프로그래밍 언어 Rust의 부상과 설계 철학

[1~4] 다음 글을 읽고 물음에 답하시오.

현대 사회에서 소프트웨어의 규모와 복잡성이 증가함에 따라, 메모리 관리 오류로 인한 보안 취약점과 시스템 장애가 심각한 문제로 대두되었다. 전통적으로 시스템 프로그래밍 영역에서는 C나 C++와 같이 개발자가 직접 메모리를 할당하고 해제하는 수동 메모리 관리 방식이 주를 이루었다. 이는 뛰어난 실행 성능을 보장하지만, 널 포인터 역참조나 매달린 포인터와 같은 치명적인 버그를 발생시킬 위험을 내포한다. 반면, Java나 Go 같은 언어는 가비지 컬렉터(GC)를 도입하여 메모리 안전성을 확보했으나, 예측 불가능한 런타임 일시 정지로 인해 실시간 처리가 중요한 시스템에는 부적합했다. 이러한 배경 속에서 Rust는 ㉠가비지 컬렉터(GC) 없이도 메모리 안전성과 고성능을 동시에 달성하고자 하는 야심 찬 시도로 등장하였다.

Rust의 핵심 설계 철학은 ㉡제로 코스트 추상화와 컴파일 시점 안전성 보장이다. 제로 코스트 추상화란 사용하지 않는 기능에 대가를 지불하지 않으며, 사용하는 기능은 손으로 작성한 코드와 동일한 성능을 내는 원칙이다. 이를 통해 개발자는 고수준 언어의 편리함을 누리면서도 저수준 언어의 성능을 그대로 활용할 수 있다. Rust가 GC 없이 메모리 안전성을 보장하는 비결은 소유권(Ownership) 시스템에 있다. 소유권의 규칙에 따라 모든 값은 정확히 하나의 소유자를 가지며, 소유자가 스코프를 벗어나면 값은 자동으로 해제된다. 소유권이 다른 변수로 이동하면 이전 소유자는 해당 값을 사용할 수 없어, 이중 해제나 잘못된 참조를 컴파일러가 원천적으로 차단한다.

소유권을 매번 이전하는 것은 비효율적이므로, Rust는 빌림(Borrowing)이라는 참조 메커니즘을 도입한다. 빌림 검사기는 ㉢공유 xor 가변이라는 규칙을 강제한다. 즉, 불변 참조는 동시에 여러 개 존재할 수 있지만, 가변 참조는 한 시점에 단 하나만 존재해야 하며, 가변 참조와 불변 참조는 동시에 공존할 수 없다. 어떤 데이터를 변경하는 주체가 있을 때 다른 주체가 이를 읽거나 쓰려 하면 데이터 경쟁(data race)이 발생하는데, Rust는 이를 타입 시스템 수준에서 정적 분석하여 컴파일을 거부한다. 이는 다른 언어들이 런타임 락이나 테스팅에 의존하던 동시성 버그를 사전에 제거하여 '두려움 없는 동시성'을 가능하게 한다. 나아가 참조가 가리키는 대상보다 오래 살아남는 매달린 포인터를 방지하기 위해, 모든 참조에 수명(Lifetime)이라는 정보를 부여하고 컴파일러가 이를 추적하도록 한다.

메모리 안전성에 더해, Rust는 타입 시스템을 통해 논리적 오류까지 구조적으로 방지한다. null 참조를 언어 차원에서 배제하여 '10억 달러짜리 실수'로 불리는 null 포인터 문제를 해결했으며, 값의 부재는 Option〈T〉로, 실패 가능성은 Result〈T, E〉로 타입 수준에서 명시한다. 패턴 매칭을 통해 모든 경우의 수를 빠짐없이 처리하도록 강제함으로써, 새로운 조건이 추가되었을 때 발생할 수 있는 처리 누락을 컴파일 시점에 잡아낸다. 또한 상속을 지원하지 않는 대신 트레이트(Trait)라는 인터페이스 메커니즘으로 다형성을 제공하며, 정적 디스패치를 통한 단조화(monomorphization)와 동적 디스패치(dyn Trait) 중 상황에 맞는 방식을 선택할 수 있어 성능과 유연성 사이의 균형을 잡는다.

이러한 안전성과 성능의 이점은 Rust의 활용 영역을 시스템 프로그래밍을 넘어 다방면으로 확장시켰다. 2025년에는 Linux 커널에 Rust로 작성된 드라이버가 메인 트리에 병합되었고, 미국 정부 역시 메모리 안전한 언어로의 전환을 공식 권장하는 등 산업계 전반에서의 채택이 가속화되고 있다. 또한 WebAssembly(Wasm)와의 뛰어난 연동성을 바탕으로 웹 브라우저는 물론 서버리스 및 엣지 컴퓨팅 환경에서 고성능 코드를 실행하는 데 활용된다. 비동기 프로그래밍을 지원하는 강력한 웹 프레임워크와 Cargo 패키지 관리자가 제공하는 생산성 높은 생태계는 개발자들이 복잡한 백엔드 시스템을 안정적으로 구축할 수 있게 한다.

물론 Rust가 완벽한 것은 아니다. 소유권, 빌림, 수명, 트레이트 등의 개념을 동시에 익혀야 하는 가파른 학습 곡선은 초기 개발자의 생산성을 가로막는 주요 장벽으로 지적된다. 컴파일러와 '싸우며' 코드를 작성해야 하는 경험은 빠른 프로토타이핑이 필요한 상황에서 부담으로 작용할 수 있다. 그럼에도 불구하고, 언어 설계는 항상 타협의 예술이며 Rust가 선택한 ㉣컴파일 타임 복잡성을 감수하고 런타임 안전성을 얻는다는 타협은 고가용성과 보안이 필수적인 현대 IT 인프라에서 분명히 옳은 선택이었다. 앞으로 시스템 소프트웨어의 기반은 점차 Rust로 대체될 것이며, 이 과정에서 우리는 더 안전하고 신뢰할 수 있는 소프트웨어 환경을 얻게 될 것이다.

1. 윗글의 내용과 일치하지 않는 것은?

① C/C++는 수동 메모리 관리를 통해 뛰어난 실행 성능을 보장하지만, 널 포인터 역참조와 같은 치명적 버그의 발생 위험을 내포하고 있다.

② Rust의 소유권 시스템은 값의 이중 해제나 잘못된 참조를 컴파일러가 원천적으로 차단하는데, 이는 소유권이 이전된 이전 소유자가 해당 값을 사용할 수 없게 만드는 규칙에 기반한다.

③ 가비지 컬렉터를 도입한 언어들은 런타임 일시 정지로 인해 실시간 처리가 중요한 시스템에 부적합할 수 있으므로, Rust는 가비지 컬렉터 없이도 컴파일 시점 정적 분석을 통해 데이터 경쟁을 사전에 제거한다.

④ Rust는 null 참조를 배제하고 Option〈T〉와 Result〈T, E〉를 통해 값의 부재와 실패 가능성을 타입 수준에서 명시하며, 패턴 매칭을 통해 모든 경우의 수를 빠짐없이 처리하도록 강제한다.

⑤ Rust는 상속 대신 트레이트를 통한 다형성을 제공하는데, 이는 정적 디스패치와 동적 디스패치를 상황에 맞게 선택할 수 있게 하여 성능 저하 없이 유연성을 확보하는 메커니즘이다.

2. 윗글의 내용과 일치하는 것은?

① Rust의 빌림 검사기는 가변 참조와 불변 참조가 동시에 존재하는 것을 허용하지 않으므로, 다수의 주체가 동일한 데이터를 동시에 읽으려 할 때 반드시 단일 가변 참조로 직렬화하여 데이터 경쟁을 방지한다.

② Rust는 상속을 지원하지 않는 대신 트레이트를 통해 다형성을 제공하며, 정적 디스패치 시 컴파일 타임에 모든 경우의 수를 빠짐없이 처리하도록 강제하여 처리 누락을 방지한다.

③ Rust에서 값의 부재와 실패 가능성이 각각 Option〈T〉와 Result〈T, E〉로 타입 수준에서 명시되므로, null 참조를 언어 차원에서 배제한 것은 패턴 매칭의 전제 조건이 된다.

④ Rust의 소유권 시스템은 소유권이 다른 변수로 이동하면 이전 소유자가 해당 값을 사용할 수 없게 하여 이중 해제를 차단하므로, 매달린 포인터를 수명 정보로 방지하는 것과 동일한 컴파일 시점 메커니즘이다.

⑤ Rust는 가비지 컬렉터를 도입하지 않으면서 소유권 시스템을 통해 메모리를 자동으로 해제하므로, 예측 불가능한 런타임 일시 정지 문제를 해결하면서도 개발자가 직접 메모리를 할당하고 해제하는 수동 관리의 성능을 보장한다.

3. 윗글의 논리에 비추어 추론한 것으로 가장 적절한 것은?

① Rust의 빌림 검사기가 '공유 xor 가변' 규칙을 강제하므로, 단일 스레드 환경에서는 데이터 경쟁이 발생하지 않는다는 점에서 다중 스레드 환경과 동일하게 '두려움 없는 동시성'이 보장된다.

② Rust가 상속을 배제하고 트레이트로 다형성을 제공하는 것은, 상속이 초래할 수 있는 런타임 비용을 제거함으로써 제로 코스트 추상화 원칙을 관철하려는 선택이다.

③ 가비지 컬렉터 기반 언어가 예측 불가능한 런타임 일시 정지로 실시간 처리에 부적합했다는 윗글의 진술은, 가비지 컬렉터가 메모리 안전성 자체를 보장하지 못한다는 전제에 근거한다.

④ Rust의 소유권 시스템이 이중 해제를 컴파일 시점에 차단한다는 점은, 소유권 이동 후 이전 소유자의 접근이 런타임 패닉을 유발하는 방식으로 안전성을 확보한다는 것을 함의한다.

⑤ Rust가 참조에 수명 정보를 부여하고 컴파일러가 이를 추적하도록 한다는 점은, 참조의 유효성 검사를 런타임으로 미루는 방식이 매달린 포인터를 근본적으로 해결할 수 없다는 판단에 기반한다.

4. 밑줄 친 ㉣의 문맥적 의미로 가장 적절한 것은?

① 가비지 컬렉터 도입으로 인한 런타임 일시 정지를 감수하고, 대신 개발자의 직접 메모리 관리 부담을 제거하여 안전성을 확보한다는 의미

② 소유권·빌림·수명 규칙을 학습하고 컴파일러의 엄격한 검사에 맞추어 코드를 작성하는 부담을 안는 대신, 실행 시점의 메모리 안전성과 데이터 경쟁 방지를 보장받는다는 의미

③ 컴파일 시점에 수행하는 정적 분석의 범위를 축소하여 빌드 시간을 단축하는 대신, 런타임 락과 테스팅에 의존하여 동시성 버그를 사후에 검증한다는 의미

④ 고수준 언어의 편리한 추상화를 포기하고 손으로 작성한 저수준 코드에 의존함으로써, 런타임 오버헤드 없이 실행 성능을 극대화한다는 의미

⑤ 트레이트의 정적 디스패치만을 허용하여 단조화로 인한 코드 크기 증가를 감수하는 대신, 동적 디스패치의 유연성을 희생하여 런타임 안전성을 확보한다는 의미

지문 분석 및 해설

프로그래밍 언어 Rust의 부상과 설계 철학

▣ 지문 분석

Rust 언어의 메모리 안전성 및 설계 철학 분석

■ 개요

이 지문은 현대 소프트웨어의 복잡성 증가로 인한 메모리 관리 문제를 배경으로, Rust 프로그래밍 언어가 어떻게 가비지 컬렉터(GC) 없이도 메모리 안전성과 고성능을 동시에 달성하는지 그 원리와 장단점, 그리고 산업적 활용 현황을 설명하고 있다.

■ 문단별 분석

1문단 〔문제 제기 및 배경 설명〕

C/C++의 수동 메모리 관리(성능 좋지만 버그 위험)와 Java/Go의 GC 방식(안전하지만 런타임 일시 정지 문제)을 비교하며, Rust가 GC 없이 안전성과 성능을 동시에 추구하며 등장한 배경을 설명한다.

2문단 〔핵심 원리 제시 1 (소유권 시스템)〕

Rust의 핵심 설계 철학인 '제로 코스트 추상화'를 소개하고, '소유권(Ownership)' 시스템을 통해 값의 이중 해제나 잘못된 참조를 컴파일 시점에 차단하는 원리를 설명한다.

3문단 〔핵심 원리 제시 2 (빌림과 수명 시스템)〕

소유권 이전의 비효율을 해결하기 위한 '빌림(Borrowing)' 메커니즘을 설명하며, '공유 xor 가변' 규칙으로 데이터 경쟁을 막고 '수명(Lifetime)'으로 매달린 포인터를 방지하여 동시성 버그를 사전에 제거함을 강조한다.

4문단 〔안전성 보장의 확장 (논리적 오류 방지 및 타입 시스템)〕

메모리 안전성을 넘어 논리적 오류까지 방지하는 방법을 설명한다. null 참조 배제, Option<T>와 Result<T, E>를 통한 타입 수준 명시, 패턴 매칭 강제, 그리고 트레이트(Trait)를 통한 다형성 제공을 다룬다.

5문단 〔실제 활용 및 산업적 파급 효과〕

이러한 이점 덕분에 Linux 커널 드라이버, 미국 정부 권장, WebAssembly 연동, 백엔드 시스템 등 다양한 산업 분야에서 Rust의 채택이 가속화되고 있음을 구체적 사례로 보여준다.

6문단 〔한계 인정 및 최종 평가〕

가파른 학습 곡선과 컴파일러와의 싸움이라는 단점을 인정하면서도, '컴파일 타임 복잡성을 감수하고 런타임 안전성을 얻는' 타협은 현대 IT 인프라에서 옳은 선택이었음을 평가하며 마무리한다.

■ 핵심 개념

- 제로 코스트 추상화
  사용하지 않는 기능에 대가를 지불하지 않고, 사용하는 기능은 손으로 작성한 코드와 동일한 성능을 내는 Rust의 설계 원칙
- 소유권(Ownership)
  모든 값이 하나의 소유자를 가지며, 소유자가 스코프를 벗어나면 값이 자동 해제되고, 소유권 이전 시 이전 소유자의 접근을 막아 메모리를 안전하게 관리하는 시스템
- 빌림(Borrowing)과 공유 xor 가변
  소유권 이전 없이 참조를 허용하는 메커니즘. 불변 참조는 여러 개 가능하나 가변 참조는 하나만 가능하고 둘은 공존할 수 없어 데이터 경쟁을 막는 규칙
- 수명(Lifetime)
  참조가 가리키는 대상보다 오래 살아남아 매달린 포인터가 되는 것을 방지하기 위해 컴파일러가 추적하는 참조의 유효 기간 정보
- 트레이트(Trait)
  상속을 대신하여 다형성을 제공하는 인터페이스 메커니즘으로, 정적/동적 디스패치를 선택해 성능과 유연성의 균형을 잡음

■ 논리적 흐름

문제 제기(기존 언어들의 한계) → 해결 방안 제시(Rust의 소유권/빌림/수명 시스템) → 안전성 확장(논리적 오류 방지 타입 시스템) → 효과 및 활용(산업계 채택 가속) → 최종 평가(단점 존재하지만 올바른 타협). 이는 전형적인 '문제-해결-확장-평가' 구조로, 기술적 원리가 어떻게 실제 가치로 이어지는지를 논리적으로 연결하고 있다.

▣ 문제 해설

1. 정답 ⑤

▶ 핵심 지문에 명시된 기술적 세부 사항의 정확한 의미 파악. 특히 '정적 디스패치'와 '동적 디스패치'의 차이를 이해하고, 선택지에서 이를 '성능 저하 없이'라는 절대적 표현으로 과장했는지 확인해야 함.

이 문항은 지문 전반에 걸친 Rust의 특징과 작동 원리를 정확히 파악하고 있는지 묻는 문제이다. 5번 선택지는 Rust가 트레이트를 통해 다형성을 제공한다는 사실은 맞지만, 이것이 '성능 저하 없이 유연성을 확보하는 메커니즘'이라고 단정 지은 점에서 지문과 일치하지 않는다. 지문에 따르면 Rust는 정적 디스패치(단조화)와 동적 디스패치(dyn Trait) 중 '상황에 맞는 방식을 선택할 수 있어 성능과 유연성 사이의 균형을 잡는다'고 되어 있다. 동적 디스패치를 선택할 경우 런타임 비용(성능 저하)이 발생하므로 무조건 '성능 저하 없이' 유연성을 확보한다는 표현은 지문의 내용과 모순된다.

〔 선택지 분석 〕

① × 지문 1문단에 'C나 C++와 같이... 수동 메모리 관리 방식이 주를 이루었다. 이는 뛰어난 실행 성능을 보장하지만, 널 포인터 역참조나 매달린 포인터와 같은 치명적인 버그를 발생시킬 위험을 내포한다'는 내용과 정확히 일치한다.

② × 지문 2문단에 '소유권이 다른 변수로 이동하면 이전 소유자는 해당 값을 사용할 수 없어, 이중 해제나 잘못된 참조를 컴파일러가 원천적으로 차단한다'는 내용과 정확히 일치한다.

③ × 지문 1문단(GC의 런타임 일시 정지로 인한 실시간 처리 부적합)과 3문단(Rust는 데이터 경쟁을 타입 시스템 수준에서 정적 분석하여 컴파일을 거부함)의 내용을 종합한 것으로, 지문과 정확히 일치한다.

④ × 지문 4문단의 'null 참조를 언어 차원에서 배제하여... 값의 부재는 Option<T>로, 실패 가능성은 Result<T, E>로 타입 수준에서 명시한다. 패턴 매칭을 통해 모든 경우의 수를 빠짐없이 처리하도록 강제한다'는 내용과 정확히 일치한다.

⑤ ○ 지문 4문단에 따르면 Rust는 '정적 디스패치를 통한 단조화(monomorphization)와 동적 디스패치(dyn Trait) 중 상황에 맞는 방식을 선택할 수 있어 성능과 유연성 사이의 균형을 잡는다'고 되어 있다. 동적 디스패치는 유연성을 제공하지만 약간의 성능 저하(런타임 비용)를 감수하는 방식이므로, '성능 저하 없이 유연성을 확보'한다는 선택지의 표현은 지문 내용과 일치하지 않는다.

2. 정답 ⑤

▶ 핵심 지문에 제시된 기술적 개념(빌림, 트레이트, 패턴 매칭, 소유권, 수명)의 정확한 작동 원리와 인과관계를 파악하고, 섞여 있는 정보의 참거짓을 판별해야 합니다.

이 문항은 윗글의 내용과 일치하는 것을 찾는 사실 확인 문항입니다. 5번은 Rust가 가비지 컬렉터(GC)를 사용하지 않고도 소유권 시스템을 통해 메모리를 자동으로 해제하여 런타임 일시 정지 문제를 해결하며, 동시에 수동 메모리 관리(C/C++)에 필적하는 고성능을 보장한다는 지문의 핵심 주장을 정확히 요약

하고 있습니다. 다른 선택지들은 지문의 개별 개념들을 혼합하여 인과관계나 작동 방식을 왜곡했습니다.

〔 선택지 분석 〕

① × 오답입니다. 지문에 따르면 '불변 참조는 동시에 여러 개 존재할 수 있습니다.' 즉, 다수의 주체가 동일한 데이터를 동시에 읽으려 할 때(불변 참조)는 직렬화할 필요 없이 동시 처리가 가능합니다. 가변 참조가 단 하나만 존재해야 하는 것은 데이터를 '변경'할 때의 규칙입니다.

② × 오답입니다. 트레이트를 통한 정적 디스패치와, 모든 경우의 수를 빠짐없이 처리하도록 강제하는 패턴 매칭은 서로 다른 독립적인 기능입니다. 지문에서 패턴 매칭은 '값의 부재(Option〈T〉)와 실패 가능성(Result〈T, E〉)'을 처리할 때 언급되었습니다.

③ × 오답입니다. 지문은 null 참조를 배제한 것과 Option〈T〉, Result〈T, E〉를 사용한 것을 모두 '타입 시스템을 통해 논리적 오류를 방지하는 예시'로 나열했을 뿐, null 배제가 패턴 매칭의 전제 조건이라는 인과관계는 언급하지 않았습니다. 또한 패턴 매칭은 모든 경우의 수를 처리하도록 강제하는 기능 자체입니다.

④ × 오답입니다. 소유권 이동을 통한 이중 해제 차단과 수명 정보를 통한 매달린 포인터 방지는 모두 컴파일 시점에 이루어지는 안전성 보장 기능이지만, 지문에서 이 둘이 '동일한 컴파일 시점 메커니즘'이라고 설명하지는 않았습니다. 이는 각각 소유권 시스템과 수명(Lifetime) 시스템이라는 별개의 메커니즘입니다.

⑤ ○ 정답입니다. 지문의 1문단(GC의 런타임 일시 정지 문제, 수동 메모리 관리의 성능)과 2문단(GC 없이 소유권 시스템으로 메모리 자동 해제 및 안전성, 고성능 달성)의 내용이 정확히 결합된 표현입니다.

3. 정답 ⑤

▶ 핵심 Rust의 핵심 안전성 메커니즘(소유권, 빌림, 수명)이 '컴파일 시점'의 정적 분석을 통해 런타임 오류를 사전에 방지한다는 지문의 논리를 정확히 파악해야 합니다.

이 문항은 지문에 제시된 Rust의 설계 방식이 어떤 문제의식이나 배경에서 비롯되었는지를 논리적으로 추론하는 문제입니다. 5번은 Rust가 매달린 포인터 문제를 해결하기 위해 '수명(Lifetime)' 정보를 부여하고 '컴파일러'가 이를 추적하도록 한다는 점에 주목합니다. 지문에서 다른 언어들은 '런타임 락이나 테스팅에 의존'하여 동시성 버그를 해결하려 했으나 Rust는 이를 '컴파일 시점'에 정적 분석하여 사전 제거한다고 설명합니다. 따라서 Rust가 참조의 유효성 검사를 런타임이 아닌 컴파일 타임으로 가져온 것은, 런타임 검증 방식의 근본적 한계를 극복하기 위한 의도적인 설계 판단이라고 추론하는 것이 타당합니다.

〔 선택지 분석 〕

① × 오답입니다. 지문에 따르면 '두려움 없는 동시성'은 데이터 경쟁이 타입 시스템 수준에서 정적 분석되어 컴파일을 거부함으로써 가능해집니다. 단일 스레드 환경은 애초에 동시성이 발생하지 않는 환경이므로, 다중 스레드 환경에서 발생할 수 있는 동시성 버그를 사전에 제거했다는 의미의 '두려움 없는 동시성'과는 논리적 맥락이 다릅니다.

② × 오답입니다. 지문에서 Rust가 상속을 배제하고 트레이트를 도입한 이유로 '상속의 런타임 비용'을 언급한 바 없습니다. 지문은 상속 대신 트레이트를 통해 정적/동적 디스패치를 선택하여 '성능과 유연성 사이의 균형'을 잡는다고 설명합니다. 상속 자체가 제로 코스트 추상화에 위배된다는 논리적 연결은 지문에서 찾을 수 없습니다.

③ × 오답입니다. 지문에서 GC 기반 언어가 실시간 처리에 부적합했던 이유는 '예측 불가능한 런타임 일시 정지' 때문이며, 이는 GC가 메모리 안전성을 확보하지 못해서가 아니라 GC의 작동 방식(런타임 비용) 때문입니다. GC는 메모리 안전성을 보장하지만 런타임 성능의 예측성이 떨어진다는 지문의 내용과 모순됩니다.

④ × 오답입니다. 지문은 Rust가 이중 해제나 잘못된 참조를 '컴파일러가 원천적으로 차단'한다고 명시합니다. 즉, 소유권 이동 후 이전 소유자가 접근하는 코드는 컴파일 자체가 되지 않으므로 프로그램이 실행조차 되지 않습니다. '런타임 패닉을 유발하는 방식'이라는 서술은 지문의 핵심인 '컴파일 시점 안전성 보장'과 정면으로 모순됩니다.

⑤ ○ 정답입니다. 지문은 다른 언어들이 런타임에 의존하여 버그를 처리하던 방식과 대조적으로, Rust는 컴파일러가 수명을 추적하여 매달린 포인터를 방지한다고 설명합니다. 이는 런타임 검증 방식(락이나 테스팅)으로는 근본적인 해결에 한계가 있다는 판단 하에, 컴파일 시점의 정적 분석으로 안전성을 확보하려는 Rust의 설계 철학과 논리적으로 부합합니다.

4. 정답 ②

▶ 핵심 밑줄 친 표현의 전후 문맥(가파른 학습 곡선, 컴파일러와의 싸움 vs 안전성과 성능의 이점)을 파악하여, '컴파일 타임 복잡성'이 가리키는 구체적 행위와 '런타임 안전성'이 가리키는 결과를 짝지을 수 있어야 합니다.

밑줄 친 ㉣ '컴파일 타임 복잡성을 감수하고 런타임 안전성을 얻는다'는 문장은 마지막 문단에서 언급된 Rust의 '가파른 학습 곡선'과 '컴파일러와 싸우며 코드를 작성해야 하는 경험'이라는 단점을 극복하고 얻는 이점을 요약한 것입니다. 즉, 개발자가 소유권, 빌림, 수명 등의 복잡한 규칙을 학습하고 컴파일러의 엄격한 검사를 통과하기 위해 노력(컴파일 타임 복잡성 감수)해야 하지만, 그 대가로 프로그램 실행 중(런타임)에 발생할 수 있는 메모리 오류나 데이터 경쟁을 원천적으로 방지(런타임 안전성 획득)할 수 있다는 의미입니다. 따라서 2번이 가장 적절합니다.

〔 선택지 분석 〕

① × 지문에 따르면 Rust는 가비지 컬렉터(GC)를 '도입'한 것이 아니라 '없이도' 메모리 안전성을 달성한 언어입니다. GC 도입으로 인한 런타임 일시 정지를 감수한다는 내용은 지문에서 언급된 Java나 Go의 특징이며 Rust의 설계 철학과 반대됩니다.

② ○ 지문의 마지막 문단에서 언급된 '소유권, 빌림, 수명 등의 개념을 동시에 익혀야 하는 가파른 학습 곡선'과 '컴파일러와 싸우며 코드를 작성해야 하는 경험'이 바로 '컴파일 타임 복잡성'에 해당합니다. 또한, 이를 통해 얻게 되는 '실행 시점의 메모리 안전성과 데이터 경쟁 방지'가 '런타임 안전성'에 해당하므로 가장 적절한 해석입니다.

③ × 지문에서 Rust는 컴파일 시점 정적 분석의 범위를 '축소'하는 것이 아니라, 오히려 타입 시스템 수준에서 정적 분석을 강화하여 동시성 버그를 '사전에 제거'한다고 설명합니다. 런타임 락이나 테스팅에 의존하는 것은 지문에 명시된 '다른 언어들'의 방식입니다.

④ × 지문에 따르면 Rust는 고수준 언어의 편리함을 누리면서도 저수준 언어의 성능을 활용할 수 있는 '제로 코스트 추상화'를 채택했습니다. 고수준 언어의 편리한 추상화를 '포기'한다는 내용은 지문의 내용과 정반대입니다.

⑤ × 지문에서 Rust는 트레이트를 통해 정적 디스패치와 동적 디스패치 중 '상황에 맞는 방식을 선택'할 수 있다고 하였습니다. 정적 디스패치 '만을' 허용하여 동적 디스패치의 유연성을 희생한다는 설명은 지문의 내용과 모순됩니다.

▣ 학습 팁

1. 밑줄 친 표현의 의미를 묻는 문제는 그 표현이 등장한 문단의 전후 문맥을 반드시 확인해야 합니다. 밑줄 바로 앞의 단점(가파른 학습 곡선, 컴파일러와의 싸움)과 밑줄 바로 뒤의 장점(고가용성, 보안 등)을 정확히 짝지어 보면 '컴파일 타임 복잡성'과 '런타임 안전성'이 각각 무엇을 의미하는지 쉽게 유추할 수 있습니다.
2. 전문 용어가 포함된 선택지를 검증할 때는, 해당 용어가 지문에서 어떤 대상의 특징으로 설명되었는지를 정확히 기억해야 합니다. 예를 들어 '가비지 컬렉터(GC) 도입'은 Java/Go의 특징이지 Rust의 특징이 아니므로, 이를 혼동하면 오답을 고르게 됩니다.
3. 지문에서 '~만을', '~포기', '~축소'와 같이 극단적인 표현이 선택지에 등장하면 일단 의심해봐야 합니다. 지문에서 '상황에 맞게 선택', '동시에 달성' 등으로 설명한 유연한 특징을 극단적으로 제한하는 선택지는 오답일 확률이 높습니다.

프로그래밍 언어 Rust의 부상과 설계 철학

빠른 정답

1	2	3	4	
⑤	⑤	⑤	②	

총 4문항