

Korea University MatKor 스터디

Week 7: Macros

Chris Ohk

utilForever@gmail.com

- 매크로의 기초
 - 매크로 전개
 - 의도치 않은 결과
 - 반복
- 기본 제공 매크로
- 매크로 디버깅

- json! 매크로 만들기
 - 프래그먼트 타입
 - 매크로의 재귀
 - 매크로와 트레잇 함께 쓰기
 - 범위 한정과 하이지닉 매크로
 - 매크로 가져오기와 내보내기
- 매칭 중에 발생하는 구문 오류 피하기
- macro_rules!에서 더 나아가기

- Rust는 함수만 가지고 할 수 있는 수준을 뛰어넘어서 다방면으로 언어를 확장하기 위한 방법인 **매크로(Macro)**를 지원한다.
- Rust의 매크로는 패턴 매칭이 있어서 유지보수하기 좋고 손쉽게 작성할 수 있다.
- 우리는 지금까지 다양한 매크로를 사용했었다.
 - `print!`
 - `vec!`
 - `panic!`
 - `assert_eq!`

- assert_eq! 매크로의 소스 코드를 살펴보자.
 - Rust에서는 매크로를 정의할 때 주로 macro_rules!를 쓴다.
 - 정의하는 부분을 보면, assert_eq 뒤에는 !가 없다. (호출할 때만 붙이고, 정의할 때는 붙이지 않는다.)
 - 하지만 모든 매크로가 이런 식으로 정의되는 건 아니다.
 - macro_rules!를 비롯해 file!과 line! 같은 일부 매크로는 컴파일러 안에 내장되어 있다.
 - 절차적 매크로라고 하는 또 다른 접근 방식도 있다. (후반부에 다룰 예정)

```
● ● ●

macro_rules! assert_eq {
    ($left: expr, $right: expr) => ({
        match (&$left, &$right) {
            (left_val, right_val) => {
                if (*left_val == *right_val) {
                    panic!("assertion failed: (left == right \
                            (left: {:#?}, right: {:#?}))", left_val, right_val)
                }
            }
        }
    })
}
```

매크로의 기초

Korea University MatKor 스터디
Week 7: Macros

- macro_rules!로 정의한 매크로는 전적으로 패턴 매칭에 따라 움직인다.

```
● ● ●  
( pattern1 ) => ( template1 );  
( pattern2 ) => ( template2 );  
...
```

- 패턴이나 템플릿에는 어떤 괄호를 써도 되지만, 다음과 같이 쓰는 게 관례다.

```
● ● ●  
assert_eq!(gcd(6, 10), 2);  
assert_eq![gcd(6, 10), 2];  
assert_eq![{gcd(6, 10), 2}]
```

- assert_eq!를 호출할 때는 소괄호를 쓴다.
- vec!을 호출할 때는 대괄호를 쓴다.
- macro_rules!를 호출할 때는 중괄호를 쓴다.

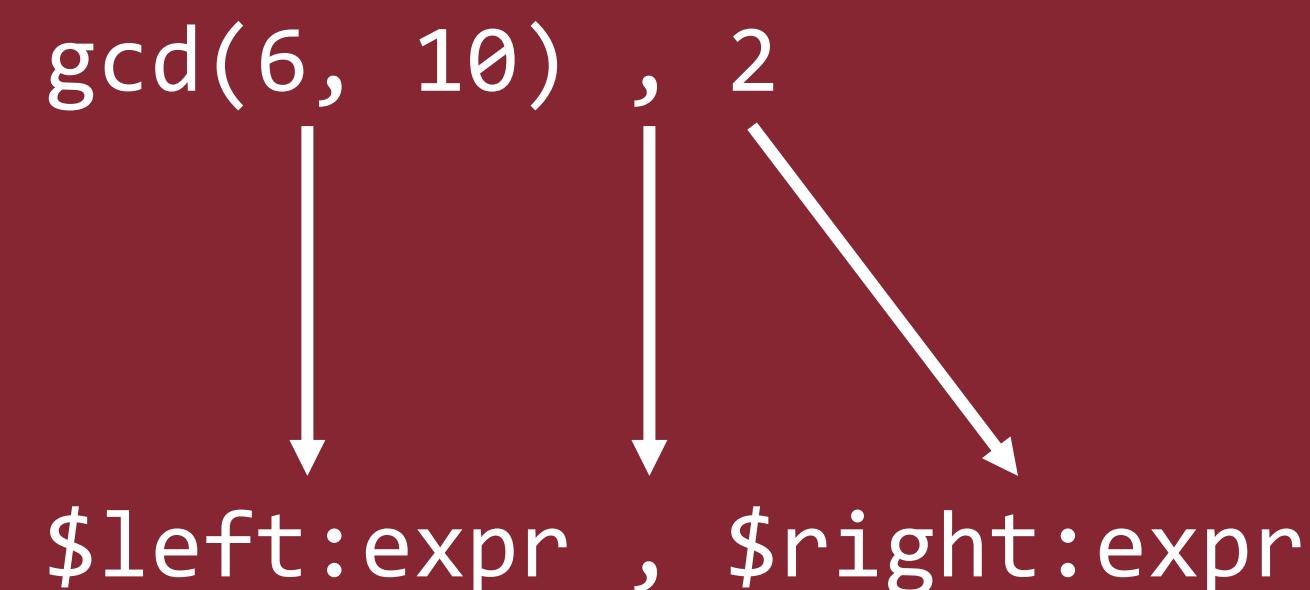
매크로의 기초

Korea University MatKor 스터디
Week 7: Macros

- 이제 매크로를 실행하는데 필요한 세부 사항들을 알아보자.
 - Rust가 정확히 어떤 식으로 프로그램에 있는 매크로 정의를 찾아 전개하는지 설명한다.
 - 매크로 템플릿을 가지고 코드를 생성하는 과정에 내재된 까다로운 부분 몇 가지를 짚어본다.
 - 패턴이 반복 구조를 처리하는 방법을 살펴본다.

- Rust에서는 컴파일 과정의 초기 단계에서 마크로 전개가 일어난다.
 - 컴파일러는 소스 코드를 처음부터 읽어나가면서 마크로를 정의하고 전개한다.
 - 프로그램의 나머지 부분을 보기 전에 각 마크로 호출을 전개하므로 아직 정의되지 않은 마크로는 호출할 수 없다.

- Rust는 먼저 인수가 패턴과 매칭되는지 본다.
 - Rust의 매크로 패턴은 언어 속의 작은 언어라고 할 수 있다. 기본적으로는 코드를 매칭하는 정규 표현식이라고 보면 된다. 그러나 정규 표현식이 문자에 작용한다면 패턴은 토큰, 즉 수/이름/구두점 등 Rust 프로그래밍의 빌딩 블록에 작용한다.
 - 패턴에 포함된 프래그먼트(Fragment) `$left:expr`은 표현식을 매칭해서 `$left`라는 이름을 배정하라고 지시한다.
 - 그런 다음 패턴에 있는 쉼표를 `gcd`의 인수 뒤에 있는 쉼표와 매칭한다.
 - 패턴에는 흥미로운 매칭 동작을 유발하는 특수 문자가 몇 가지 있다.
이 외에 쉼표와 같은 다른 문자는 전부 적힌 그대로 매칭되어야 하며, 그렇지 않으면 매칭은 실패한다.
 - 끝으로 표현식 2를 매칭하고 여기에 `$right`라는 이름을 배정한다.



- 패턴이 인수를 모두 매칭하고 나면 그에 대응하는 **템플릿(Template)**을 전개한다.
 - Rust는 `$left`와 `$right`를 매칭 과정에서 찾은 코드 프래그먼트로 대체한다.
 - 출력 템플릿에 프래그먼트 타입을 넣는 실수를 하는 경우가 종종 있으므로 주의하자.

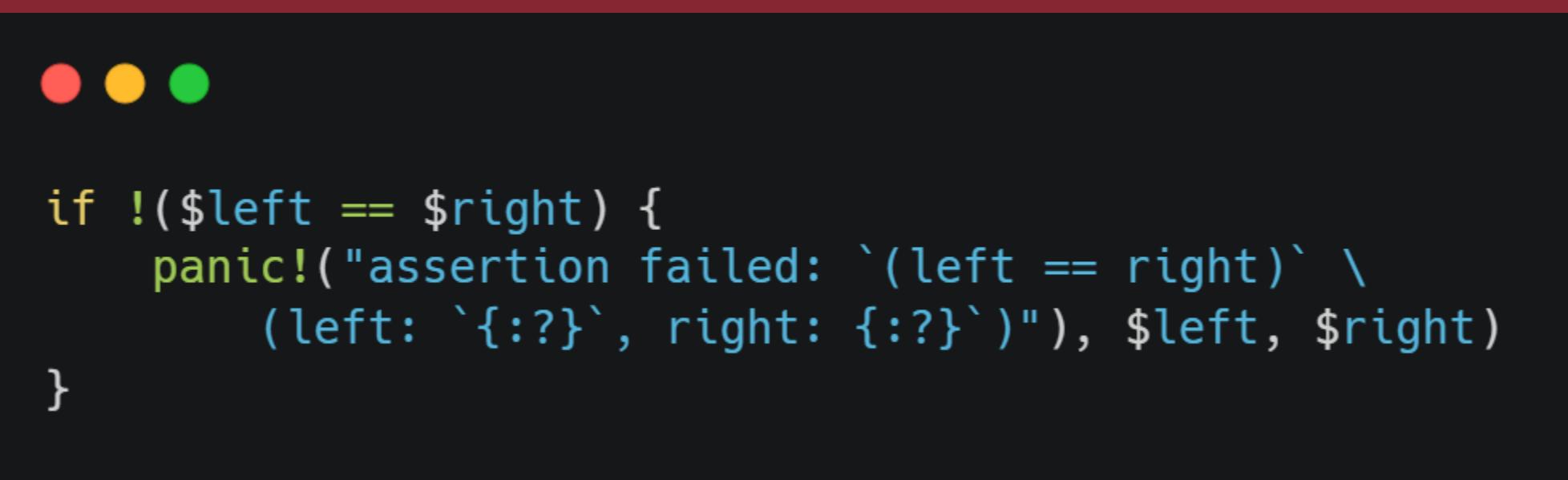
```
{  
    match (&$left, &$right) {  
        (left_val, right_val) => {  
            if !(*left_val == *right_val) {  
                panic!("assertion failed: (left == right \\\n                      (left: {:{}, right: {:{}})", left_val, right_val)  
            }  
        }  
    }  
})
```

gcd(6, 10)으로 대체됨
2로 대체됨

의도치 않은 결과

Korea University MatKor 스터디
Week 7: Macros

- 코드 프래그먼트를 템플릿 안에 끼워 넣는 동작은 일반적인 코드와는 미묘하게 다르다.
- 첫째, 왜 이 매크로는 변수 `left_val`과 `right_val`을 만들고 있는 걸까?
템플릿을 다음처럼 단순하게 만들 수 없는 이유라도 있는 걸까?



A terminal window with three colored dots (red, yellow, green) at the top. The main area contains the following text:

```
if !($left == $right) {
    panic!("assertion failed: `($left == right)` \
        (left: `{:?}` , right: `{:?}` )"), $left, $right
}
```

- 코드 분석
 - 매크로 `assert_eq!(letters.pop(), Some('z'))`를 호출할 때 어떻게 전개되는지 생각해보자.
 - Rust는 매칭된 표현식을 템플릿 이곳저곳에 끼워넣는다.
 - 그런데 오류 메시지를 만들 때마다 표현식을 처음부터 다시 평가하는 건 그리 좋은 생각이 아닌 거 같다.
그 이유는 `letters.pop()`이 벡터에서 값을 제거하므로 호출할 때마다 다른 값을 산출하기 때문이다.
 - 따라서 실제 매크로는 `$left`와 `$right`를 한 번만 계산해서 그 값을 저장해 둔다.

의도치 않은 결과

Korea University MatKor 스터디
Week 7: Macros

- 코드 프래그먼트를 템플릿 안에 끼워 넣는 동작은 일반적인 코드와는 미묘하게 다르다.
- 둘째, 왜 이 매크로는 `$left`와 `$right`의 값을 참조하는 레퍼런스를 빌려오는 걸까?
그냥 다음처럼 값을 변수에 저장하면 안되는 걸까?



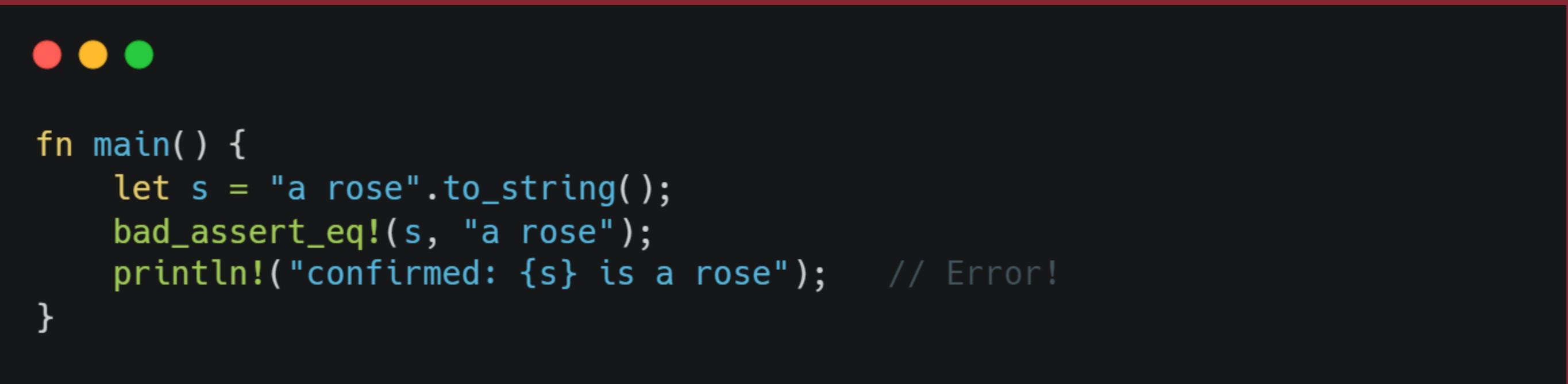
```
macro_rules! bad_assert_eq {
    ($left:expr, $right:expr) => ({
        match ($left, $right) {
            (left_val, right_val) => {
                if !(left_val == right_val) {
                    panic!("assertion failed" /* ... */);
                }
            }
        });
}
```

의도치 않은 결과

Korea University MatKor 스터디
Week 7: Macros

- **코드 분석**

- 이 코드는 지금까지 썼던 것처럼 매크로 인수가 정수인 경우에는 잘 동작한다.
그러나 `String` 변수를 `$left`나 `$right`에 넘기면, 이 코드는 값을 변수 밖으로 옮긴다.
- 단정문이 값을 옮기는 걸 바라지 않는다면 매크로가 레퍼런스를 빌려와야 한다.



```
fn main() {
    let s = "a rose".to_string();
    bad_assert_eq!(s, "a rose");
    println!("confirmed: {} is a rose"); // Error!
}
```

- Rust 표준에 있는 매크로 `vec!`은 두 가지 형태로 쓸 수 있다.

```
● ● ●  
// Usage 1  
let buffer = vec![0_u8; 1000];  
  
// Usage 2  
let numbers = vec!["udon", "ramen", "soba"];
```

- 이 매크로는 다음처럼 구현할 수 있다.

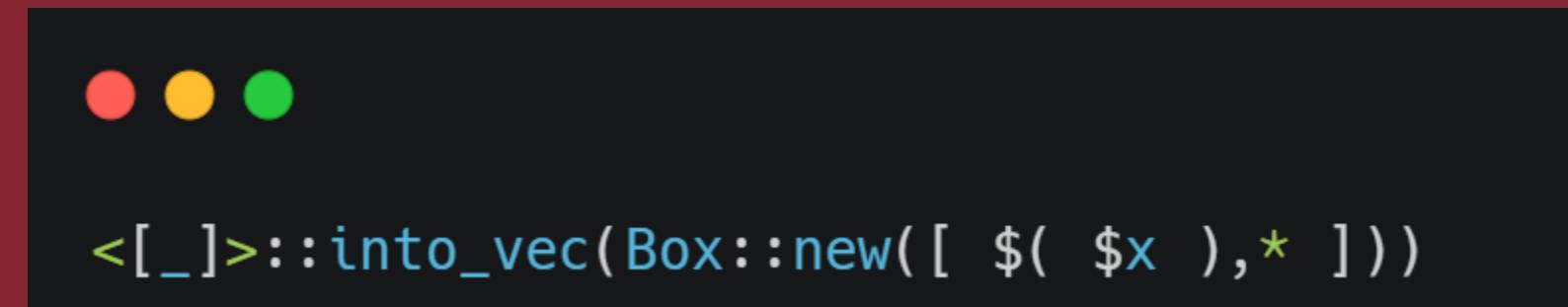
```
macro_rules! vec {
    ($elem:expr ; $n:expr) => {
        ::std::vec::from_elem($elem, $n)
    };
    ( $($x:expr),* ) => {
        <[_]>::into_vec(Box::new([ $($x),* ])))
    };
    ( $($x:expr ),+ ,) => {
        vec![ $($x),* ]
    };
}
```

- 세 가지 규칙이 어떤 식으로 동작하는지 알아보자.
 - 먼저 인수 1, 2, 3을 첫번째 규칙의 패턴 `$elem:expr ; $n:expr`과 매칭시켜 본다.
1은 표현식이 맞지만, 그 뒤에 패턴이 요구하는 세미콜론이 없으므로 매칭에 실패한다.
 - 따라서 두번째 규칙으로 넘어가서 같은 일을 반복하며, 매칭되는 규칙이 있을 때까지 처리를 이어간다.
매칭되는 규칙이 없으면 오류가 발생한다.
 - 첫번째 규칙은 `vec![0u8; 1000]`과 같은 형태를 처리한다.
여기서는 그런 일을 하는 표준 함수 `std::vec::from_elem`을 써서 간단히 해결한다.
 - 두번째 규칙은 `vec!["udon", "ramen", "soba"]`와 같은 형태를 처리한다.
패턴 `$($x:expr),*`은 지금껏 본 적 없는 반복 기능을 사용한다.
이 패턴은 쉼표로 구분된 0개 이상의 표현식을 매칭한다.
 - 일반적으로 `$(PATTERN),*` 구문은 각 아이템이 PATTERN과 매칭되는
쉼표로 구분된 임의의 리스트를 매칭하는 데 사용한다.

- 정규 표현식에는 , * 반복자가 없지만 여기에 쓰인 *의 의미는 정규 표현식과 같다.
 - * : 0개 이상, + : 1개 이상, ? : 0개 또는 1개

패턴	뜻
$\$(\dots)^*$	구분 기호 없이 0회 이상 매칭한다.
$\$(\dots),^*$	쉼표를 구분 기호 삼아 0회 이상 매칭한다.
$\$(\dots);^*$	세미콜론을 구분 기호 삼아 0회 이상 매칭한다.
$\$(\dots)^+$	구분 기호 없이 1회 이상 매칭한다.
$\$(\dots),^+$	쉼표를 구분 기호 삼아 1회 이상 매칭한다.
$\$(\dots);^+$	세미콜론을 구분 기호 삼아 1회 이상 매칭한다.
$\$(\dots)?$	구분 기호 없이 0회 또는 1회 매칭한다.
$\$(\dots),?$	쉼표를 구분 기호 삼아 0회 또는 1회 매칭한다.
$\$(\dots);?$	세미콜론을 구분 기호 삼아 0회 또는 1회 매칭한다.

- 코드 프래그먼트 `$x`는 표현식 하나가 아니라 표현식 리스트다.
이 규칙의 템플릿도 반복 문법을 쓴다.



```
<[_]>::into_vec(Box::new([ $( $x ),* ]))
```

- 이번에도 딱 그런 일을 하는 표준 메소드를 써서 해결한다.
이 코드는 박스 처리된 배열을 만든 다음 `[T]:::into_vec` 메소드를 써서 이를 벡터로 바꾼다.
- 맨 앞에 있는 `<[_]>`는 '무언가의 슬라이스' 타입을 적을 때 쓰는 조금 낯선 방법으로
이렇게 해두면 러스트가 요소 타입을 추론해준다. 타입의 이름이 평범한 식별자로 된 경우에는
표현식 그대로 써도 되지만, `fn()`이나 `&str`, `[_]` 같은 타입은 반드시 꺭쇠 괄호를 둘러야 한다.

- 템플릿에 대한 설명

- 템플릿 맨 끝에도 `$($x)`, `*` 같은 식으로 반복이 들어가 있다. 이 `$(...)`, `*`는 패턴에서 봤던 구문과 똑같다. 즉, `$x`와 매칭된 표현식 리스트를 반복 처리해서 쉼표로 구분한 뒤 전부 템플릿 안에 끼워 넣는다.
- 이 경우에는 반복 처리된 출력이 입력과 똑같은 형태를 띤다.
그러나 꼭 이렇게 해야 하는 건 아니다. 규칙을 다음처럼 작성할 수도 있다.

```
( $($x: expr ),* ) => {
    let mut v = Vec::new();
    $($v.push($x); )*
    v
}
```

- 이 템플릿에서 `$(`v.push($x); `)*`라고 된 부분은 `$x`에 있는 표현식마다 `v.push()` 호출을 끼워 넣는다. 매크로의 갈래는 여러 표현식으로 전개될 수 있지만, 여기서는 하나의 표현식만 있으면 되므로 하나의 블록으로 만들었다.

- 참고 사항

- \$(...), *를 쓰는 패턴은 후행 쉼표 옵션을 자동으로 지원하지 않는다.
- 하지만 추가 규칙을 넣어서 후행 쉼표를 지원하는 흑마법이 있다. vec! 매크로의 세번째 규칙이 하는 일이 바로 그것이다.

```
● ● ●  
  
( $( $x: expr ),+ , ) => {  
    vec![ $( $x ),* ]  
}
```

- \$(...), + , 를 써서 맨 뒤에 쉼표가 붙은 리스트를 매칭한다. 그런 다음 템플릿에서 맨 뒤에 붙은 쉼표를 빼고 vec!을 재귀적으로 호출한다. 이렇게 하고 나면 두번째 규칙이 매칭된다.

- Rust 컴파일러는 다양한 매크로를 제공한다. (`rustc`에 하드코딩되어 있다.)
 - `file!()`, `line!()`, `column!()`
 - `file!()`은 현재 파일 이름을 나타내는 문자열 리터럴로 전개된다.
 - `line()`과 `column!()`은 현재 행과 열을 나타내는 (1부터 시작하는) `u32`로 전개된다.
 - `stringify!(...토큰...)`
 - 주어진 토큰이 담긴 문자열 리터럴로 전개된다. `assert!`는 이 매크로를 써서 단정문의 코드가 담긴 오류 메시지를 생성한다.
 - 인수에 있는 매크로 호출은 전개되지 않는다. 따라서 `stringify!(line!())`은 문자열 "line!"으로 전개된다.
 - 러스트는 주어진 토큰을 가지고 문자열을 생성하기 때문에 줄 바꿈이나 주석은 문자열에 들어가지 않는다.
 - `concat!(str0, str1, ...)`
 - 인수를 전부 연결해 만든 하나의 문자열 리터럴로 전개된다.

- 또한 Rust는 빌드 환경을 질의하기 위한 매크로도 정의해 두고 있다.

- `cfg!(...)`
 - 현재 빌드 구성이 괄호 안의 조건과 일치하면 불 상수 `true`로 전개된다.
예를 들어, 디버그 단정문을 캔 채로 컴파일하는 중이면 `cfg!(debug_assertions)`는 참이다.
- `env!("VAR_NAME")`
 - 컴파일 시점에 지정한 환경 변수의 값이 담긴 문자열로 전개된다. 변수가 존재하지 않으면 컴파일 오류가 발생한다.
 - 예를 들어, 크레이트의 현재 버전 문자열을 가져오려면 다음처럼 작성하면 된다.



```
let version = env!("CARGO_PKG_VERSION");
```

- 환경 변수의 전체 목록은 <https://doc.rust-lang.org/cargo/reference/environment-variables.html>을 참고
- `option_env!("VAR_NAME")`
 - `env!`와 똑같지만 `Option<&'static str>`를 반환한다는 점이 다르다. 지정한 변수가 설정되어 있지 않으면 `None`이 된다.

- 다음 내장 매크로는 다른 파일에 있는 코드나 데이터를 가져올 때 쓴다.

- `include!("file.rs")`

- 지정한 파일의 내용으로 전개된다. 이때 파일은 유효한 Rust 코드, 즉 표현식이나 일련의 아이템만 담고 있어야 한다.

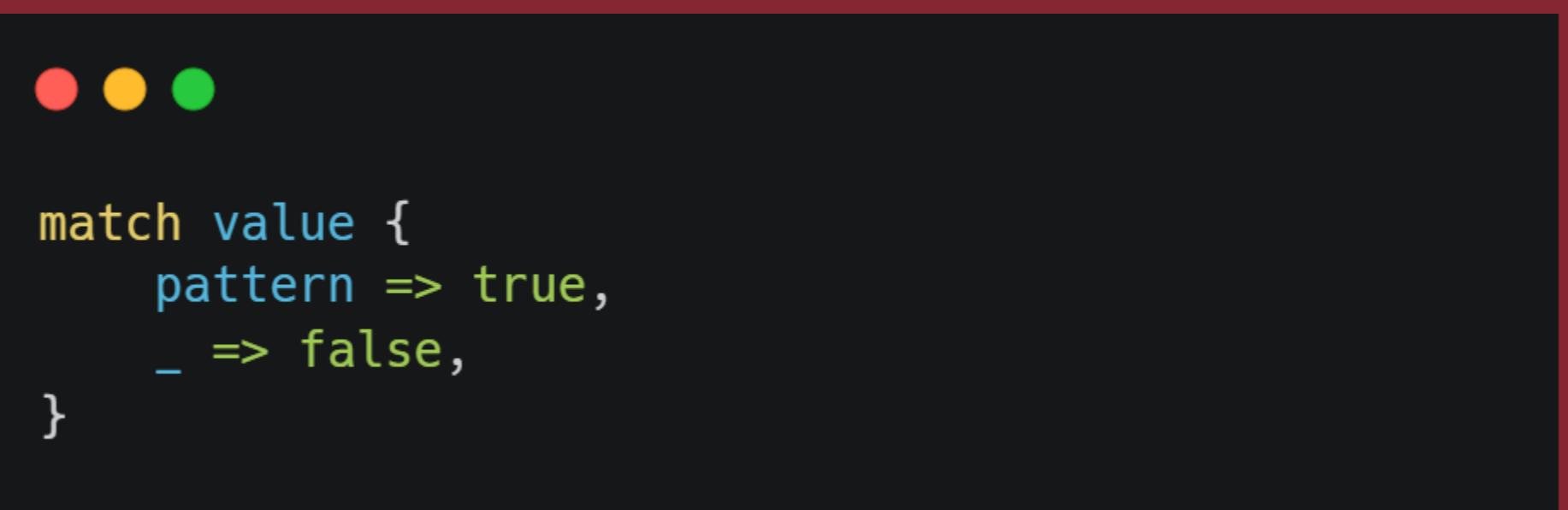
- `include_str!("file.txt")`

- 지정한 파일의 텍스트가 담긴 `&static str`로 전개된다. 다음과 같은 식으로 쓰면 된다.

```
● ● ●  
const COMPOSITOR_SHADER: &str =  
    include_str!("../resources/compositor.glsl");
```

- 파일이 존재하지 않거나 유효한 UTF-8이 아니면 컴파일 오류가 발생한다.
- `include_bytes!("file.dat")`
- 똑같지만 파일을 UTF-8 텍스트가 아니라 바이너리 데이터로 취급한다는 점이 다르다. 결과는 `&static [u8]`이다.

- 이외에도 여러 가지 편리한 매크로를 제공한다.
 - `todo!()`, `unimplemented!()`, `unreachable!()`
 - `panic!()`과 똑같지만 전달하는 의도가 다르다.
 - `todo!()`는 아직 작성하지 않은 코드의 아이디어를 전달할 때 쓴다.
 - `unimplemented!()`는 아직 처리하지 않은 `if`문과 `match`문 등에서 쓴다.
 - `unreachable!()`은 도달할 수 없는 코드라는 걸 나타내는데 쓴다.
- `matches!(value, pattern)`
- 값을 패턴과 비교해서 일치하면 `true`를, 그렇지 않으면 `false`를 반환한다. 다음처럼 작성하는 것과 같다.



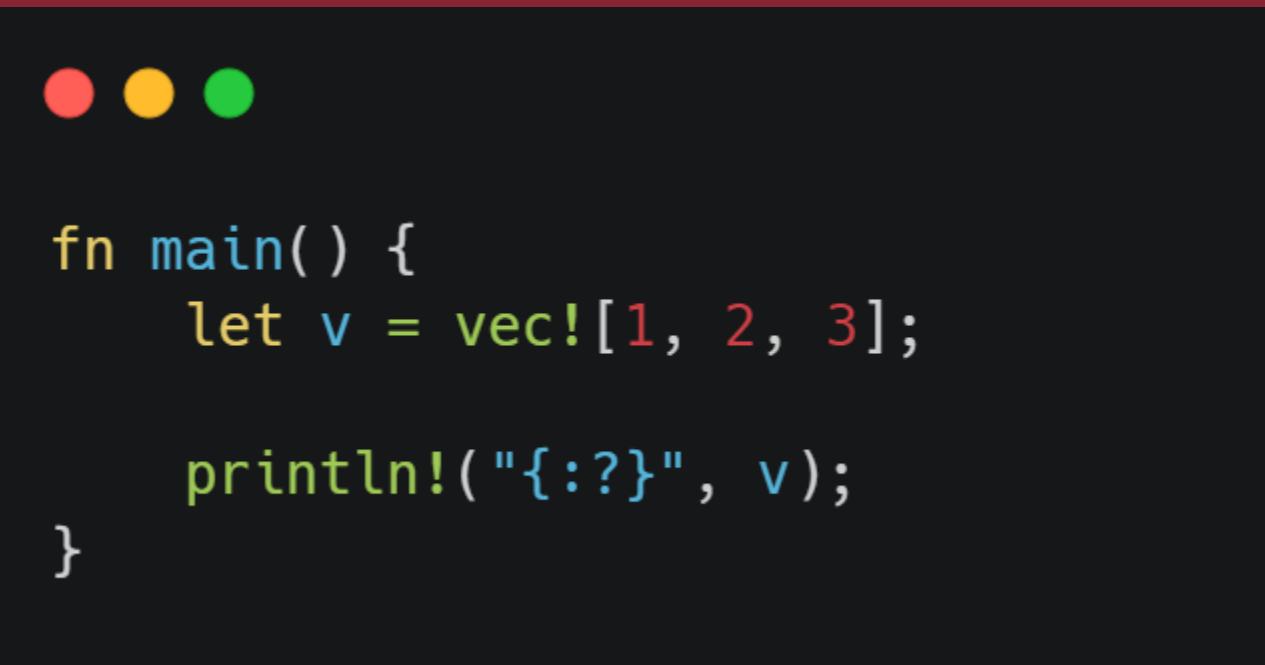
A terminal window with three colored dots (red, yellow, green) at the top. The main area contains the following Rust code:

```
match value {  
    pattern => true,  
    _ => false,  
}
```

매크로 디버깅

Korea University MatKor 스터디
Week 7: Macros

- cargo-expand 크레이트를 사용하면 매크로가 전개된 코드를 볼 수 있다.
 - 설치 방법 : cargo install cargo-expand
 - 사용 방법 : cargo expand



```
fn main() {
    let v = vec![1, 2, 3];
    println!("{:?}", v);
}
```

- cargo-expand 크레이트를 사용하면 매크로가 전개된 코드를 볼 수 있다.
 - 설치 방법 : cargo install cargo-expand
 - 사용 방법 : cargo expand

```
● ● ●

#![feature(prelude_import)]
#[prelude_import]
use std::prelude::rust_2021::*;

#[macro_use]
extern crate std;
fn main() {
    let v = <[_]>::into_vec(#[rustc_box] ::alloc::boxed::Box::new([1, 2, 3]));
    {
        ::std::io::_print(format_args!("{}?\n", v));
    };
}
```

json! 매크로 만들기

Korea University MatKor 스터디
Week 7: Macros

- json! 매크로를 점진적으로 개발하며 어떤 식으로 진행되는지 알아 보자.
먼저 JSON 데이터를 표현하기 위한 열거체를 정의해 보자.

```
#[derive(Clone, PartialEq, Debug)]
enum Json {
    Null,
    Boolean(bool),
    Number(f64),
    String(String),
    Array(Vec<Json>),
    Object(Box<HashMap<String, Json>>)
}
```

json! 매크로 만들기

Korea University MatKor 스터디
Week 7: Macros

- 아쉽게도 Json 값을 쓰는 문법은 상당히 장황하다.

```
● ● ●

let students = Json::Array(vec![
    Json::Object(Box::new(vec![
        ("name".to_string(), Json::String("Jim Blandy".to_string())),
        ("class_of".to_string(), Json::Number(1926.0)),
        ("major".to_string(), Json::String("Tibetan throat singing".to_string()))
    ].into_iter().collect())),
    Json::Object(Box::new(vec![
        ("name".to_string(), Json::String("Jason Orendroff".to_string())),
        ("class_of".to_string(), Json::Number(1702.0)),
        ("major".to_string(), Json::String("Knots".to_string()))
    ].into_iter().collect())),
]);

```

json! 매크로 만들기

Korea University MatKor 스터디
Week 7: Macros

- 이를 다음처럼 좀 더 JSON다운 문법으로 쓸 수 있으면 좋겠다.

```
● ● ●  
  
let students = json!([  
  {  
    "name": "Jim Blandy",  
    "class_of": 1926,  
    "major": "Tibetan throat singing"  
  },  
  {  
    "name": "Jason Orendorff",  
    "class_of": 1702,  
    "major": "Knots"  
  }  
]);
```

- 매크로를 작성할 때 먼저 할 일은 원하는 입력을 매칭하거나 파싱하는 방법을 찾는 것이다.
 - JSON 데이터에는 객체, 배열, 수 등 다양한 종류의 값이 있다.
 - 실제로 JSON 타입별로 규칙을 하나씩 마련해두면 될 거 같은 생각이 든다.



```
● ● ●

macro_rules! json {
    (null)      => { Json::Null };
    ([ ... ])   => { Json::Array(...); };
    ({ ... })   => { Json::Object(...); };
    (???)       => { Json::Boolean(...); };
    (???)       => { Json::Number(...); };
    (???)       => { Json::String(...); };
}
```

프로그먼트 타입

Korea University MatKor 스터디
Week 7: Macros

- 그러나 살짝 부족하다. 매크로 패턴으로는 마지막 세 경우를 구분할 방법이 없기 때문이다.
 - 이 부분을 처리하는 방법은 이후에 살펴보기로 하자.
 - 우선 처음 세 경우는 깔끔하게 다른 토큰으로 시작하므로 먼저 살펴보도록 하자.
- 첫번째 규칙은 이미 완벽히 동작한다.



```
● ● ●

macro_rules! json {
    (null) => {
        Json::Null
    }
}

#[test]
fn json_null() {
    assert_eq!(json!(null), Json::Null);
}
```

프로그먼트 타입

Korea University MatKor 스터디
Week 7: Macros

- JSON 배열 지원을 추가하기 위해서는 요소를 expr과 매칭해 보아야 할 것이다.

```
● ● ●

macro_rules! json {
    (null) => {
        Json::Null
    }
    ([ $( $element:expr ),* ] ) => {
        Json::Array(vec![ $( $element ),* ])
    };
}
```

프로그먼트 타입

Korea University MatKor 스터디
Week 7: Macros

- 안타깝게도 이 코드로는 모든 JSON 배열을 매칭하지 못한다.
 - 패턴 `$($element:expr)`, *는 '쉼표로 구분된 Rust 표현식 리스트'라는 뜻이다.
 - 그러나 JSON의 오브젝트는 유효한 Rust 표현식이 아니다. 따라서 매칭되지 않는다.



```
#[test]
fn json_array_with_json_element() {
    let macro_generated_value = json!(
        [
            {
                "pitch": 440.0
            }
        ]
    );
    let hand_coded_value =
        Json::Array(vec![
            Json::Object(Box::new(Json::Object(vec![
                ("pitch".to_string(), Json::Number(440.0))
            ]).into_iter().collect())))
        ]);
    assert_eq!(macro_generated_value, hand_coded_value);
}
```

프로그먼트 타입

Korea University MatKor 스터디
Week 7: Macros

- Rust는 여러 가지 프로그먼트 타입을 지원한다.

프로그먼트 타입	매칭(예)	뒤에 올 수 있는 것...
expr	표현식 : 2 + 2, “udon”, x.len()	=> , ;
stmt	후행 세미콜론을 포함하지 않는 표현식이나 선언문 (쓰기 어려우니 대신 expr이나 block을 써보자)	=> , ;
ty	타입 : String, Vec<u8>, (&str, bool), dyn Read + Sync	=> , ; { : > as where
path	경로 : ferns, ::std::sync::mpsc	=> , ; { : > as where
pat	패턴 : _, Some(ref x)	=> , = if in
item	아이템 : struct Point { x: f64, y: f64 }, mod ferns;	전부
block	블록 : { s += “ok\n”; true }	전부

프로그먼트 타입

Korea University MatKor 스터디
Week 7: Macros

- Rust는 여러 가지 프로그먼트 타입을 지원한다.

프로그먼트 타입	매칭(예)	뒤에 올 수 있는 것...
meta	어트리뷰트 본문 : <code>inline</code> , <code>derive(Copy, Clone)</code>	전부
literal	리터럴 값 : <code>1024</code> , <code>“Hello, world!”</code> , <code>1_000_000f64</code>	전부
lifetime	수명 : <code>‘a</code> , <code>‘item</code> , <code>‘static</code>	전부
vis	가시성 지정자 : <code>pub</code> , <code>pub(crate)</code>	전부
ident	식별자 : <code>std</code> , <code>Json</code> , <code>longish_variable_name</code>	전부
tt	토큰 트리 : <code>;</code> , <code>>=</code> , <code>{}</code> , <code>[0 1 (+ 0 1)]</code>	전부

- 표에 있는 대부분의 옵션은 Rust 문법을 엄격히 준수한다.
 - expr 타입은 Rust 표현식만 매칭한다. 그리고 ty 타입은 Rust 타입만 매칭한다.
 - 또한 확장할 수 없다. 예를 들어, expr을 인식하는 새 산술 연산자나 새 키워드를 정의할 방법이 없다.
- 마지막에 있는 ident와 tt는 Rust 코드처럼 생기지 않은 매크로 인수의 매칭을 지원한다.
 - ident는 임의의 식별자를 매칭한다.
 - tt는 단일 토큰 트리, 즉 짹이 맞는 한 쌍의 괄호 (...), [...], {...}와 중첩 토큰 트리를 포함한 그 사이에 오는 모든 것, 또는 1926이나 “Knots”처럼 괄호를 두르지 않는 단일 토큰을 매칭한다.
- Rust는 우리가 작성한 매크로를 깨뜨리는 일 없이 미래에 새 문법 기능을 추가하기 위해서 프로그먼트 바로 뒤에 있는 패턴에 올 수 있는 토큰을 제한하고 있다.
 - 예를 들어, 패턴 `$x:expr ~ $y:expr`은 `expr` 뒤에 `~`가 올 수 없으므로 오류가 발생한다.
 - 반면 패턴 `$vars:pat => $handler:expr`은 `$vars:pat` 뒤에 오는 화살표 `=>`가 `pat`에 허용되는 토큰이다.

프로그먼트 타입

Korea University MatKor 스터디
Week 7: Macros

- json! 매크로에 필요한 게 바로 토큰 트리다.
 - 모든 JSON 값은 단일 토큰 트리다. 수, 문자열, 불 값, null은 모두 단일 토큰이며, 객체와 배열은 꽉호를 두르고 있다.



```
macro_rules! json {
    (null) => {
        Json::Null
    };
    ([ $( $element:tt ),* ]) => {
        Json::Array( ... )
    };
    ({ $($key:tt : $value:tt ),* }) => {
        Json::Object( ... )
    };
    ($other:tt) => {
        ...          // TODO: Number, String, Boolean
    };
}
```

매크로의 재귀

Korea University MatKor 스터디
Week 7: Macros

- 매크로가 자기 자신을 호출하는 간단한 예는 이미 살펴본 바 있다.
 - 앞에서 구현한 `vec!`은 후행 쉼표를 지원하기 위해서 재귀를 쓴다.
 - 하지만 `json!`은 자기 자신을 재귀적으로 호출해야 한다.
- 다음처럼 재귀를 쓰지 않고도 JSON 배열을 지원할 수도 있지 않을까?

```
● ● ●  
  
([ $( $element:tt ),* ]) => {  
    Json::Array(vec![ $( $element ),* ])  
};
```

- 그러나 이 코드는 제대로 동작하지 않는다.
- 이렇게 하면 JSON 데이터(`$element` 토큰 트리)가 Rust 표현식 안에 그대로 들어간다. 이 둘은 서로 다른 언어다.

매크로의 재구

Korea University MatKor 스터디
Week 7: Macros

- 배열의 각 요소를 JSON 형식에서 Rust로 변환해야 한다.

다행히 그런 일을 하는 매크로가 있는데, 지금 작성하고 있는 매크로가 바로 그것이다!

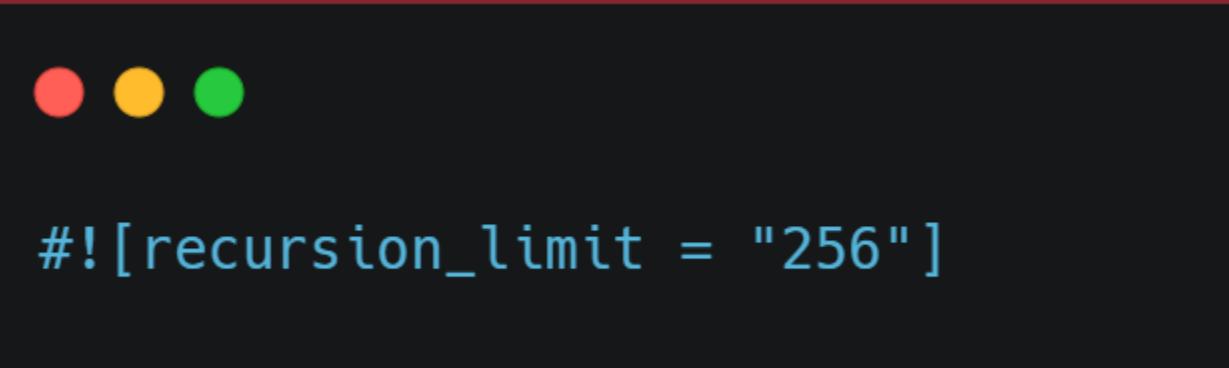
```
● ● ●  
([ $( $element:tt ),* ]) => {  
    Json::Array(vec![ $( json!($element) ),* ])  
};
```

- 오브젝트도 같은 식으로 지원할 수 있다.

```
● ● ●  
({ $($key: tt : $value:tt ),* }) => {  
    Json::Object(Box::new(vec![  
        $($key.to_string(), json!($value)),*  
    ].into_iter().collect()))  
};
```

- 참고 사항

- 컴파일러는 매크로의 재귀 호출 횟수를 기본 64회로 제한하고 있다.
복잡한 재귀를 쓰는 매크로의 경우 제한 횟수를 넘어가는 경우도 있다.
- 이때는 매크로를 사용하는 크레이트 맨 위에 다음 어트리뷰트를 추가해 횟수를 늘리면 된다.



```
#![recursion_limit = "256"]
```

매크로와 트레잇 함께 쓰기

Korea University MatKor 스터디
Week 7: Macros

- 복잡한 매크로를 작성하는 건 늘 퍼즐 맞추기처럼 어렵다.
 - json!(true), json!(1.0), json!("yes")를 지원해서 그 값이 무엇이든 간에 적절한 종류의 Json 값으로 변환해야 한다. 그러나 매크로는 타입을 잘 구분하지 못한다.
 - 예를 들어, 다음과 같은 식으로 코드를 작성했다고 하자.

```
● ● ●

macro_rules! json {
    (true) => {
        Json::Boolean(true)
    };
    (false) => {
        Json::Boolean(false)
    };
}
```

- 이런 접근 방식은 금방 한계가 드러난다.
불 값은 두 개뿐이라 어떻게 할 수 있을지 몰라도 수와 문자열을 이런 식으로 처리하는 건 불가능하다.

매크로와 트레잇 함께 쓰기

Korea University MatKor 스터디
Week 7: Macros

- 다행히도 다양한 타입의 값을 지정한 타입의 값으로 변환하는 표준적인 방법이 있다.
바로 From 트레잇이다. 이 트레잇을 몇 가지 타입을 대상으로 구현하기만 하면 된다.

```
● ● ●

impl From<bool> for Json {
    fn from(b: bool) -> Json {
        Json::Boolean(b)
    }
}

impl From<i32> for Json {
    fn from(i: i32) -> Json {
        Json::Number(i as f64)
    }
}

impl From<String> for Json {
    fn from(s: String) -> Json {
        Json::String(s)
    }
}

impl<'a> From<&'a str> for Json {
    fn from(s: &'a str) -> Json {
        Json::String(s.to_string())
    }
}
```

매크로와 트레잇 함께 쓰기

Korea University MatKor 스터디

Week 7: Macros

- 12가지 수치 탑입의 구현은 전부 비슷하므로 매크로를 작성하면 반복 작업을 줄일 수 있다.

매크로와 트레잇 함께 쓰기

Korea University MatKor 스터디
Week 7: Macros

- 이제 `Json::from(value)`를 써서 지원하는 모든 타입의 값을 Json으로 변환할 수 있다.
- 이렇게 하면 불 값, 수, 문자열을 처리할 수 있다.



```
( $other:tt ) => {
    Json::from($other)
};
```

매크로와 트레잇 함께 쓰기

Korea University MatKor 스터디
Week 7: Macros

- 완성된 매크로 코드는 다음과 같다.

```
● ● ●

macro_rules! json {
    (null) => {
        Json::Null
    };
    ([ $( element:tt ),* ]) => {
        Json::Array(vec![ $( json!($element) ),* ])
    };
    ({ $($key:tt : $value:tt ),* }) => {
        Json::Object(Box::new(vec![
            $( ($key.to_string(), json!($value)) ),*
        ].into_iter().collect())))
    };
    ( $other:tt ) => {
        Json::from($other)
    };
}
```

범위 한정과 하이닉 매크로

Korea University MatKor 스터디
Week 7: Macros

- Rust에서 매크로를 작성하는 게 너무나 까다로운 이유는 전개 과정에서 서로 다른 범위에 있는 코드를 한 곳에 붙여 넣기 때문이다.
- 예를 들어, JSON 오브젝트를 파싱하는 규칙을 재작성해서 임시 벡터를 없애 보자.

```
● ● ●

({ $($key:tt : $value:tt ),* }) => {
{
    let mut fields = Box::new(HashMap::new());
    $($fields.insert($key.to_string(), json!($value)); )*
    Json::Object(fields)
}
};
```

- 이제 HashMap을 채울 때 collect()을 쓰는 게 아니라 .insert() 메소드를 반복해서 호출한다. 이 말은 맵을 fields라고 하는 임시 변수에 저장해야 한다는 뜻이다.

범위 한정과 하이닉 매크로

Korea University MatKor 스터디
Week 7: Macros

- 그러나 `json!`을 호출하는 코드가 우연히 `fields`라는 이름의 변수를 쓴다면 어떨까?

```
● ● ●  
  
let fields = "Fields, W.C.";  
let role = json!({  
    "name": "Larson E. Whipsnade",  
    "actor": fields  
});
```

- 이 매크로를 전개하면 사용 중인 `fields`가 서로 다른 것을 가리키는 두 코드를 한 곳에 붙여 넣게 된다.

```
● ● ●  
  
let fields = "Fields, W.C.";  
let role = {  
    let mut fields = Box::new(HashMap::new());  
    fields.insert("name".to_string(), Json::from("Larson E. Whipsnade"));  
    fields.insert("actor".to_string(), Json::from(fields));  
    Json::Object(fields)  
}
```

- 코드 분석
 - 임시 변수를 쓰는 매크로라면 피할 수 없는 함정 같아 보일 수도 있겠고, 어떻게 하면 고칠 수 있을지 고민할 수도 있다.
 - 이럴 때는 아무래도 `json!` 매크로가 정의하는 변수의 이름을 호출부가 넘기지 않을 법한 이름으로 바꿔야 할 것이다.
이럴테면 `fields`를 `__json$fields`로 바꾸는 식이다.
 - 그런데 놀라운 점은 매크로를 바꾸지 않아도 문제없이 동작한다는 것이다. Rust가 알아서 변수의 이름을 바꿔준다!
 - 스킴 매크로가 처음 구현한 이 기능을 하이지닉(Hygiene)이라고 하므로 Rust에서는 이를 하이지닉 매크로라고 한다.

- 하이닉 매크로를 이해하는 가장 쉬운 방법은 매크로가 전개될 때마다 매크로 자체에서 전개되어 나온 부분이 다른 색으로 칠해진다고 생각하는 것이다.
- 색이 다른 변수는 마치 다른 이름을 가진 것처럼 처리된다.
- "name"과 "actor"처럼 매크로 호출부가 넘겨서 출력 안에 붙여넣기된 코드는 원래 색을 유지하는 반면, 매크로 템플릿에서 비롯된 토큰은 다른 색으로 칠해진다.

```
let fields = "Fields, W.C.";  
let role = {  
    let mut fields = Box::new(HashMap::new());  
    fields.insert("name".to_string(), Json::from("Larson E. Whipsnade"));  
    fields.insert("actor".to_string(), Json::from(fields));  
    Json::Object(fields)  
}
```

매크로 가져오기와 내보내기

Korea University MatKor 스터디
Week 7: Macros

- Rust에서는 컴파일 과정의 초기 단계에서 프로젝트의 전체 모듈 구조가 파악되기 전에 매크로 전개가 일어나므로 이를 가져오고 내보내기 위한 특별한 방식을 제공한다.
- 한 모듈에서 볼 수 있는 매크로는 자동으로 자식 모듈에서 볼 수 있다.
어떤 모듈에 있는 매크로를 부모 모듈쪽으로 내보내려면 `#[macro_use]` 애트리뷰트를 쓴다.
- 예를 들어, `lib.rs`가 다음과 같다고 하자.

```
● ● ●  
  
#[macro_use] mod macros;  
mod client;  
mod server;
```

- 이 코드는 `macros` 모듈에 정의된 매크로를 전부 `lib.rs`로 가져오기 때문에 `client`와 `server`를 포함한 크레이트 전역에서 볼 수 있다.

마크로 가져오기와 내보내기

Korea University MatKor 스터디
Week 7: Macros

- Rust에서는 컴파일 과정의 초기 단계에서 프로젝트의 전체 모듈 구조가 파악되기 전에 마크로 전개가 일어나므로 이를 가져오고 내보내기 위한 특별한 방식을 제공한다.
 - #[macro_export]가 붙은 마크로는 자동으로 pub가 되며 다른 아이템과 마찬가지로 경로를 써서 참조할 수 있다.
 - 예를 들어, lazy_static 크레이트는 #[macro_export]가 붙은 lazy_static이라고 하는 마크로를 제공한다. 이 마크로를 여러분의 크레이트에서 쓰려면 다음과 같이 사용하면 된다.

```
● ● ●  
use lazy_static::lazy_static;  
lazy_static!{ }
```

- 마크로를 가져오고 난 뒤에는 다른 아이템처럼 쓰면 된다.

```
● ● ●  
use lazy_static::lazy_static;  
  
mod m {  
    crate::lazy_static!{ }  
}
```

매크로 가져오기와 내보내기

Korea University MatKor 스터디
Week 7: Macros

- 물론, 실제로 이렇게 한다는 건 여러분의 매크로가 다른 모듈에서 호출될 수도 있다는 뜻이다.
따라서 내보낸 매크로는 범위 안에 있는 다른 무언가에 의존하면 안된다.
- 매크로가 쓰이는 범위 안에 무엇이 있을지 알 길이 없을 뿐만 아니라 표준 프렐류드의 기능도 숨길 수 있기 때문이다.
- 이 문제를 해결하기 위해서는 매크로에서 쓰는 모든 이름을 절대 경로로 써야 한다.
`macro_rules!`는 이를 돋기 위해 특별한 프래그먼트인 `$crate`를 제공한다.
- `$crate`는 매크로뿐만 아니라 아무 경로에나 쓸 수 있는 키워드인 `crate`와는 다르다.
매크로가 정의되어 있는 크레이트의 루트 모듈을 가리키는 절대 경로처럼 동작한다.
- `Json`이 아니라 `$crate::Json`이라고 쓰면 `Json`을 가져오지 않은 상태에서도 동작한다.
`HashMap`은 `::std::collections::HashMap`이나 `$crate::macros::HashMap`으로 바꿀 수 있다.
- 단, `$crate`는 크레이트의 비공개 기능을 접근하는데 쓸 수는 없으므로 후자의 경우에는 `HashMap`을 다시 내보내야 한다.
그러면 실제로 `::jsonlib`과 같은 평범한 경로로 전개되며, 가시성 규칙은 영향을 받지 않는다.

매크로 가져오기와 내보내기

Korea University MatKor 스터디
Week 7: Macros

- \$crate를 쓰도록 바꾼 json! 매크로의 최종 버전은 다음과 같다.

```
● ● ●

pub use std::collections::HashMap;
pub use std::boxed::Box;
pub use std::string::ToString;

#[macro_export]
macro_rules! json {
    (null) => {
        $crate::Json::Null
    };
    ([ $($element:tt ),* ]) => {
        $crate::Json::Array(vec![ $( json!($element) ),* ])
    };
    ({ $($key:tt : $value:tt ),* }) => {
        {
            let mut fields = $crate::macros::Box::new($crate::macros::HashMap::new());
            $($
                fields.insert($crate::macros::ToString::to_string($key), json!($value));
            )*
            $crate::Json::Object(fields)
        }
    };
    ($other:tt) => {
        $crate::Json::from($other)
    };
}
```

매칭 중에 발생하는 구문 오류 피하기

Korea University MatKor 스터디
Week 7: Macros

- 다음 매크로는 아무런 문제도 없어 보이지만 Rust에서 몇 가지 문제를 떠안긴다.

```
● ● ●

macro_rules! complain {
    ($msg : expr) => {
        println!("Complain filed: {}", $msg)
    };
    ($user : $userid:tt , $msg : expr) => {
        println!("Complaint from user: {}: {}", $userid, $msg)
    };
}
```

- 이 매크로를 다음처럼 호출한다고 하자.

```
● ● ●

complain!(user: "jimb", "the AI lab's chatbots keep picking on me");
```

- **코드 분석**
 - 언뜻 보기에도 이 코드는 당연히 두 번째 패턴과 매칭될 것처럼 보인다.
그러나 Rust에서 첫 번째 규칙인 `$msg:expr`과의 매칭을 시도할 때 일이 틀어지기 시작한다.
 - 물론 `user: "jimb"`은 표현식이 아니므로 구문 오류가 발생한다.
Rust는 어려운 매크로 디버깅의 짐을 덜어주기 위해서 문법 오류를 비밀로 하길 거부한다.
따라서 즉시 그 사실을 알리고 컴파일을 멈춘다.
 - 패턴에 있는 다른 토큰이 매칭에 실패하면 Rust가 다음 규칙으로 넘어가므로 문제될 게 없다.
치명적인 건 프래그먼트를 매칭하려고 할 때만 발생하는 구문 오류 뿐이다.
 - 문제를 이해하는 건 어렵지 않은데, 핵심은 바로 잘못된 규칙의 프래그먼트 `$msg:expr`과 매칭하려고 한다는 점이다.
이리로 빠지는 건 계획에 없던 일이므로 매칭되면 안 된다.
 - 참고 사항 : 이 오류는 Rust 1.70.0까지만 발생하며, Rust 1.71.0 부터는 정상적으로 컴파일된다.

매칭 중에 발생하는 구문 오류 피하기

Korea University MatKor 스터디
Week 7: Macros

- 해결 방법

- 이 문제를 쉽게 피해가는 방법은 두 가지다.
- 첫 번째 방법은 혼동하기 쉬운 규칙을 바꾸는 것이다.

예를 들어, 매크로에 있는 모든 패턴을 서로 다른 식별자로 시작하게 바꿀 수 있다.



```
macro_rules! complain {
    ($msg : $msg:expr) => {
        println!("Complain filed: {}", $msg)
    };
    ($user : $userid:tt, $msg : $msg:expr) => {
        println!("Complaint from user: {}: {}", $userid, $msg)
    };
}
```

- 매크로 인수가 msg로 시작하면 첫 번째 규칙을 타고 user로 시작하면 두 번째 규칙을 탄다.
어느 쪽이든 프로그먼트를 매칭해 보기 전에 올바른 규칙을 타게 된다는 걸 알 수 있다.

매칭 중에 발생하는 구문 오류 피하기

Korea University MatKor 스터디
Week 7: Macros

- 해결 방법

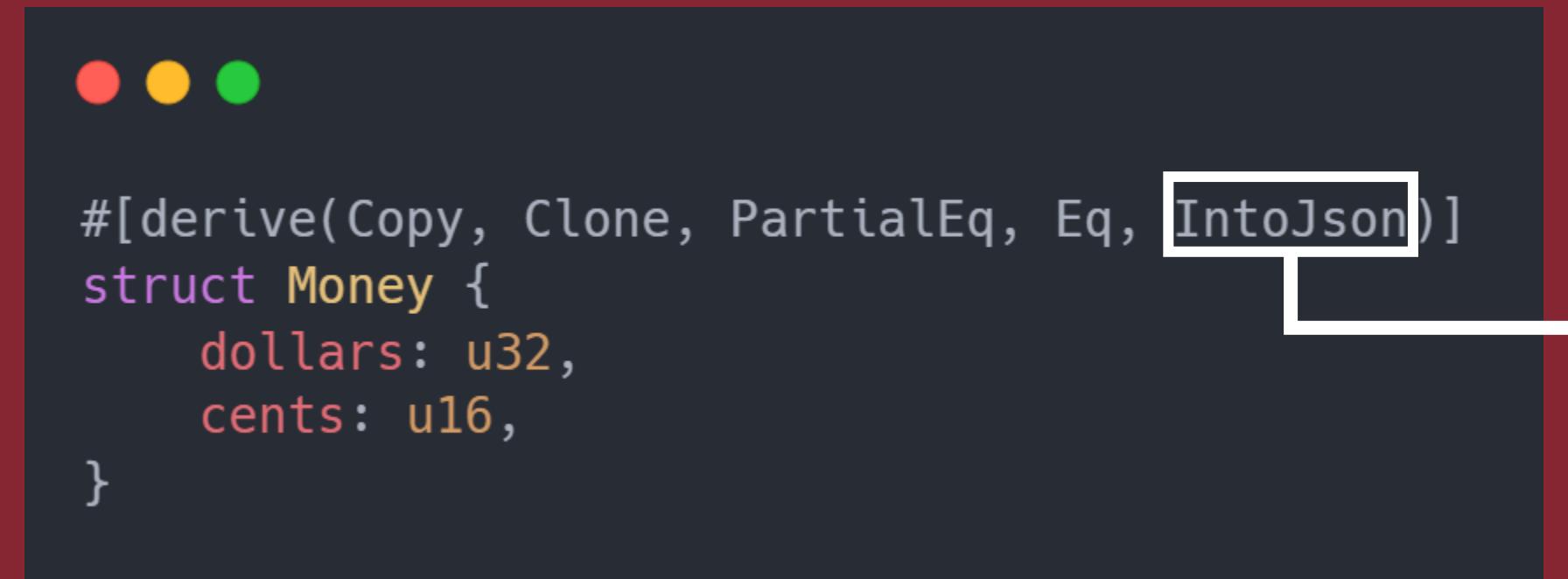
- 이 문제를 쉽게 피해가는 방법은 두 가지다.
- 두 번째 방법은 보다 구체적인 규칙을 먼저 두는 것이다.

user : 규칙을 먼저 두면 구문 오류를 일으키는 규칙에 도달할 일이 없으므로 complain! 문제가 해결된다.

```
● ● ●

macro_rules! complain {
    (user : $userid:tt , $msg : expr) => {
        println!("Complaint from user: {}:{}: {}", $userid, $msg)
    };
    ($msg : expr) => {
        println!("Complain filed: {}", $msg)
    };
}
```

- 매크로 패턴은 훨씬 복잡한 입력도 파싱할 수 있지만, 복잡성이 금방 겉잡을 수 없이 퍼진다.
 - 대니얼 킵(Danial Keep)이 쓴 "The Little Book of Rust Macros"는 고급 macro_rules! 프로그래밍을 다루고 있다. 매크로 전개와 관련해 여기서 다룬 것보다 더 자세히 설명하며, 복잡한 입력을 파싱하는 영리한 기법 몇 가지를 소개한다.
- Rust 1.15에는 절차적 매크로(Procedural Macro)라고 하는 별도의 메커니즘이 도입됐다.
 - 절차적 매크로를 쓰면 #[derive] 애트리뷰트를 확장해서 자동으로 생성되는 사용자 정의 구현을 처리할 수 있을 뿐만 아니라 사용자 정의 애트리뷰트와 앞서 이야기한 macro_rules! 매크로처럼 호출되는 새 매크로를 만들 수 있다.



A screenshot of a terminal window showing Rust code. The code defines a struct Money with fields dollars and cents, and applies the derive macro to add Copy, Clone, PartialEq, Eq, and IntoJson traits. A red arrow points from the IntoJson trait name to a callout box containing the text "자동으로 생성되는 사용자 정의 구현".

```
#[derive(Copy, Clone, PartialEq, Eq, IntoJson)]
struct Money {
    dollars: u32,
    cents: u16,
}
```

자동으로 생성되는 사용자 정의 구현

- 절차적 매크로
 - “절차적”인 이유는 매크로가 일련의 선언적 규칙이 아니라 Rust 함수 형태로 구현되기 때문이다. 이 함수는 얇은 추상층을 통해서 컴파일러와 소통하며, 임의의 복잡도를 가질 수 있다.
 - 예를 들어 diesel 데이터베이스 라이브러리는 절차적 매크로를 써서 컴파일 타임에 데이터베이스에 연결하고, 그 데이터베이스의 스키마를 토대로 코드를 생성한다.
 - 컴파일러 내부와 소통하기 때문에 효과적인 매크로를 작성하려면 컴파일러의 동작 방식을 이해해야 한다. 자세한 내용은 <https://doc.rust-lang.org/book/ch19-06-macros.html#procedural-macros-for-generating-code-from-attributes> 또는 <https://doc.rust-lang.org/reference/procedural-macros.html> 참고

- <https://www.rust-lang.org/>
- <https://doc.rust-lang.org/book/>
- <https://tourofrust.com/>
- The Rust Programming Language (No Starch Press, 2019)
- Programming Rust, 2nd Edition (O'Reilly, 2021)

Thank you!