

# Rusty Runtimes

Aditya Siram

September 17, 2016

# Outline

- Rust is a systems programming language
- Less used for language development
- Explores Rust for language implementation
- And as a compilation target!

# About Me

- ML/Lisp background
- Long time user, first time implementor
- Very little low level knowledge
- Mostly managed runtimes

- Lot of ML influence
  - Pattern matching
  - Emphasis in immutability
- Easy to learn.
- Mature metaprogramming.
- But still mostly imperative

- Lisp-ish

```
(defun adder (X) (+ 1 X))
```

- Scheme-ish

```
(defun length (list accum)  
  (cond ((= () list) accum)  
        (true (length (tl list) (+ accum 1)))))
```

- TCO'ed

- Curried!

```
(let F (map (lambda X (+ 1 X)))  
      (F (cons 1 (cons 2 (cons 3 ())))))
```

- Tiny
- Has a spec!

- Base types

```
#[derive(Debug, Clone)]
pub enum KlToken {
    Symbol(String),
    Number(KlNumber),
    String(String),
    Cons(Vec<KlToken>),
    Recur(Vec<KlToken>)
}
```

- Numbers

```
#[derive(Debug, Clone)]
pub enum KlNumber {
    Float(f64),
    Int(i64),
}
```



- Nom.
- Very macro heavy!

# Parsing a string

- Top level

```
named!(klstring<KlToken>,  
  chain!(  
    char!('\\"') ~  
    contents: many0!(klstringinnards) ~  
    char!('\\"'),  
    || KlToken::String(make_quoted_string(contents))  
  )  
);
```

- Innards

```
named!(klstringinnards< &[u8] >,  
  escaped!(none_of!("\\"), '\\', one_of!("\n\\"))  
);
```

- Rust macro debugging is nice!

```
named!(klstring<KlToken>,  
  chain!(  
    ...  
    ...  
    char!("hello"),  
    || ...  
  ));
```

- Error

```
--> src/main.rs:442:1  
  |  
442 | named!(klstring<KlToken>,  
  | ^  
  |
```

# Parsing Symbols

- Parsing symbols

```
named!(klsymbol<KlToken>,  
  chain!(  
    initial: one_of!(CHARACTERS) ~  
    remainder: many0!(  
      alt_complete!(  
        one_of!(DIGITS) |  
        one_of!(CHARACTERS)  
      )  
    ),  
  || {  
    let mut res : Vec <char> = vec![initial];  
    res.extend(remainder);  
    KlToken::Symbol(res.into_iter().collect())  
  })  
);
```

# Writing A Macro

- s-expression

```
(func b 1 2 3)
```

- Parser

```
named!(klsexp<KlToken>,  
  chain!(  
    char!('(') ~  
    inner: many0_until!(char!(')'), klsexpinnards) ~  
    char!(')'),  
    || {  
      KlToken::Cons(inner)  
    }  
  )  
);
```

# Writing A Macro

```
macro_rules! many0_until (
  ($input:expr, $stopmac:ident!( $($args:tt)* ), $submac:ident!(
    {
      let mut res = Vec::new();
      let mut input = $input;
      let mut loop_result = Ok(());

      while input.input_len() != 0 {
        match $stopmac!(input, $($args)*) {
          IResult::Error(_) => {
            match $submac!(input, $($args2)*) {
              IResult::Error(_) => {
                break;
              },
              IResult::Incomplete(Needed::Unknown) => {
                ...
              }
            }
          }
        }
      }
    }
  )
)
```

# KLambda Types

```
#[derive(Clone, Debug)]
pub enum KElement {
    Symbol(String),
    Number(KNumber),
    String(String),
    Cons(Vec<Rc<KElement>>),
    Closure(KClosure),
    Vector(Rc<UniqueVector>),
    Stream(Rc<KStream>),
    Nil,
    Recur(Vec<Rc<KElement>>)
}
```

```
#[derive(Clone)]
pub enum K1Closure {
    FeedMe(Rc<Fn(Rc<K1Element>) -> K1Closure>),
    Thunk(Rc<Fn() -> Rc<K1Element>>),
    Done(Result<Option<Rc<K1Element>>, Rc<K1Error>>),
    Trampoline(Rc<Fn() -> Rc<K1Element>>)
}
```



# Example

- Turning a string into a symbol

```
pub fn intern() -> K1Closure {
  FeedMe(
    Rc::new(
      | string | {
        match &*string {
          &K1Element::String(ref s) => {
            Done(Ok(Some(Rc::new(Symbol(s.clone())))))
          },
          _ => Done(shen_make_error("..."))
        }
      })
    )
  )
}
```

# Example

- Pos

```
pub fn pos() -> K1Closure {  
  FeedMe(  
    Rc::new(| string | {  
      FeedMe(  
        Rc::new(move | number | {  
          let string = string.clone();  
          match &*string {  
            &K1Element::String(ref s) => {  
              ...  
            },  
            ...  
          },  
          _ => ...  
        })  
      })  
    })  
  )  
}
```

# Example

- And

```
pub fn and () -> K1Closure {  
    ...  
    | a_thunk | {  
        ...  
        move | b_thunk | {  
            let forced = shen_force_thunk(a_thunk.clone())  
            match &*forced {  
                ...  
                _ => {  
                    let forced = shen_force_thunk(b_thunk)  
                    match &*forced {  
                        ...  
                        _ => true  
                    }  
                }  
            }  
        }  
    }  
    ...  
}
```

# Stored in a Function Table

- Global mutable function table

```
thread_local!(  
    static FUNCTION_TABLE: RefCell<HashMap<String, Klclosure>>  
        RefCell::new(HashMap::new())  
)
```

- Bootstrapping

```
pub fn fill_function_table() {  
    FUNCTION_TABLE.with(| function_table | {  
        let mut map = function_table.borrow_mut();  
        map.insert("pos" , pos());  
        map.insert("and" , and());  
        ...  
    })  
}
```

```
pub fn lookup_function(s: &String) -> Option<KlClosure> {  
    FUNCTION_TABLE.with(|table|{  
        let table = table.borrow();  
        let function = table.get(s);  
        match function {  
            Some(f) => Some((*f).clone()),  
            None => None  
        }  
    })  
}
```

- Function calls

```
(cons 1 ())
```

- Rust output

```
match function_apply(String::from("cons"),
                      vec![Rc::new(Number(Int(1))),
                          Rc::new(Cons(vec! []))])
{
    Ok(c) => c.clone(),
    Err(s) => Done(shen_make_error(s.clone().as_str()))
}
```

- Lets

```
(let X 1 (+ X X))  
((lambda X (+ X X)) 1)
```

- Lambda

```
match apply_lambda(  
    FeedMe(  
        Rc::new(move |X| {  
            let X_Copy = (*X).clone();  
            match function_apply(  
                String::from("+"), vec![  
                    Rc::new(X_Copy.clone()),  
                    Rc::new(X_Copy.clone())  
                ])  
            {  
                Ok(c) => ...,  
                Err(s) => ..  
            }  
        }  
    )))  
    Rc::new(KlElement::Number(KlNumber::Int(1)))
```



- Lambda

```
match apply_lambda(  
    ...  
        (move |X| {  
            let X_Copy = (*X).clone();  
            function_apply(  
                String::from("+"), vec![  
                    (X_Copy.clone()),  
                    (X_Copy.clone())  
                ]  
            )  
        {  
            Ok(c) => ...,  
            Err(s) => ..  
        }  
    )))
```

K1Number::Int(1)

```
(let X 2 (let Y (* X X) X))  
((lambda X ((lambda Y X) (* X X))) 2)
```

# Code Generation

```
match lambda_apply(  
  FeedMe(Rc...(move |X| {  
    let X_Copy = (*X).clone();  
    match lambda_apply(  
      FeedMe(Rc::new(move |Y| {  
        let X = X.clone();  
        let X_Copy = (*X).clone();  
        let Y_Copy = (*Y).clone();  
        K1Closure::Done(Ok(Some(Y_Copy.clone())))  
      })),  
    match function_apply(String::from("+"), vec![  
      Rc::new(X_Copy.clone()),  
      Rc::new(X_Copy.clone())])  
    {  
      ...  
    })),  
  Pattern(K1Element::Number(K1Number::Int(1)))
```

# Code Generation

```
match lambda_apply(  
    (move |X| {  
        let X_Copy = (*X).clone();  
        match lambda_apply(  
            (move |Y| {  
                let X = X.clone();  
                let X_Copy = (*X).clone();  
                let Y_Copy = (*Y).clone();  
  
                match function_apply(  
                    ("+"), vec![  
                        (X_Copy.clone()),  
                        (X_Copy.clone())])  
                {  
                    ...  
                })  
            })  
        )  
    })
```