요구사항

```swift
private func appendTokenToJSONObject(_ token:Token, _ key:String, value:Any)
    throws -> Token {
    switch token {
    case .jsonObject(var objects):
        if value is Double {
            objects[key] = .number(value: value as! Double)
        } else if value is String {
            objects[key] = .string(value: value as! String)
        } else if value is Bool {
            objects[key] = .bool(value: value as! Bool)
        } else {
            throw JSONLexer.Error.invalidFormatCurlyBrace
        }
        return Token.jsonObject(objects)
    default: throw JSONLexer.Error.invalidFormatCurlyBrace
    }
}

private mutating func bracket() throws -> Token {
    var token:Token = Token.jsonArray(tokens: [])
    // bracket이면 다음 거
    advance()

    while let nextCharacter = peak() {
        switch nextCharacter {
        case numbers :
            let value = try getNumber()
            token = try appendTokenToJSONArray(token, value)

        // 공백과 쉼표는 다음으로
        case blank, comma, closeCurlyBrace: advance()
        case quotationMark:
            advance()
            let value = try doubleQuote()
            token = try appendTokenToJSONArray(token, value)

        case firstCharacterOfBool:
            let value = try getBool()
            token = try appendTokenToJSONArray(token, value)
        case openCurlyBrace :
            let objectToken = try curlyBrace()
            token = try appendTokenToJSONArray(token, objectToken)
        case closeBracket where position == input.index(before: input.endIndex) :
            return token
        default: throw JSONLexer.Error.invalidFormatBracket
        }
    }
    throw JSONLexer.Error.invalidFormatBracket
}

private func appendTokenToJSONArray(_ token:Token, _ value:Any) throws -> Token {
    switch token {
    case .jsonArray(var tokens):
        if value is Double {
            tokens.append(.number(value:value as! Double))
        } else if value is String {
            tokens.append(.string(value:value as! String))
        } else if value is Bool {
            tokens.append(.bool(value: value as! Bool))
        } else if value is Token {

            var jsonObjects:[String:Token] = [:]
            jsonObjects =  try getJsonObjects(value)
            tokens.append(.jsonObject(jsonObjects))
```

value의 타입을 좀 더 구체적으로 만들고 다형성을 이용해서 if else를 제거하라!

접근방법

1. 4월2일 피드백 강의 때 테스트 이론에서의 힌트
  “ 구현 내용은 같고 타입만 다를면 제네릭을 활용하세요. ”

-> 종합해보면 제네릭으로 다형성만들고 if else를 제거 해야 한다...?

다양한 타입이 같은 방식으로 동작하는 코드는 어떻게 작성할 수 있을까? 제네릭을 적용하면 컴파일러가 알지 못하는 타입을 사용하여 타입이나 함수를 작성할 수 있다. from BNR

다형성 : 형(type)을 여러개 받을 수 있게
제네릭은 형(type)을 그 때 그 때 다르게 할 수 있다.

계속 고민 했는데 제네릭은 아닌 것 같습니다만...

다른 접근방법 protocol

Protocol 사용하여 차근차근

1. value:Any
   value에는 사실 String, Bool, Double만 들어간다.
   Any가 아닌 좀 더 구체적인 타입을 만들어보자!

2. 이 3개를 다 담을 수 있는 그릇을 만들자! (다형성)
   빈 protocol TokenBasicValueable {}를 생성해서
   Double. String, Bool이 위 프로토콜을 준수하도록 만든다.

```
32
33  protocol TokenBasicValueable {}
34
35  extension Double : TokenBasicValueable {}
36
37  extension String : TokenBasicValueable {}
38
39  extension Bool : TokenBasicValueable {}
40
```

다른 접근방법

3. 적용

```
private func appendTokenToJSONObject(_ token:Token, _ key:String, _ value:Any) throws -> Token {
    switch token {
    case .jsonObject(var objects):
        if value is Double {
            objects[key] = .number(value: value as! Double)
        } else if value is String {
            objects[key] = .string(value: value as! String)
        } else if value is Bool {
            objects[key] = .bool(value: value as! Bool)
        } else {
            throw JSONLexer.Error.invalidFormatCurlyBrace
        }
        return Token.jsonObject(objects)
    default: throw JSONLexer.Error.invalidFormatCurlyBrace
    }
```

```
private func appendTokenToJSONObject(_ token:Token, _ key:String, _ value:TokenBasicValueable) throws -> Token {
    switch token {
    case .jsonObject(var objects):
        if value is Double {
            objects[key] = .number(value: value)
        } else if value is String {
            objects[key] = .string(value: value)
        } else if value is Bool {
            objects[key] = .bool(value: value)
        } else {
            throw JSONLexer.Error.invalidFormatCurlyBrace
        }
        return Token.jsonObject(objects)
    default: throw JSONLexer.Error.invalidFormatCurlyBrace
    }
}
```

다른 접근방법

4. 문제 if else는 여전히 남아 있다.

```
private func appendTokenToJSONObject(_ token:Token, _ key:String, _ value:TokenBasicValueable) throws -> Token {
    switch token {
    case .jsonObject(var objects):
        if value is Double {
            objects[key] = .number(value: value)
        } else if value is String {
            objects[key] = .string(value: value)
        } else if value is Bool {
            objects[key] = .bool(value: value)
        } else {
            throw JSONLexer.Error.invalidFormatCurlyBrace
        }
        return Token.jsonObject(objects)
    default: throw JSONLexer.Error.invalidFormatCurlyBrace
    }
}
```

```
enum Token{
    case string(value:String)
    case number(value:Double)
    case bool(value:Bool)
    case jsonArray(tokens:[Token])
    case jsonObject(JSONObject)
}
```

연관 값은 protocol을 사용해서 하나로 만들었지만
토큰은 합치지 않았음. string, number, bool이 그대로 존재
따라서 여전히 if else로 분기를 해줘야 함

다른 접근방법

5. 하나로 합치기 (소스가 깔끔)

```
private func appendTokenToJSONObject(_ token:Token, _ key:String, _ value:TokenBasicValueable) throws -> Token {
    switch token {
    case .jsonObject(var objects):
        if value is Double {
            objects[key] = .number(value: value)
        } else if value is String {
            objects[key] = .string(value: value)
        } else if value is Bool {
            objects[key] = .bool(value: value)
        } else {
            throw JSONLexer.Error.invalidFormatCurlyBrace
        }
        return Token.jsonObject(objects)
    default: throw JSONLexer.Error.invalidFormatCurlyBrace
    }
}
```

```
enum Token{
    case string(value:String)
    case number(value:Double)
    case bool(value:Bool)
    case jsonArray(tokens:[Token])
    case jsonObject(JSONObject)
}
```

```
private func appendTokenToJSONObject(_ token:Token, _ key:String, _ value:TokenBasicValueable) throws -> Token {
    switch token {
    case .jsonObject(var objects):
        objects[key] = .basicValue(value:value)
        return Token.jsonObject(objects)
    default: throw JSONLexer.Error.invalidFormatCurlyBrace
    }
}
```

```
enum Token:TokenBasicValueable{
    case basicValue(value:TokenBasicValueable)
    case jsonArray(tokens:[Token])
    case jsonObject(JSONObject)
}
```

지금까지 잘못된 방법이었습니다.

죄송합니다.

잘못된 방법에 대한 피드백

```
32
33 protocol TokenBasicValueable {}
34
35 extension Double : TokenBasicValueable {}
36
37 extension String : TokenBasicValueable {}
38
39 extension Bool : TokenBasicValueable {}
40
```

이렇게 아무런 메소드가 없이 프로토콜만 채택하는 건 의미가 없습니다. 다형성으로 해결하라는 것은 여러 객체에 동일한 시그니처를 갖는 메소드가 있어서 호출할 때는 타입과 상관없이 호출해도, (다양한 형태로) 각자 타입의 메소드가 호출되는 방식을 말합니다.
from jk

다형성은 형은 여러개 받는 것 뿐만이 아니다!!!

피드백 반영

```
32
33 protocol TokenBasicValueable {}
34
35 extension Double : TokenBasicValueable {}
36
37 extension String : TokenBasicValueable {}
38
39 extension Bool : TokenBasicValueable {}
40
```

시그니처 메소드 만들기

```
32
33 protocol TokenBasicValueable {
34     func getToken() -> Token
35 }
36
37 extension Double : TokenBasicValueable {
38     func getToken() -> Token {
39         return .number(value: self)
40     }
41 }
42
43 extension String : TokenBasicValueable {
44     func getToken() -> Token {
45         return .string(value:self)
46     }
47 }
48
49 extension Bool : TokenBasicValueable {
50     func getToken() -> Token {
51         return .bool(value:self)
52     }
53 }
54
```

피드백 반영

```
private func appendTokenToJSONObject(_ token:Token, _ key:String, _ value:TokenBasicValueable) throws -> Token {
    switch token {
    case .jsonObject(var objects):
        objects[key] = .basicValue(value:value)
        return Token.jsonObject(objects)
    default: throw JSONLexer.Error.invalidFormatCurlyBrace
    }
}
```

시그니처 메소드 사용

```
41
42    private func appendTokenToJSONObject(_ token:Token, _ key:String, _ value:TokenBasicValueable) throws -> Token {
43        switch token {
44        case .jsonObject(var objects):
45            objects[key] = value.getToken()
46            return .jsonObject(objects)
47        default: throw JSONLexer.Error.invalidFormatCurlyBrace
48        }
49    }
50
```

```
enum Token:TokenBasicValueable{
    case basicValue(value:TokenBasicValueable)
    case jsonArray(tokens:[Token])
    case jsonObject(JSONObject)
}
```

```
enum Token{
    case string(value:String)
    case number(value:Double)
    case bool(value:Bool)
    case jsonArray(tokens:[Token])
    case jsonObject(JSONObject)
}
```

최종 결과

```
private func appendTokenToJSONObject(_ token:Token, _ key:String, _ value:Any) throws -> Token {
    switch token {
    case .jsonObject(var objects):
        if value is Double {
            objects[key] = .number(value: value as! Double)
        } else if value is String {
            objects[key] = .string(value: value as! String)
        } else if value is Bool {
            objects[key] = .bool(value: value as! Bool)
        } else {
            throw JSONLexer.Error.invalidFormatCurlyBrace
        }
        return Token.jsonObject(objects)
    default: throw JSONLexer.Error.invalidFormatCurlyBrace
    }
}
```

```
41
42  private func appendTokenToJSONObject(_ token:Token, _ key:String, _ value:TokenBasicValueable) throws -> Token {
43      switch token {
44      case .jsonObject(var objects):
45          objects[key] = value.getToken()
46          return .jsonObject(objects)
47      default: throw JSONLexer.Error.invalidFormatCurlyBrace
48      }
49  }
50
```

다형성 효과

장점 :

if-else 혹은 swift-case로 타입별로 나눠진 코드가 객체 내부로 분산되고, 호출하는 상위 모듈에서 코드가 비교문 없이 간결해진다. 그러면 상위 모듈이나 하위에 각 타입별 객체가 자신의 역할에 충실하게 된다. from JK

단점:

많은 시간을 들여 고민을 해야 합니다...(저만 그럴 수도 IO 89라서...)

이해를 돕기 위해
간단한 버전의 다형성 만들기 코딩