Rust patterns

BORN TO FRAG

github.com/Kixunil

Newtype

```rust
struct Height(f64);

impl Height {
    const fn from_meters(meters: f64) -> Self {
        Height(f64)
    }

    const fn from_feet(feet: f64) -> Self {
        Self::from_meters(feet * 0.3048)
    }
}
```

```
1 fn get_stuff(value1: u32, value2: u32) -> u64 {
2     let x = external_library::foo(value1);
3     external_library::bar(x, value2)
4 }
```

```rust
fn get_stuff(value1: u32, value2: u32) -> Option<u64> {
    let x = external_library::foo(value1)?;
    Some(external_library::bar(x, value2))
}
```

```
fn compute_average(items: &[usize]) -> usize {
    items.iter().copied().sum::<usize>() / items.len()
}
```

```rust
fn compute_average(items: &[usize]) -> Option<usize> {
    items.iter().copied().sum::<usize>().checked_div(items.len())
}
```

```rust
fn compute_average(items: &[usize]) -> Result<usize, EmptySliceError> {
    items.iter().copied().sum::<usize>().checked_div(items.len()).ok_or(EmptySliceError)
}

#[derive(Debug, Clone)]
struct EmptySliceError;

// impl Display, Error
```

```rust
fn compute_average(items: NonEmptySlice<'_>) -> usize {
    items.0.iter().copied().sum::<usize>() / items.0.len()
}

struct NonEmptySlice<'a>(&'a [usize]);

impl<'a> TryFrom<&'a [usize]> for NonEmptySlice<'a> {
    type Error = EmptySliceError;
    fn try_from(slice: &'a [usize]) -> Result<Self, Self::Error> {
        if slice.is_empty() {
            Err(EmptySliceError)
        } else {
            Ok(Self(slice))
        }
    }
}

#[derive(Debug, Clone)]
struct EmptySliceError;

// impl Display, Error
```

```rust
fn compute_average(items: &NonEmptySlice) -> usize {
    items.0.iter().copied().sum::<usize>() / items.0.len()
}

#[repr(transparent)]
struct NonEmptySlice([usize]);

impl<'a> TryFrom<&'a [usize]> for &'a NonEmptySlice {
    type Error = EmptySliceError;
    fn try_from(slice: &'a [usize]) -> Result<Self, Self::Error> {
        if slice.is_empty() {
            Err(EmptySliceError)
        } else {
            Ok(unsafe { &*(slice as *const _ as *const NonEmptySlice) })
        }
    }
}

#[derive(Debug, Clone)]
struct EmptySliceError;

// impl Display, Error
```

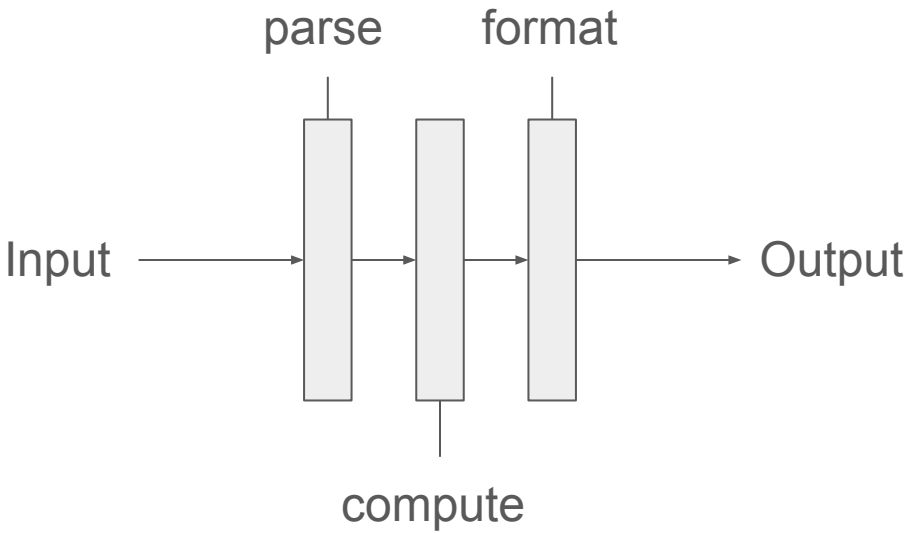
```rust
impl<'a> From<&'a [usize; 1]> for &'a NonEmptySlice {
    fn from(value: &'a [usize; 1]) -> Self {
        unsafe { &*(value as &[_] as *const _ as *const NonEmptySlice) }
    }
}
```

```rust
    pub const fn new(slice: &[usize]) -> Result<&Self, EmptySliceError> {
        if slice.is_empty() {
            Err(EmptySliceError)
        } else {
            Ok(unsafe { &*(slice as *const _ as *const NonEmptySlice) })
        }
    }
```

```
17  const FOO: &NonEmptySlice = match NonEmptySlice::new(&[42]) { Ok(val) => val, Err(_) => panic!("empty"), };
```

```rust
use non_empty_slice::NonEmptySlice;

mod non_empty_slice {
    #[repr(transparent)]
    pub struct NonEmptySlice([usize]);

    impl NonEmptySlice {
        pub fn len(&self) -> usize {
            self.0.len()
        }

        pub fn iter(&self) -> core::slice::Iter<'_, usize> {
            self.0.iter()
        }
    }

    impl<'a> TryFrom<&'a [usize]> for &'a NonEmptySlice {
```

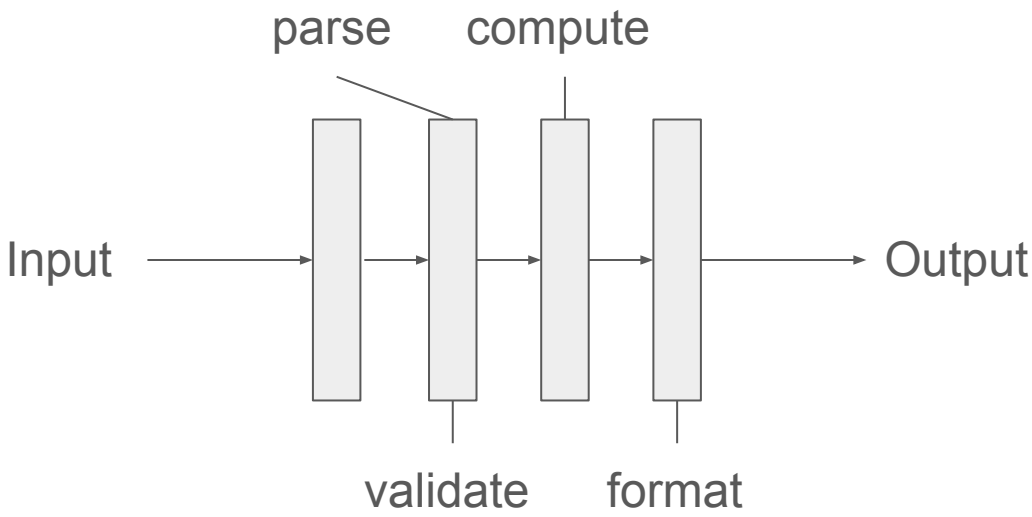

Input
Output


parse
format
Input
Output
compute

```rust
use serde; // 1.0.197
use std::fmt;

#[derive(serde::Deserialize)]
struct Input {
    items: NonEmptyVec,
}

#[derive(serde::Deserialize)]
#[serde(try_from = "Vec<usize>")]
struct NonEmptyVec(Vec<usize>);

impl TryFrom<Vec<usize>> for NonEmptyVec {
    type Error = EmptySliceError;
    fn try_from(vec: Vec<usize>) -> Result<Self, Self::Error> {
        if vec.is_empty() {
            Err(EmptySliceError)
        } else {
            Ok(Self(vec))
        }
    }
}
```

```rust
use serde; // 1.0.197
use toml; // 0.8.10
use std::fmt;

#[derive(serde::Deserialize)]
struct Input {
    items: toml::Spanned<NonEmptyVec>,
}
```

```rust
use serde; // 1.0.197
use toml; // 0.8.10
use std::fmt;

#[derive(serde::Deserialize)]
struct RawInput {
    items: toml::Spanned<Vec<usize>>,
}

struct Input {
    items: toml::Spanned<NonEmptyVec>,
}
```


parse
compute
Input
Output
validate
format

https://github.com/Kixunil/debcrafter

```rust
pub struct Address<Checked: Validation> {
    inner: AddressInner,
    _phantom: std::marker::PhantomData<Checked>,
}

enum AddressChecked {}
enum AddressUnchecked {}

pub trait Validation: sealed::Validation {}

mod sealed {
    pub trait Validation {}
}

// impl Validation for AddressChecked & AddressUnchecked
```

```rust
impl Address<AddressUnchecked> {
    pub fn require_network(self, network: Network) -> Result<Address<AddressUnchecked>, Error> {
        /* ... */
    }
}

impl Address<AddressChecked> {
    pub fn use_address(&self) {
        /* ... */
    }
}
```

```
1 struct Foo {
2     bar: String, /* &str? */
3 }
```

```
1 struct Foo<T: Borrow<str>> {
2     bar: T,
3 }
```

```
1  struct Foo<T: Borrow<Bar>> {
2      bar: T,
3  }
4
5  struct Bar { data: [u8; 1024] }
```

```rust
struct Foo<T: AsRef<Bar>> {
    bar: T,
}

struct Bar { data: [u8; 1024] }

impl AsRef<Bar> for Bar {
    fn as_ref(&self) -> &Bar {
        self
    }
}
```

```rust
struct Foo<T: SaneRef<Bar>> {
    bar: T,
}

struct Bar { data: [MaybeUninit<u8>; 1024] }

/// Safety: the returned reference must be the same every call until the object
/// is modified.
unsafe trait SaneRef<T>: AsRef<T> {}

impl AsRef<Bar> for Bar {
    fn as_ref(&self) -> &Bar {
        self
    }
}
```