では、教育用プログラミング言語の言語仕様、および構文規則を決定し、構文に誤りがないことが確認できた。また、本研究で設計した教育用プログラミング言語を使用することにより、データ抽象、および形式的手法を取り入れたプログラミングを教育できると期待される。

今後の課題としては、意味解析部、中間形生成部、分割コンパイル機能、コード生成部の設計と試作が挙げられる。

謝辞

本研究を進めるにあたり、二年間御指導いただきました野呂昌満助教授、そして、有益なアドバイスをいただきました大学院の塩田康隆さん、山野篤さんに深く感謝致します。

参 考 文 献

[1] Bjarne Stroustrup:*The C++ Programing Language Second Edition*,Addison-Wesle y Publishing Company,(1991).

[2] 石塚圭樹, 横手靖彦:「改訂新版 オブジェクト指向プログラミング」, アスキー出版局,(1993).

[3] Part-I: Henry Ledgard:*ADA:An Introduction by Henry Ledgard*,Springer-Verlag,(1981), Part-II: *Ada Reference'Manual*,(1980): 上條史彦, 細谷僚一 永瀬淳夫, 石原憲一訳, 「Ada 入門／和訳規約」, (1983).

[4] Daniel F.Stubbs,Neil W.Webre:*Data Structures with Abstract Data Types and Pascal(second edition)*,Brooks/Cole,(1989): 小山裕徳訳, 「Pascal によるデータ構造」, オーム社,(1994).

[5] Alfred V.Aho,Ravi Sethi,Jeffrey D.Ullman: *Compilers Principles,Techniques,and Tools*,Addison-Wesley Publishing Company,(1986): 原田賢一訳, コンパイラ I, サイエンス社,(1990).

[6] John W.L.Ogikvie:*MODULA-2 PROGRAMING*, McGraw-Hill Inc,(1985): 中村和郎訳, 「MODULA-2 プログラミング」,(1986).