컴파일러 기말과제

소프트웨어학부, 20163091 김민주

5가지 언어 Go, Rust, Scala, Kotlin, Ruby에 대해 알아볼 수 있는 기회가 되었고, 각 언어의 Syntax와 특징에 대해 조사하였고, 간단한 코드(num이라는 정수형 변수에 5값을 집어넣고 hello, wolrd를 출혁하는 코드)를 만들었습니다.

1. Go

코드

```
package main
import   “fmt”

func main(){
        var num int
        num = 5
        fmt.Println( “hello, world.” )
}
```

Syntax

```
Production  = production_name "=" [ Expression ] "." .
Expression  = Alternative { "|" Alternative } .
Alternative = Term { Term } .
Term        = production_name | token [ "…" token ] | Group | Option | Repetition .
Group       = "(" Expression ")" .
Option      = "[" Expression "]" .
Repetition  = "{" Expression "}" .
```

특징

중괄호를 입력한 후에 개행해야한다.
c++과 java와 달리 ;을 사용할 필요가 없지만 한 줄에 여러 코드를 쓴다면
;을 사용해 각 코드를 구분해 주어야 한다.
c++과 java와 같이 들여쓰기로 tab을 사용한다.
Python과 같이 한 함수가 여러 리턴값을 줄 수 있다.
변수들의 타입을 따로 지정해주지 않아도 되고 함수의 파라미터에서 변수의 타입을 적을 때 변수명 뒤에 적는다.
C와 C++과 같이 포인터가 존재한다.
C++, JAVA와 달리 for문에서 조건을 맡을 부분에 괄호()가 필요하지 않고 {}는 필요하다.
import시 한번에 여러 패키지를 import할 수 있다.
예

```
import(
          “fmt”
          “math/rand”
)
```

2. Rust

코드

```
fn main() {
        let num = 5;
        println!("hello, world.");
}
```

Syntax

```
<postal-address> ::= <name-part> <street-address> <zip-part>

        <name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL>
                | <personal-part> <name-part>

<personal-part> ::= <initial> "." | <first-name>

<street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

        <zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""
        <opt-apt-num> ::= <apt-num> | ""
```

특징

모듈의 이름은 C++과 같이 ::으로 구분된다.
if문과 while같은 경우 사용할 문장이 1문장일 때 다른 언어의 경우 중괄호를 하지 않고 들여쓰기를 하기도 하는데 Rust의 경우 무조건 중괄호를 사용해야한다.
지역변수를 나타내기 위해 let을 사용하고 만일 가변적인 지역변수를 나타내려면 let mut을 사용한다.
반복문에서 c++의 continue의 역할을 Rust에선 loop가 맡는다.
C언어와 같이 구조체가 존재하고, 빌린 포인터라는 것이 존재한다.
매크로의 경우 뒤에 !을 붙인다.

3. Scala

코드

```
object Simple extends App {
        val num = 5
        println( “hello, world.” )
}
```

Syntax

```
UnicodeEscape ::=  '\' 'u' { 'u' } hexDigit hexDigit hexDigit hexDigit
hexDigit      ::=  '0' | … | '9' | 'A' | … | 'F' | 'a' | … | 'f'

whiteSpace    ::=  '\u0020' | '\u0009' | '\u000D' | '\u000A'
upper         ::=  'A' | … | 'Z' | '$' | '_'  // and Unicode category Lu
lower         ::=  'a' | … | 'z'  // and Unicode category Ll
letter        ::=  upper | lower // and Unicode categories Lo, Lt, Nl
digit         ::=  '0' | … | '9'
paren         ::=  '(' | ')' | '[' | ']' | '{' | '}'
delim         ::=  '`' | '"' | '"' | '.' | ';' | ','
opchar        ::= // printableChar not matched by (whiteSpace | upper | lower |
                  // letter | digit | paren | delim | opchar | Unicode_Sm | Unicode_So)
printableChar ::= // all characters in [\u0020, \u007F] inclusive
charEscapeSeq ::=  '\' ( 'b' | 't' | 'n' | 'f' | 'r' | '"' | ''' | '\' )

op            ::=  opchar {opchar}
varid         ::=  lower idrest
plainid       ::=  upper idrest
                |  varid
                |  op
id            ::=  plainid
                |  '`' stringLiteral '`'
idrest        ::=  {letter | digit} [ '_' op]

integerLiteral    ::=  (decimalNumeral | hexNumeral) [ 'L' | 'l' ]
decimalNumeral    ::=  '0' | nonZeroDigit {digit}
hexNumeral        ::=  '0' ( 'x' | 'X' ) hexDigit {hexDigit}
digit             ::=  '0' | nonZeroDigit
nonZeroDigit      ::=  '1' | … | '9'

floatingPointLiteral
                  ::=  digit {digit} '.' digit {digit} [exponentPart] [floatType]
                |  '.' digit {digit} [exponentPart] [floatType]
                |  digit {digit} exponentPart [floatType]
                |  digit {digit} [exponentPart] floatType
exponentPart      ::=  ( 'E' | 'e' ) [ '+' | '-' ] digit {digit}
floatType         ::=  'F' | 'f' | 'D' | 'd'

booleanLiteral    ::=  'true' | 'false'

characterLiteral ::=  ''' (printableChar | charEscapeSeq) '''

stringLiteral     ::=  '"' {stringElement} '"'
                |  '"""' multiLineChars '"""'
stringElement     ::=  (printableChar except '"' )
                |  charEscapeSeq
multiLineChars    ::=  {[ '"' ] [ '"' ] charNoDoubleQuote} { '"' }
```

```
symbolLiteral     ::=   ‘’   plainid

comment           ::=   ‘/*’   “any sequence of characters; nested comments are allowed”   ‘*/’
                  |   ‘//’   “any sequence of characters up to end of line”

nl                ::=   “newlinecharacter”
semi              ::=   ‘;’   |   nl {nl}

  Literal           ::=   [ ‘–’ ] integerLiteral
                        |   [ ‘–’ ] floatingPointLiteral
                        |   booleanLiteral
                        |   characterLiteral
                        |   stringLiteral
                        |   symbolLiteral
                        |   ‘null’

  QualId            ::=   id { ‘.’   id}
  ids               ::=   id { ‘,’   id}

  Path              ::=   StableId
                        |   [id   ‘.’ ]   ‘this’
  StableId          ::=   id
                        |   Path   ‘.’   id
                        |   [id   ‘.’ ]   ‘super’   [ClassQualifier]   ‘.’   id
  ClassQualifier    ::=   ‘[’   id   ‘]’

  Type              ::=   FunctionArgTypes   ‘=>’   Type
                        |   InfixType [ExistentialClause]
  FunctionArgTypes  ::=   InfixType
                        |   ‘(’   [ ParamType { ‘,’   ParamType } ]   ‘)’
  ExistentialClause ::=   ‘forSome’   ‘{’   ExistentialDcl {semi ExistentialDcl}   ‘}’
  ExistentialDcl    ::=   ‘type’   TypeDcl
                        |   ‘val’   ValDcl
  InfixType         ::=   CompoundType {id [nl] CompoundType}
  CompoundType      ::=   AnnotType { ‘with’   AnnotType} [Refinement]
                        |   Refinement
  AnnotType         ::=   SimpleType {Annotation}
  SimpleType        ::=   SimpleType TypeArgs
                        |   SimpleType   ‘#’   id
                        |   StableId
                        |   Path   ‘.’   ‘type’
                        |   ‘(’   Types   ‘)’
  TypeArgs          ::=   ‘[’   Types   ‘]’
  Types             ::=   Type { ‘,’   Type}
  Refinement        ::=   [nl]   ‘{’   RefineStat {semi RefineStat}   ‘}’
  RefineStat        ::=   Dcl
                        |   ‘type’   TypeDef
                        |
```

```
TypePat          ::=  Type

Ascription       ::=   ':'   InfixType
                   |   ':'   Annotation {Annotation}
                   |   ':'    '_'    '*'

Expr             ::=  (Bindings | [ 'implicit' ] id |  '_' )  '=>'  Expr
                   |  Expr1
Expr1            ::=  `if' `(' Expr `)' {nl} Expr [[semi] `else' Expr]
                   |  `while' `(' Expr `)' {nl} Expr
                   |  `try' (`{' Block `}' | Expr) [`catch' `{' CaseClauses `}'] [`finally' Expr]
                   |  `do' Expr [semi] `while' `(' Expr ')'
                   |  `for' (`(' Enumerators `)' | `{' Enumerators `}') {nl} [`yield'] Expr
                   |  `throw' Expr
                   |  `return' [Expr]
                   |  [SimpleExpr `.'] id `=' Expr
                   |  SimpleExpr1 ArgumentExprs `=' Expr
                   |  PostfixExpr
                   |  PostfixExpr Ascription
                   |  PostfixExpr `match' `{' CaseClauses `}'
PostfixExpr      ::=  InfixExpr [id [nl]]
InfixExpr        ::=  PrefixExpr
                   |  InfixExpr id [nl] InfixExpr
PrefixExpr       ::=  [ '-'  |  '+'  |  '~'  |  '!' ] SimpleExpr
SimpleExpr       ::=   'new'  (ClassTemplate | TemplateBody)
                   |  BlockExpr
                   |  SimpleExpr1 [ '_' ]
SimpleExpr1      ::=  Literal
                   |  Path
                   |   '_'
                   |   '('  [Exprs]  ')'
                   |  SimpleExpr  '.'  id
                   |  SimpleExpr TypeArgs
                   |  SimpleExpr1 ArgumentExprs
                   |  XmlExpr
Exprs            ::=  Expr { ','  Expr}
ArgumentExprs    ::=   '('  [Exprs]  ')'
                   |   '('  [Exprs  ',' ] PostfixExpr  ':'   '_'   '*'   ')'
                   |  [nl] BlockExpr
BlockExpr        ::=   '{'  CaseClauses  '}'
                   |   '{'  Block  '}'
Block            ::=  BlockStat {semi BlockStat} [ResultExpr]
BlockStat        ::=  Import
                   |  {Annotation} [ 'implicit'  |  'lazy' ] Def
                   |  {Annotation} {LocalModifier} TmplDef
                   |  Expr1
                   |
ResultExpr       ::=  Expr1
                   |  (Bindings | ([ 'implicit' ] id |  '_' )  ':'  CompoundType)  '=>'  Block
```

```
Enumerators       ::=  Generator {semi Generator}
Generator         ::=  Pattern1  '<-'  Expr {[semi] Guard | semi Pattern1  '='  Expr}

CaseClauses       ::=  CaseClause { CaseClause }
CaseClause        ::=   'case'  Pattern [Guard]  '=>'  Block
Guard             ::=   'if'  PostfixExpr

Pattern           ::=  Pattern1 {  '|'  Pattern1 }
Pattern1          ::=  varid  ':'  TypePat
                     |   '_'    ':'  TypePat
                     |  Pattern2
Pattern2          ::=  varid [ '@'  Pattern3]
                     |  Pattern3
Pattern3          ::=  SimplePattern
                     |  SimplePattern { id [nl] SimplePattern }
SimplePattern     ::=   '_'
                     |  varid
                     |  Literal
                     |  StableId
                     |  StableId  '('  [Patterns  ')'
                     |  StableId  '('  [Patterns  ','  ] [varid  '@'  ]  '_'    '*'    ')'
                     |   '('  [Patterns]  ')'
                     |  XmlPattern
Patterns          ::=  Pattern [  ','  Patterns]
                     |   '_'  *

TypeParamClause   ::=   '['  VariantTypeParam {  ','  VariantTypeParam}  ']'
FunTypeParamClause::=   '['  TypeParam {  ','  TypeParam}  ']'
VariantTypeParam  ::=  {Annotation} [  '+'  |  '-'  ] TypeParam
TypeParam         ::=  (id |  '_'  ) [TypeParamClause] [  '>:'  Type] [  '<:'  Type]
                       {  '<%'  Type} {  ':'  Type}
ParamClauses      ::=  {ParamClause} [[nl]  '('  'implicit'  Params  ')'  ]
ParamClause       ::=  [nl]  '('  [Params]  ')'
Params            ::=  Param {  ','  Param}
Param             ::=  {Annotation} id [  ':'  ParamType] [  '='  Expr]
ParamType         ::=  Type
                     |   '=>'  Type
                     |  Type  '*'
ClassParamClauses ::=  {ClassParamClause}
                       [[nl]  '('  'implicit'  ClassParams  ')'  ]
ClassParamClause  ::=  [nl]  '('  [ClassParams]  ')'
ClassParams       ::=  ClassParam {  ','  ClassParam}
ClassParam        ::=  {Annotation} {Modifier} [(`val' | `var')]
                       id  ':'  ParamType [  '='  Expr]
Bindings          ::=   '('  Binding {  ','  Binding}  ')'
Binding           ::=  (id |  '_'  ) [  ':'  Type]

Modifier          ::=  LocalModifier
```

```
                    |  AccessModifier
                    |  'override'
LocalModifier    ::=  'abstract'
                    |  'final'
                    |  'sealed'
                    |  'implicit'
                    |  'lazy'
AccessModifier   ::=  ( 'private'  |  'protected' ) [AccessQualifier]
AccessQualifier  ::=  '['  (id |  'this' )  ']'

Annotation       ::=  '@'  SimpleType {ArgumentExprs}
ConstrAnnotation ::=  '@'  SimpleType ArgumentExprs

TemplateBody     ::=  [nl]  '{'  [SelfType] TemplateStat {semi TemplateStat}  '}'
TemplateStat     ::=  Import
                    |  {Annotation [nl]} {Modifier} Def
                    |  {Annotation [nl]} {Modifier} Dcl
                    |  Expr
                    |
SelfType         ::=  id [ ':'  Type]  '=>'
                    |  'this'  ':'  Type  '=>'

Import           ::=  'import'  ImportExpr { ','  ImportExpr}
ImportExpr       ::=  StableId  '.'  (id |  '_'  | ImportSelectors)
ImportSelectors  ::=  '{'  {ImportSelector  ',' } (ImportSelector |  '_' )  '}'
ImportSelector   ::=  id [ '=>'  id |  '=>'  '_' ]

Dcl              ::=  'val'  ValDcl
                    |  'var'  VarDcl
                    |  'def'  FunDcl
                    |  'type'  {nl} TypeDcl

ValDcl           ::=  ids  ':'  Type
VarDcl           ::=  ids  ':'  Type
FunDcl           ::=  FunSig [ ':'  Type]
FunSig           ::=  id [FunTypeParamClause] ParamClauses
TypeDcl          ::=  id [TypeParamClause] [ '>:'  Type] [ '<:'  Type]

PatVarDef        ::=  'val'  PatDef
                    |  'var'  VarDef
Def              ::=  PatVarDef
                    |  'def'  FunDef
                    |  'type'  {nl} TypeDef
                    |  TmplDef
PatDef           ::=  Pattern2 { ','  Pattern2} [ ':'  Type]  '='  Expr
VarDef           ::=  PatDef
                    |  ids  ':'  Type  '='  '_'
FunDef           ::=  FunSig [ ':'  Type]  '='  Expr
                    |  FunSig [nl]  '{'  Block  '}'
```

```
                    |   'this'   ParamClause ParamClauses
                        ( '='   ConstrExpr | [nl] ConstrBlock)
TypeDef           ::=  id [TypeParamClause]   '='   Type

TmplDef           ::=  [ 'case' ]   'class'   ClassDef
                    |  [ 'case' ]   'object'   ObjectDef
                    |   'trait'   TraitDef
ClassDef          ::=  id [TypeParamClause] {ConstrAnnotation} [AccessModifier]
                        ClassParamClauses ClassTemplateOpt
TraitDef          ::=  id [TypeParamClause] TraitTemplateOpt
ObjectDef         ::=  id ClassTemplateOpt
ClassTemplateOpt  ::=   'extends'   ClassTemplate | [[ 'extends' ] TemplateBody]
TraitTemplateOpt  ::=   'extends'   TraitTemplate | [[ 'extends' ] TemplateBody]
ClassTemplate     ::=  [EarlyDefs] ClassParents [TemplateBody]
TraitTemplate     ::=  [EarlyDefs] TraitParents [TemplateBody]
ClassParents      ::=  Constr { 'with'   AnnotType}
TraitParents      ::=  AnnotType { 'with'   AnnotType}
Constr            ::=  AnnotType {ArgumentExprs}
EarlyDefs         ::=  '{'   [EarlyDef {semi EarlyDef}]   '}'    'with'
EarlyDef          ::=  {Annotation [nl]} {Modifier} PatVarDef

ConstrExpr        ::=  SelfInvocation
                    |  ConstrBlock
ConstrBlock       ::=   '{'   SelfInvocation {semi BlockStat}   '}'
SelfInvocation    ::=   'this'   ArgumentExprs {ArgumentExprs}

TopStatSeq        ::=  TopStat {semi TopStat}
TopStat           ::=  {Annotation [nl]} {Modifier} TmplDef
                    |  Import
                    |  Packaging
                    |  PackageObject
                    |
Packaging         ::=   'package'   QualId [nl]   '{'   TopStatSeq   '}'
PackageObject     ::=   'package'    'object'   ObjectDef

CompilationUnit   ::=  { 'package'   QualId semi} TopStatSeq
```

특징

세미콜론을 사용하지 않아도 되지만 조건이 붙거나 한 문장에 여러 구문을 넣을 경우 세미콜론을 사용한다.
변수를 선언할 때 정적 변수를 선언할 때에는 val을 사용하고, 가변 변수는 var을 사용하여 변수를 선언한다.
익명 함수를 사용할 수 있다. 함수에 필요한 많은 줄과 여러 변수를 이용하지 않고 간단한 함수를 만들어 바로 사용할 수 있도록 하는 것이다.
한 줄에 여러 변수를 바인딩 할 수 있다.(리터럴 바인딩) 예 (var (x, y, z) = (1, 2, 3)
C와 같이 클래스가 존재하며 생성자 또한 존재하고 추상클래스와 클래스 상속 또한 존재한다.
익명 함수를 만드는 것과 같이 익명 클래스 또한 만들 수 있다.
자바의 interface와 비슷한 트레잇(traits)가 존재한다. interface와 다른 점은 상속할 때 섞어서 사용이 가능하다는 점이다.

4. Kotlin

코드

```
package kr.ac.kookmin.minjoo

fun main(args: Array<String>){
        var num = 5
        println("hello, world.")
}
```

Syntax

<Package definition and imports>

Package specification should be at the top of the source file:

```
package my.demo

import kotlin.text.*

// ...
```

It is not required to match directories and packages: source files can be placed arbitrarily in the file system.

<Program entry point>

An entry point of a Kotlin application is the main function.

```
fun main() {
    println("Hello world!")
}
```

<Functions>

Function having two Int parameters with Int return type:

```
fun sum(a: Int, b: Int): Int {
    return a + b
}
```

Function with an expression body and inferred return type:

```
fun sum(a: Int, b: Int) = a + b
```

Function returning no meaningful value:

```
fun printSum(a: Int, b: Int): Unit {
    println("sum of $a and $b is ${a + b}")
}
```

Unit return type can be omitted:

```
fun printSum(a: Int, b: Int) {
    println("sum of $a and $b is ${a + b}")
}
```

<Variables>
Read-only local variables are defined using the keyword val. They can be assigned a value only once.

```
val a: Int = 1  // immediate assignment
val b = 2   // `Int` type is inferred
val c: Int  // Type required when no initializer is provided
c = 3       // deferred assignment
```

Variables that can be reassigned use the var keyword:

```
var x = 5 // `Int` type is inferred
x += 1
```

Top-level variables:

```
val PI = 3.14
var x = 0

fun incrementX() {
    x += 1
}
```

<Comments>
Just like most modern languages, Kotlin supports single-line (or end-of-line) and multi-line (block) comments.

```
// This is an end-of-line comment

/* This is a block comment
   on multiple lines. */
```
Block comments in Kotlin can be nested.

```
/* The comment starts here
/* contains a nested comment */
and ends here. */
```

<String templates>

```
var a = 1
// simple name in template:
val s1 = "a is $a"

a = 2
// arbitrary expression in template:
val s2 = "${s1.replace("is", "was")}, but now is $a"
```

<Conditional expressions>

```
fun maxOf(a: Int, b: Int): Int {
    if (a > b) {
        return a
    } else {
        return b
    }
}
```

In Kotlin, if can also be used as an expression:

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b
```

<Nullable values and null checks>

A reference must be explicitly marked as nullable when null value is possible.

Return null if str does not hold an integer:

```
fun parseInt(str: String): Int? {
    // ...
}
```

Use a function returning nullable value:

```
fun printProduct(arg1: String, arg2: String) {
    val x = parseInt(arg1)
    val y = parseInt(arg2)

    // Using `x * y` yields error because they may hold nulls.
    if (x != null && y != null) {
        // x and y are automatically cast to non-nullable after null check
        println(x * y)
    }
    else {
        println("'$arg1' or '$arg2' is not a number")
    }
}
```

Target platform: JVMRunning on kotlin v. 1.4.10

or

```
// ...
if (x == null) {
    println("Wrong number format in arg1: '$arg1'")
    return
}
if (y == null) {
    println("Wrong number format in arg2: '$arg2'")
    return
}
```

```
// x and y are automatically cast to non-nullable after null check
println(x * y)
```

<Type checks and automatic casts>
The is operator checks if an expression is an instance of a type. If an immutable local variable or property is checked for a specific type, there's no need to cast it explicitly:

```
fun getStringLength(obj: Any): Int? {
    if (obj is String) {
        // `obj` is automatically cast to `String` in this branch
        return obj.length
    }

    // `obj` is still of type `Any` outside of the type-checked branch
    return null
}
```

or

```
fun getStringLength(obj: Any): Int? {
    if (obj !is String) return null

    // `obj` is automatically cast to `String` in this branch
    return obj.length
}
```

or even

```
fun getStringLength(obj: Any): Int? {
    // `obj` is automatically cast to `String` on the right-hand side of `&&`
    if (obj is String && obj.length > 0) {
        return obj.length
    }

    return null
}
```

<for loop>

```
val items = listOf("apple", "banana", "kiwifruit")
for (item in items) {
    println(item)
}
```

or

```
val items = listOf("apple", "banana", "kiwifruit")
for (index in items.indices) {
    println("item at $index is ${items[index]}")
```

```
}
Target platform: JVMRunning on kotlin v. 1.4.10

<while loop>
val items = listOf("apple", "banana", "kiwifruit")
var index = 0
while (index < items.size) {
    println("item at $index is ${items[index]}")
    index++
}

<when expression>
fun describe(obj: Any): String =
    when (obj) {
        1          -> "One"
        "Hello"    -> "Greeting"
        is Long    -> "Long"
        !is String -> "Not a string"
        else       -> "Unknown"
    }

<Ranges>
Check if a number is within a range using in operator:

val x = 10
val y = 9
if (x in 1..y+1) {
    println("fits in range")
}

Check if a number is out of range:

val list = listOf("a", "b", "c")

if (-1 !in 0..list.lastIndex) {
    println("-1 is out of range")
}
if (list.size !in list.indices) {
    println("list size is out of valid list indices range, too")
}

Iterating over a range:

for (x in 1..5) {
    print(x)
}

or over a progression:
```

```
for (x in 1..10 step 2) {
    print(x)
}
println()
for (x in 9 downTo 0 step 3) {
    print(x)
}
```

<Collections>
Iterating over a collection:

```
for (item in items) {
    println(item)
}
```

Target platform: JVMRunning on kotlin v. 1.4.10
Checking if a collection contains an object using in operator:

```
when {
    "orange" in items -> println("juicy")
    "apple" in items -> println("apple is fine too")
}
```

Using lambda expressions to filter and map collections:

```
val fruits = listOf("banana", "avocado", "apple", "kiwifruit")
fruits
  .filter { it.startsWith("a") }
  .sortedBy { it }
  .map { it.toUpperCase() }
  .forEach { println(it) }
```

Creating basic classes and their instances

```
val rectangle = Rectangle(5.0, 2.0)
val triangle = Triangle(3.0, 4.0, 5.0)
```

특징
세미콜론(;)을 사용하지 않는다.
Scala와 같이 정적 변수의 경우 선언할 때 val을 사용하고 일반적인 변수에는 var을 사용한다.
함수선언에서 한줄 선언이 가능하다 예 fun sum(a: Int, b: Int): Int = a + b
null 변수를 선언할 때 변수형 뒤에 ?을 붙여준다. 예 var num: Int? = null
Any라는 키워드는 java의 Object에 해당되고 is는 instanceof와 기능이 같다.
C++과 java의 swith문이 when이라는 키워드로 존재한다. break는 존재하지 않고 case 대신 중괄호 {}를 사용한다.
foreach문에서 자기 자신에게 접근하려면 it을 사용한다.
java와 같이 class가 존재하고 data class라는 것이 존재한다.
class와 비슷한 object가 존재한다. 생성자가 없고 객체화 없이 다른 클래스와 함수에서 접근 가능하다.
함수는 fun으로 선언해준다.

5. Ruby

코드

```
num = 5
puts  “hello, world.”
```

Syntax

```
PROGRAM : COMPSTMT
T : ";" | "\n" //a newline can terminate a statement
COMPSTMT : STMT {T EXPR} [T]
STMT : CALL do ["|" [BLOCK_VAR] "|"] COMPSTMT end
| undef FNAME
| alias FNAME FNAME
| STMT if EXPR
| STMT while EXPR
| STMT unless EXPR
| STMT until EXPR
| "BEGIN" "{" COMPSTMT "}" //object initializer
| "END" "{" COMPSTMT "}" //object finalizer
| LHS = COMMAND [do ["|" [BLOCK_VAR] "|"] COMPSTMT end]
| EXPR
EXPR : MLHS = MRHS
| return CALL_ARGS
| yield CALL_ARGS
| EXPR and EXPR
| EXPR or EXPR
| not EXPR
| COMMAND
| ! COMMAND
| ARG
CALL : FUNCTION
| COMMAND
COMMAND : OPERATION CALL_ARGS
| PRIMARY.OPERATION CALL_ARGS
| PRIMARY :: OPERATION CALL_ARGS
| super CALL_ARGS
FUNCTION : OPERATION ["(" [CALL_ARGS] ")"]
| PRIMARY.OPERATION "(" [CALL_ARGS] ")"
| PRIMARY :: OPERATION "(" [CALL_ARGS] ")"
| PRIMARY.OPERATION
| PRIMARY :: OPERATION
| super "(" [CALL_ARGS] ")"
| super
ARG : LHS = ARG
| LHS OP_ASGN ARG
| ARG .. ARG | ARG ... ARG
```

```
| ARG + ARG | ARG - ARG | ARG * ARG | ARG / ARG
| ARG % ARG | ARG ** ARG
| + ARG | - ARG
| ARG "|" ARG
| ARG ^ ARG | ARG & ARG
| ARG <=> ARG
| ARG > ARG | ARG >= ARG | ARG < ARG | ARG <= ARG
| ARG == ARG | ARG === ARG | ARG != ARG
| ARG =~ ARG | ARG !~ ARG
| ! ARG | ~ ARG
| ARG << ARG | ARG >> ARG
| ARG && ARG | ARG || ARG
| defined? ARG
| PRIMARY
PRIMARY: "(" COMPSTMT ")"
| LITERAL
| VARIABLE
| PRIMARY :: IDENTIFIER
| :: IDENTIFIER
| PRIMARY "[" [ARGS] "]"
| "[" [ARGS [,]] "]"
| "{" [ARGS | ASSOCS [,]] "}"
| return ["(" [CALL_ARGS] ")"]
| yield ["(" [CALL_ARGS] ")"]
| defined? "(" ARG ")"
| FUNCTION
| FUNCTION "{" ["|" [BLOCK_VAR] "|"] COMPSTMT "}"
| if EXPR THEN
COMPSTMT
{elsif EXPR THEN
COMPSTMT}
[else
COMPSTMT]
end
| unless EXPR THEN
COMPSTMT
[else
COMPSTMT]
end
| while EXPR DO COMPSTMT end
| until EXPR DO COMPSTMT end
| case COMPSTMT
when WHEN_ARGS THEN COMPSTMT
{when WHEN_ARGS THEN COMPSTMT}
[else
COMPSTMT]
end
| for BLOCK_VAR in EXPR DO
COMPSTMT
```

end
| begin
COMPSTMT
{rescue [ARGS] DO
COMPSTMT}
[else
COMPSTMT]
[ensure
COMPSTMT]
end
| class IDENTIFIER [< IDENTIFIER]
COMPSTMT
end
| module IDENTIFIER
COMPSTMT
end
| def FNAME ARGDECL
COMPSTMT
end
| def SINGLETON (. | ::) FNAME ARGDECL
COMPSTMT
end
WHEN_ARGS : ARGS [, * ARG] | * ARG
THEN : T | then | T then //"then" and "do" can go on next line
DO : T | do | T do
BLOCK_VAR : LHS | MLHS
MLHS : MLHS_ITEM , [MLHS_ITEM (, MLHS_ITEM)*] [* [LHS]]
| * LHS
MLHS_ITEM : LHS | "(" MLHS ")"
LHS : VARIABLE
| PRIMARY "[" [ARGS] "]"
| PRIMARY.IDENTIFIER
MRHS : ARGS [, * ARG] | * ARG
CALL_ARGS : ARGS
| ARGS [, ASSOCS] [, * ARG] [, & ARG]
| ASSOCS [, * ARG] [, & ARG]
| * ARG [, & ARG] | & ARG
| COMMAND
ARGS : ARG (, ARG)*
ARGDECL : "(" ARGLIST ")"
| ARGLIST T
ARGLIST : IDENTIFIER(,IDENTIFIER)*[, *[IDENTIFIER]][,&IDENTIFIER]
| *IDENTIFIER[, &IDENTIFIER]
| [&IDENTIFIER]
SINGLETON : VARIABLE
| "(" EXPR ")"
ASSOCS : ASSOC {, ASSOC}
ASSOC : ARG => ARG
VARIABLE : VARNAME | nil | self

LITERAL : numeric | SYMBOL | STRING | STRING2 | HERE_DOC | REGEXP
The following are recognized by the lexical analyzer.
OP_ASGN : += | -= | *= | /= | %= | **=
| &= | |= | ^= | <<= | >>=
| &&= | ||=
SYMBOL : :FNAME | :VARNAME
FNAME : IDENTIFIER | .. | "|" | ^ | & | <=> | == | === | =~
| > | >= | < | <= | + | - | * | / | % | **
| << | >> | ~ | +@ | -@ | [] | []=
OPERATION : IDENTIFIER [! | ?]
VARNAME : GLOBAL | @IDENTIFIER | IDENTIFIER
GLOBAL : $IDENTIFIER | $any_char | $-any_char
STRING : " {any_char} "
| ' {any_char} '
| ` {any_char} `
STRING2 : %[Q|q|x]char {any_char} char
HERE_DOC : <<(IDENTIFIER | STRING)
{any_char}
IDENTIFIER
REGEXP : / {any_char} / [i|o|p]
| %r char {any_char} char
IDENTIFIER : sequence in /[a-zA-Z_]{a-zA-Z0-9_}/.

특징
컴파일러가 필요없고 java와 같은 객체지향 언어이다. 하지만 모든 것이 객체라는 특징을 가지고 있다.
변수 선언 시 변수명을 무조건 쓸 필요 없다.
세미콜론(;)을 사용하지 않아도 된다.
java와 같이 class가 존재하며 class의 이름은 무조건 대문자 영어로 시작해야 한다.
함수는 def 함수명 end 형식이며, 함수명은 소문자 영어와 _를 사용한다.
return값이 true false일 경우 함수명 뒤에 ?을 붙인다.
함수에서 return을 사용하지 않은 경우 마지막 수행된 구문을 반환한다.
scala와 같이 익명 함수가 존재한다.