Ancient Rust

Florian “Florob” Zeitz

2025-06-04

All the Sigils

```
#[deriving(Eq)]
/// If an error occurs while parsing some XML,
/// this is the structure which is returned
pub struct Error {
    /// The line at which the error occurred
    line: uint,
    /// The column at which the error occurred
    col: uint,
    /// A message describing the type of the error
    msg: @~str
}
```

~T owned box (Box<T> today)

@T managed box,
"heap allocation with the lifetime managed by a task-local garbage collector[1]" (removed from the language)

Special cases (break composition, no DSTs yet):

~[T] growable vector (Vec<T> today)

~str growable string (String today)

[1] Up until its removal was actually implemented as reference counting

do Expressions

```
fn each(v: &[int], op: &fn(v: &int)) {
    let mut n = 0;
    while n < v.len() {
        op(&v[n]);
        n += 1;
    }
}

each([1, 2, 3], |n| {
    do_some_work(n);
});

do each([1, 2, 3]) |n| {
    do_some_work(n);
}
```

```
do thread::spawn() || {
    each([1, 2, 3], |n| {
        do_some_work(n);
    });
}

// Can omit empty argument lists
do thread::spawn {
    do each([1, 2, 3]) |n| {
        do_some_work(n);
    }
}
```

- sugar for passing closures to functions
- only works for the last argument

`for` Loops (Interior vs. Exterior Iteration)

```
fn each(v: &[int], op: &fn(v: &int) → bool) {
    let mut n = 0;
    while n < v.len() {
        if !op(&v[n]) {
            break;
        }
        n += 1;
    }
}

for each([2, 4, 8, 5, 16]) |n| {
    if *n % 2 ≠ 0 {
        println("found odd number!");
        break;
    }
}

fn contains(v: &[int], elt: int) → bool {
    for each(v) |x| {
        if (*x == elt) { return true; }
    }
    false
}
```

```
for vec.iter().advance |elem| {
  println(
      fmt!("%?", elem)
  );
}
```

`break`
: returns `false` from the closure

`loop` (later `continue`)
: returns `true` from the closure

`return`
: returns from enclosing scope

- switched to today's exterior iteration in 0.8

Conditions

```
condition! {
    pub unrecognized_entity: (~str) → ~str;
}

/// Unescapes all valid XML entities in a string.
pub fn unescape(input: &str) → ~str {
    ...
    result.push_str(
        unrecognized_entity::cond.raise(entity)
    );
    ...
}

do unrecognized_entity::cond.trap(|ent| {
    if ent.as_slice() == " " {
        ~"\u00a0"
    } else {
        ent
    }
}).inside {
    let unesc = unescape(" &foo;");
    assert_eq!(unesc, ~"\u00a0&foo;");
}
```

- “less blunt then panic”
- “less cumbersome then `Result`”
- callers can register handlers
- innermost handler is called
- panics if no handler exists
- avoids having to thread callbacks through function calls

Green Threads

```
do task::spawn {
    do_some_work();
}
```

- up until 1.0.0-alpha Rust had a runtime
- could spawn green threads onto the runtime
- each task has its own stack
- used segmented stacks, start small, grow when required
  - runtime and complexity overhead
- blocking a task can block the runtime/worker thread

Questions?

https://babelmonkeys.de/~florob/talks/RC-2025-06-04-ancient-rust.pdf

Thank you for your attention. Any questions?