Class 2: Getting Started with Rust

Action Items

- If you think you are in the class but did not already submit a cs4414 student survey, you need to meet with me this week. I will assume only students who submitted the survey are actually in the class, and only those students will be assigned teammates for Problem Set 2.
- You should have at least completed the Exercises for Problem Set 1 before Thursday's class. Problem Set 1 is due on Tuesday, September 10.

Programming Languages

Why are there so many programming languages?

- Languages change the way we **think**.
- Languages provide **abstractions** for machine resources. The abstractions they provide are engineering tradeoffs between:
  - *expressiveness* and *"truthiness"*
  - *freedom* and *safety*
  - *flexibility* and *simplicity*
  - *efficiency* and *ease-of-use*

Which of these should we prefer for a programming language for systems programming?

What's the difference between a language and an operating system?

Rust

Rust is a systems programming language developed by Mozilla Research. It is a new and immature language: the release we are using is Version 0.7 (released in July 2013).

Rust is designed with a particular focus on providing *safety* and *efficiency*. This means it provides programmers with a lot of control over how memory is managed, but without the opportunity to shoot yourself in the foot so readily provided by C/C++. Much of the design is driven by the needs of Servo, an experimental, highly-parallelizable web browser engine.

Today, we will introduce some of the basics you'll need to get started programming in Rust (and for completing PS1). In later classes, we'll look more deeply as some of the most interesting aspects of how types, memory management, and concurrency work in Rust.

First Rust Program

Let's get started with a simple Rust program:

```
fn max(a: int, b: int) -> int {
    if a > b {
        a
    } else {
        b
    }
}

fn main() {
    println(fmt!("Max: %?", max(3, 4)));
}
```

If you are familiar with C (or at least C++) and Java, most of this should look fairly familiar, but some things a bit unusual.

Statements and Expressions

Hypothesized Rust grammar (the Rust manual has a grammar, but "looking for consistency in the manual's grammar is bad: it's entirely wrong in many places"):

IfExpression ::= **if** *Expression Block* [**else** *Block*]

Block ::= { [*Statement* Expr*] }

Expression ::= *Block*

How is the meaning of ; different in Rust and Java?

Higher-Order Functions

We can make new functions directly:

> *LambdaExpression* ::= **|** *Parameters* **|** *Block*

Define a function, `make_adder(int) -> ~fn(int) -> int`, that takes an integer as input and returns a function that takes and int and returns the sum of the original and input int.

```
fn ntimes(f: extern fn(int) -> int, times: int) -> ~fn(int) -> int {



}
```

```
fn double(a: int) -> int {
    a * 2
}

fn main() {
    let quadruple = ntimes(double, 2);
    println(fmt!("quad: %?", quadruple(2))); // 8
}
```

Reading a File

We will use the `std:io::file_reader` function to read a file (as you'll note if you follow the link, the current Rust documentation is very sparse and incomplete!):

```
    fn file_reader(path: &Path) -> Result<@Reader, ~str>
```

This function takes as `Path` as input and returns as `Result<@Reader, ~str>`.

Result is an enumerated type with two type parameters. Its value is either:

- `Ok(T)` - A successful value of type `T`
- `Err(U)` - An error of type `U`

Result types provide a way of dealing with errors without all the uncertainty and complexity of exceptions. So, in this case, `Result<@Reader, ~str>` is either a `@Reader`, corresponding to successfully opening the file, or an `~str`, corresponding to an error message when the file cannot be opened. (The `@` and `~` indicate different types of pointers; we'll get to that later.)

Why are (Java-style) exceptions a bad thing in a language focused on safety?

We can use pattern matching to control execution based on the type of the `Enum`:

```
fn load_file(pathname : ~str) -> ~[~str] {
    let filereader : Result<@Reader, ~str> = io::file_reader(~path::Path(pathname));
    match filereader {
        Ok(reader) => reader.read_lines(),
        Err(msg) => fail!("Cannot open file: " + msg),
    }
}
```

Note that `match` is an expression just like `if`! There's no need to `return reader.read_lines()`, its the value of the expression that is the body of `load_file`. (Also, note that this isn't a smart way to do this for large files, since it requires storing the whole file in memory.)

Match expressions must be *complete*:

```
 match filereader {
        Ok(reader) => reader.read_lines(),
 }
```

produces a compile-time error.

Match predicates can be arbitrary expressions:

```
fn collatz(n : int) -> int {
    let mut count = 0;
    let mut current = n;

    while current != 1 {
        current =
            match current {
               current if current % 2 == 0 => current / 2,
               _ => 3 * current + 1
            };
        count += 1
```

```
    }
    count
}

fn main() {
    for std::int::range(1, 100) |i| {
        println(fmt!("hailstone %d = %d", i, collatz(i)));
    }
}
```

Ways to Get Help

1. Search engines are your friend: reading files in rust Hints: unfortunately "rust" is a common word, and not (yet) a widely used language. If you know a specific function, easy to search on that (but harder if you don't know).
2. Stackoverflow can be great... but not much Rust discussion yet.
3. **Experiment!** The compiler is very helpful, and mostly provides good error and warning messages. Use `fmt!("%?", x)` to print out (almost) anything.
4. Ask for help:

- Piazza course forum
- IRC - most of the Rust developers hang out there and are quick to answer questions

If you figure something useful out that is not well documented, **document it**! Help out the class, the Rust community, and gain fame and fortune for yourself by creating good documentation and posting it in useful ways (course Piazza forum, your blog, reddit, etc.).

zhttpto.rs

```
extern mod extra;

use extra::{uv, net_ip, net_tcp};
use std::str;

static BACKLOG: uint = 5;
static PORT:    uint = 4414;
static IPV4_LOOPBACK: &'static str = "127.0.0.1";

fn new_connection_callback(new_conn :net_tcp::TcpNewConnection,
                           _killch: std::comm::SharedChan<Option<extra::net_tcp::TcpErrData>>)
{
    do spawn {
        let accept_result = extra::net_tcp::accept(new_conn);
        match accept_result {
```

```
            Err(err) => {
               println(fmt!("Connection error: %?", err));
            },
            Ok(sock) => {
                let peer_addr: ~str = net_ip::format_addr(&sock.get_peer_addr());
                println(fmt!("Received connection from: %s", peer_addr));

                let read_result = net_tcp::read(&sock, 0u);
                match read_result {
                    Err(err) => {
                        println(fmt!("Receive error: %?", err));
                    },
                    Ok(bytes) => {
                        let request_str = str::from_bytes(bytes.slice(0, bytes.len() - 1));
                        println(fmt!("Request received:\n%s", request_str));
                        let response: ~str = ~"HTTP/1.1 ..."; // full response removed
                        net_tcp::write(&sock, response.as_bytes_with_null_consume());
                    },
                };
            }
        }
    };
}

fn main() {
    net_tcp::listen(net_ip::v4::parse_addr(IPV4_LOOPBACK), PORT, BACKLOG,
                    &uv::global_loop::get(),
                    |_chan| { println(fmt!("Listening on tcp port %u ...", PORT)); },
                    new_connection_callback);
}
```

Links

Tony Hoare's talk, *Null References: The Billion Dollar Mistake*