Rust を用いたプログラミング教育について

箕原辰夫
Email: *minohara@cuc.ac.jp*

千葉商科大学政策情報学部政策情報学科

◎Key Words　プログラミング教育，Rust，コンパイラ，オブジェクト指向プログラミング言語，C/C++

1. はじめに

C/C++言語の後継と目されている、オープンソースのコミュニティベースで開発されている Rust プログラミング言語[1]が昨今注目されている。コンパイラ型の言語で、C 言語よりも高速に実行される場合も計測されている。Rust を用いたオペレーティングシステムや組込みシステムのコーディングも盛んに行なわれている。Rust は、C/C++と比べて、メモリ管理が厳格であり、実行時の安全性に優れているからである。しかしながら、そのために、プログラムの文法が非常にわかりにくくなっている面も否めない。プログラム初心者の教育には向かない。C/C++あるいは Java での教育を経た後に、組込みシステムやオペレーティングシステムに特化した教育場面で使うべきである。Rust には、Python の Python Tutor[2]と同様に、Web 上でも実行できるインタープリタ（Rust Playground[3]）も開発されている。VSCode[4]上での Rust の開発および Rust Playground を用いて、Rust 言語をゼミナールで採り上げて、2022 年度春学期に授業を展開した結果について報告する。

2. 各回の授業内容の構成

2.1 授業内容

最初は、基本的な式から始まり、ターミナルとの入出力、制御構文に行く形になるが、ここまでにおいても、独特の記法がでてくるので、Rust を熟知していないと難航することになる。C 言語の後継なので、プログラムは main 関数の定義から始まる。出力するのが、関数呼出しではなく、マクロ呼出しになるので、print(), printf()等の関数呼出しの記述法ではなく、println!();として、!マークをつけなければならない。マクロの文法的な詳細は、C 言語をかなり熟知した学生しかわからないだろう。単なる約束として進むしかない。入力については、悲惨で、プロンプトを出して入力を促す Python の input( )関数に該当するものがなく、かなり面倒な記述をしなければならない。Java も最初は同じ状態で、後から Scanner クラスが導入されたのと同じような経緯を辿るのではないかと思われる。たぶん、マクロとして C 言語の scanf に該当するものが登場するのではないかと推測される。

各回の授業内容は表1のようになっている。実際には、各回の内容が前後にずれているが、主に扱った内容として記述した。参照した文献に従った結果、かなり初期に関数定義を導入しているのが、これまでの新しい言語を扱った授業内容とは異なっている部分である。Swift や Julia と同様に Option 型が導入されており、また switch 文の後継である複雑な match 文（この頃は Python 3.10 以降にも導入されている）もお約束の如く、導入されている。

表 1 授業内容（第 1 回～第 13 回）

授業回	実際に授業で扱った内容
第 1 回	開発環境の導入と最初のプログラム
第 2 回	Rust の型とリテラル・構造的なリテラル
第 3 回	変数への代入と型指定を伴う初期値代入
第 4 回	ターミナルからの入力、型変換
第 5 回	関数定義と呼出し、局所変数、ブロック
第 6 回	フォーマット文字列の指定
第 7 回	if 文による条件分岐、loop 文による繰返し
第 8 回	while, for 文による繰返し・その他の制御文
第 9 回	match 文による条件分岐
第 10 回	Option 型の導入、if let 文・while let 文
第 11 回	クロージャ、ジェネリック型
第 12 回	構造体、メソッドの定義、enum 型
第 13 回	クレート、GTK の導入

2.2 参考とした文献・Web サイト

Rust のコミュニティでは、日本語訳も含めた積極的なドキュメントが制作されており、入門書の「The Rust Programming Language[5]」および、少し詳しく解説されている「Rust by Example[6]」などが標準の参考文献、あるいは教科書して使用することができる。また、公式のリファレンスとしては、「Crate std[7]」があるが、こちらの日本語訳は、Rust by Example の一部として出されている模様である。文法の詳細なリファレンス[8]を初期には見つけられなかったため、毎回の授業では試行錯誤することが多かった。また、それでもわからないところは、日本では Qiita (quiita.com) や Stack overflow (stackoverflow.com) などに追うところが大きい。

Rust のライブラリについては、クレート（crate）という形でまとめられており、それを扱った総合的なサイト（crate.io）を参照した。

3. Rust 特有の文法規則

ここでは、間違って Rust をプログラミング教育に導入しなければならなくなった教員のために、Rust 特有の文法的な特徴（嵌まりポイントとも言える）をいくつか紹介する。これら以外にも、学生にとってはかなり嵌まるポイントがあると思われる。

3.1 変数への代入・書換え

Rust のメモリ管理の安全性、不要なメモリの早期の廃棄について考え方が現れているのが、変数への代入である。let で導入された変数は、局所変数用のスタック上にメモリが確保されて、そこにオブジェクトへの参照や値が置かれるのであるが、再度同じ変数に let で代入された場合は、その変数が持っていたメモリは一旦解放されて、新しく変数へのメモリが割り当てられて、上書きされる形になる。変数が持つ値を、直接書き換えるためには、let mut（mutable の略）で変数が導入されなければならない。

```
let x = 32;         // 変更不可能な変数の導入
let x = x + 24;     // 再度の let で上書きされる
let mut y = 32;     // 変更可能な変数の導入
y = y + 24;         // 通常のような代入ができる
```

同じことは構造体についても適用される。たとえば、以下のような構造体を導入した場合、let 文で導入された場合は、初期値代入後は書き換えができないので、中のメンバーの値を書き換えたい場合は、let mut 文で導入する必要がある。

```
struct Point { x: i64, y: i64, }
fn main( ) {
    let mut p = Point { x: 34, y: 89 };
    p.x = 12; // let mut の場合は書換え可能
}
```

3.2 オブジェクトの移動

Python の場合、関数のパラメータとしてオブジェクトを渡す場合（リストなどの構造的リテラルも含む）は、基本的には参照渡しになっており、リストの中の要素などを置き換えることは可能になっている。これと対照的なのが、Java の配列であり、Java で引数として関数に配列を渡すと、すべてコピーされる形になっている。Rust の場合、どちらの形でも採れるように、参照渡しの場合には、C++と同様に&をつけて構造体等のオブジェクトを渡す形になるが、値渡しにした場合は、関数側にオブジェクトが移動して、呼出し側からは、使えない状態になる。

これもメモリ運用の効率化という目的に合致した取組みに起因しているものと思われる。オブジェクトのために確保したメモリを、そのまま関数側で運用するために呼出し側からは削除されることになっている。なお、当然のごとく、受け渡されて移動してきた構造物の中身（構造体のメンバー）を書き換えるときは、mut を付けておく必要がある。呼出し側で移動されて使えなくなったオブジェクトを再度利用するためには、関数側で、return で受け渡されたオブジェクトを再度戻して、呼出し側で上書きする必要がある。

3.3 不統一な制御構文の記法

画期的と思われるのは、関数定義で return なども記述しなくても、ブロックの最後に実行された式が、そのブロックの評価値として残ることである。これは、関数定義以外のブロックでも同じことが起こる。Rust では、ブロックは複数の文をまとめるものではなく、複数の文や式を含む評価されるべき式としての役割をもっている。ただし、関数定義においては、return があった方がわかりやすいのではないだろうか。

```
fn cube( x : i64 ) -> i64 { x * x * x; }
```

面白いのは、loop 文による繰返しの中で break 文を使ってブロックから抜け出すときに、関数定義の中の return 文と同様に、break 文の後に式を記述し、値を返すことが可能になっている。ところが for 文や while 文による繰返しの中では、break 文の後に式を記述することができない。この不統一性は、将来の言語拡張がされるときに解消されるのではないかとも思われる。

```
let mut n = 1;
let square_under_hundred = loop {
    if n * n > 100 { break n*n; }
    n = n + 1;
};
```

match 文は、switch 文の拡張で複雑なパターンマッチングができる条件分岐ができる制御構文になっているが、なぜか、分岐の最後をカンマ (,) で終了する必要がある。match 文自体はブロックの後にセミコロンで終わる必要がある。これは、match 文が文というよりも式であることに起因している。

```
for n in 1..20 {
  match n {
    2 | 3 | 5 | 7 | 11 | 13 | 17 | 19 =>
        println!( "{} is prime number", n ),
    _ => (),
  };
}
```

3.4 ブロックによるクロージャ

関数の引数の型を動的に変更できる方式として、C++/Java のようなテンプレートを用いたジェネリック型以外に、ブロックによるクロージャを用いることができる。このクロージャは、||で囲むことによって、仮引数の変数を導入することができる。クロージャの評価値は、例によって、ブロックの最後に書かれている式に依存する。

```
println!( "{}", {|x| x * x}( 12 ) );
println!( "{}", {|x| x * x}( 34.7 ) );
```

しかしながら、一旦クロージャを何らかの変数に割り当てて名前付きにしてしまうと、「最初に呼出しされて割り当てられた引数に対しての型」に固定されてしまう。そのため、別の型を引数にして呼び出すとエラーになってしまう。コンパイラ言語の宿命というものであろうか。

```
let square = |x| x * x;
println!( "{}", square( 12.2 ) ); // 実数用として固定
```

3.5 Option 型

Swift 以降の言語に特徴なのがオプショナル型である。値が入っているとき、入っていないときを区別して使うために、通常の型にラップして掛けるものになっている。過年度紹介した Swift や Julia では、ラップ・アンラップのための演算子があるが、Rust では型指定と Some 関数を使ってそれらを代用している。また、Rust では、Option 型と Result 型（失敗したときに使う）の 2 つが用意されており、更に使い分けが強要されている。

```
fn squareOp( a: i64 ) -> Option<i64> {
    return (a > 10) ? Some( a * a ) : None;
}
fn main( ) {
  if let Some( v ) = squareOp ( 7 ) {
    println!( "Found Seven at {}", v );
  }
  let value = 22;
  match squareOp( value ) {
   Some( v ) => println!( "square of {} is {}", value, v ),
    None => println!( "{} is not greater than 10", value ),
  }
}
```

Result 型では、Some と None の代わりに、Ok と Err を使う形になる。

```
fn input_int( prompt: &str )
  -> Result<i64, <i64 as FromStr>::Err> {
    print!( "{}", prompt );
    stdout().flush().unwrap();
    let mut buffer = String::new();
    let stdin = io::stdin();
    stdin.read_line(&mut buffer).expect( "入力失敗" );
    let buffer = buffer.trim();
    return String::from( buffer ).parse()
}
fn main() {
    match input_int( "整数を入力:" ) {
      Ok( result ) => println!( "{}", result ),
      Err( msg ) => println!( "変換エラー{}", msg ),
}
```

3.6 構造体によるクラスの定義

構造体の定義は前出であるが、これも match 文が式であるのと同様の考え方で、最後はカンマ（,）で終わる必要がある。また、構造体をオブジェクトとして捉えて、メソッドを追加する場合は、impl という名前のついた別ブロックを構成する必要がある。これは、仕様と実装の切り分けを意図したものと考えられるが、Swift のクラスの外付けの拡張と同様に、ライブラリの構成の仕方によっては、わかりにくくなる可能性がある。

```
struct Point { x: i64, y: i64, }
impl Point {
    fn move_point( &mut self, dx:i64, dy: i64 ) {
      self.x += dx;
      self.y += dy;
    }
}
```

3.7 静的な型推論

いくつかの例でみてきたように、let 文で代入される変数やクロージャに関数名をつけた場合は、初期値で代入された値によって（クロージャの場合は最初呼ばれた引数の値)、その変数は何型であるかが型推論される。これは、コンパイル時に推論されるので、静的な型推論と考えることができる。インタープリタ言語との明確な違いになっており、Rust の特徴であるとも捉えることができる。なお、let 文で代入するときに、いちいち型を指定することも可能になっている。関数定義の際には、引数や戻り値の型を指定する必要があるので、これは従来のコンパイラ型の言語と同様になっている。

3.8 他の言語との比較

Rust と最近の他の言語とをプログラミング教育という観点から比較してみたい。対象となるのは、Python, Julia, Swift, JavaScript である。以下にそれぞれの言語の用途を比較してみたが、C/C++/C#/Java 言語等のコンパイラベースの言語と違うのは、それぞれ型推論の機構が組み込まれているところである。型推論の機構がないプログラミング言語では、せいぜい、ジェネリック型ぐらいでしか

対応ができない。ただし、Rust に特徴的なのは、型推論が、動的ではなくて、コンパイル時に静的に行なわれるところである。それが、他の型推論機構を持つ言語と一線を画している。

表2 各言語の比較

言語	主な用途	型推論
Python	汎用・プログラミング教育	動的
Julia	数値計算	動的
Swift	アプリケーション開発	動的
JavaScript	Web アプリケーション	動的
Rust	組込みシステム・OS 開発	静的

4. Rust の開発環境の設定

4.1 開発環境の設定

Web サイトから Rust の開発環境をダウンロードすると、開発環境設定用の rustup コマンドと Rust コンパイラの rustc コマンドが使えるようになる。

また、プロジェクトパッケージ開発のための管理システムとして、cargo というコマンドが用意されており、プロジェクト開発用の環境も整っている。しかしながら、授業で扱うように、1 ファイルで 1 プログラムの場合には、プロジェクト開発のフォルダをいちいち作って、設定を行なうのは面倒である。Python でも PyCharm などがこの方式を採用している。授業では、ターミナルからコンパイラの rustc を用いてコンパイルを行ない、プログラムの実行をそのままターミナルから行なう方式（C 言語と同じ）を採用した。

開発環境の設定が面倒な場合は、標準ライブラリだけが利用できる Rust Playground[3]の開発環境があるため、Web 上で記述、実行まで行ない、コピー&ペーストでファイルに保存するという方式を採ることもできる。できれば、Rust Playground には、ファイルのロード・セーブができるようなボタンの追加が欲しいところである。

なお、外部ライブラリを利用する場合は、必然的に前出の cargo と呼ばれるプロジェクト開発用の環境を利用する必要がある。

4.2 VSCode 上での Rust プログラムの実行

VSCode では、プロジェクト開発主体に機能拡張が用意されている。VSCode 上でも単なる個別のファイルのコンパイル・実行を行なうには、プロジェクト開発の部分（cargo のための設定）を使わずに、ターミナルから、コンパイルして、実行をさせる必要がある。VSCode 上では、プロジェクト開発の設定をして、main 関数が 1 つしかない場合は、main 関数のところに、「Run, Debug」のためのアイコンが現れるので、コンパイル・実行を連続的に行なうことができる。

4.3 外部ライブラリの利用

Python の pip によるインストールと同様に、rustup コマンドによって、外部ライブラリ（Rust ではクレート：Crate と呼ばれている）をインストールできるが、pip コマンドと異なり、残念なことに、外部ライブラリのインストールは、rustup コマンドだけで終わらないものもある。授業で導入した GTK[9]は、GUI アプリケーションを作成するためのライブラリであり、過年度 Julia 上で使ってみて、安定して使えるものであったので、Rust 上でも導入を試みたが、Windows 版では、GTK の本体も合わせてインストールする必要がある。そのためには、MSYS2[9]を利用して、mingw-w64[9]ベースの gcc（Gnu C Compiler）をインストールする必要があった。Unix ベースでの開発環境（Mac OS や Linux）を利用した方が、Rust の場合は外部ライブラリを利用した開発環境構築が楽なのではないかと思われた。

5. おわりに

プログラミング初等教育として Rust を採用するのは、学生が混乱をするので、やめた方が良い。順当に、Python などのインタープリタベースの言語や、コンパイラベースでの言語では、従来からの C 言語、あるいは Java 言語を使うべきではないかと思われる。Rust は、最初のプログラミング言語を学んだ後、特定の用途、メモリが限られていて、厳密にメモリ管理をしなければならない組込みシステムや、メモリの利用の安全性が保証されなければならない（ポインタがおかしな場所を指さない）オペレーティングシステムの開発などを扱う場合に、導入するべきではないかと思われる。ただ、組込みシステムでも MicroPython が動いており、普及しているため、用途はオペレーティングシステムの開発に限られるのかも知れない。ただし、C/C++言語や Unix などに熟練した学生であれば、汎用のプログラミング開発言語として、学ぶ価値はあるだろう。

参考文献

(1) “Rust Programming Language,” Rust Foundation, https://www.rust-lang.org/ja
(2) “Python Tutor,” https://pythontutor.com/
(3) “Rust Playground,” https://play.rust-lang.org/
(4) “Visual Studio Code,” https://code.visualstudio.com/
(5) “The Rust Programming Language,” Steve Klabnik, Carol Nichols, https://doc.rust-lang.org/book/, 日本語訳は https://doc.rust-jp.rs/book-ja/
(6) “Rust by Example,” https://doc.rust-lang.org/stable/rust-by-example/, 日本語訳は https://doc.rust-jp.rs/rust-by-example-ja/
(7) “Crate std,” Version 1.62.0, https://doc.rust-lang.org/std/
(8) “The Rust reference,” https://doc.rust-lang.org/reference/index.html
(9) “GTK,” GNOME Foundation, https://www.gtk.org/
(10) “MSYS2,” https://www.msys2.org/
(11) “MinGW-w64,” https://www.mingw-w64.org/