Languages in Racket

Matthew Flatt

University of Utah
Why Language Extensibility?

Your programming language isn't good enough, yet

684 pages
[insert demo here]

example languages
Different levels of language extension...

• Syntactic abstraction

```
(define (roman->number str)
  (rx-case str
    ["([XV]*)\(\{1,4\}\)" xv i] (+ (roman->number xv)
      (string-length i))]
    ....))
```

• New language constructs

```
(class object% (define/public method ....) ....)
```

• New languages

```
(: factorial (Number -> Number))

@section{Hello}

int f(int n) { return n+1; }
```

... in one framework
Implementing a Text Adventure Game

You're standing in a field.  
There is a house to the north.  
> north  
You are standing in front of a house.  
There is a door here.  
> open door  
The door is locked.  
>
[insert demo here]

*play the game*
Implementing a Text Adventure Game

- **Places**
- **Things**
- **Verbs**
  - global intransitive verbs
  - place-local intransitive verbs
  - thing-specific transitive verbs
Implementing a Text Adventure Game

• **Places**

• **Things**

• **Verbs**
  - global intransitive verbs
  - place-local verbs
  - thing-specific verbs

Objects?

Need not only serialize, but save & restore variables

Methods?

Must convert between string command and method call

9-12
Domain-Specific Programming

The **programming language** approach:

- Provide expressive constructs
Domain-Specific Programming

The **Lisp** approach:

- Provide expressive constructs
- Enable syntactic abstraction
Domain-Specific Programming

The **Scheme** approach:

- Provide expressive constructs
- Enable syntactic abstraction
- Make syntactic abstraction easy
Domain-Specific Programming

The **Racket** approach:

- Provide expressive constructs
- Enable syntactic abstraction
- Make syntactic abstraction easy
- Smooth the path from syntactic abstraction to language construction
Implementing a Text Adventure Game

(define-verbs all-verbs
    [north (n) "go north"]
    [get _ (grab take) "take"]
    ....)

(define-actions everywhere-actions
    ([quit (printf "Bye!\n") (exit)]
     [look (show-current-place)]
     ....))

(define-thing cactus
    [get "Ouch!"])

(define-place desert
"You're in a desert. There is nothing for miles around."
(cactus key)
([north start]
  [south desert] ....))

....
Implementing a Text Adventure Game

===VERBS===
north, n
"go north"
....
===EVERYWHERE===
quit
(begin (printf "Bye!\n") (exit))
....
===THINGS===
---cactus---
get
"Ouch!"
....
===PLACES===
---desert---
"You're in a desert. There is nothing for miles around."
[cactus, key]
north start
south desert
....
Preprocessors vs. Macros

```java
import java.io.*
house = ...

game.maze ➤ mazec ➤ javac
```
Preprocessors vs. Macros

```java
import
java.io.*
house.*
game.java
```

```java
house = ...
house.maze
```

```java
javac
mazec
javac
```
Preprocessors vs. Macros

- **game.java**
  - `import java.io.*
house.*`
  - `javac`

- **house.maze**
  - `house = ...
  ...
  mazec`
  - `mazec`
  - `javac`

- **maze.java**
  - `javac`
  - `java`
Preprocessors vs. Macros

```
(require "io.rkt")
(define-syntax define-place ...
(define-place house .....)
```

`raco make`

`game.rkt`
Preprocessors vs. Macros

```
(require "io.rkt"
  "maze.rkt")
(define-place house ....)
```

`raco make`

game.rkt

```
(define-syntax define-place
  ...
)
```

`raco make`

maze.rkt
Preprocessors vs. Macros

```
(game.rkt)
(require "io.rkt"
  "house.rkt")
(raco make)

(house.rkt)
(require "maze.rkt")
(define-place house ....)
(raco make)

(maze.rkt)
(define-syntax define-place ...
(raco make)
```
[insert programming here]

game implementation overview
Simple Pattern-Based Macros

(define-syntax-rule 
  
)

- **define-syntax-rule** indicates a simple-pattern macro definition
Simple Pattern-Based Macros

(define-syntax-rule pattern
  template)

• A pattern to match

• Produce result from template
Simple Pattern-Based Macros

(define-syntax-rule (swap a b))

- Pattern for this macro: \texttt{(swap a b)}
- Each identifier matches anything in use
  
  \begin{align*}
  (\text{swap } x \ y) & \implies a \text{ is } x \\
  & \quad b \text{ is } y \\
  (\text{swap } 9 \ (+ \ 1 \ 7)) & \implies a \text{ is } 9 \\
  & \quad b \text{ is } (+ \ 1 \ 7)
  \end{align*}
Simple Pattern-Based Macros

(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))

Bindings substituted into template to generate the result

(swap x y)  ⇒  (let ([tmp y])
             (set! y x)
             (set! x tmp))

(swap 9 (+ 1 7))  ⇒  (let ([tmp (+ 1 7)])
                      (set! (+ 1 7) 9)
                      (set! 9 tmp))
Pattern-Based Macros

(define-syntax flip
  
)

(let ([x 0] [y 1])
  (flip x y))

(let ([xb (box 0)] [yb (box 1)])
  (f xb yb))

(define (f xb yb)
  (flip in xb yb))
Pattern-Based Macros

(define-syntax flip
  )

- `define-syntax` indicates a macro definition
Pattern-Based Macros

(define-syntax flip
  (syntax-rules (in)
    )))

- `syntax-rules` means a pattern-matching macro
- `(in)` means that `in` is literal in patterns
Pattern-Based Macros

(define-syntax flip
  (syntax-rules (in)
    ([pattern template]
      ...
      [pattern template]))

• Any number of patterns to match
• Produce result from template of first match
Pattern-Based Macros

(define-syntax flip
  (syntax-rules (in)
    [(flip in a b) ....]
    [(flip a b) ]))

Two patterns for this macro

- \((\text{flip in } x_b \ y_b)\) matches first pattern
- \((\text{flip } x \ y)\) falls through to second pattern
Pattern-Based Macros

(define-syntax flip
  (syntax-rules (in)
    [[(flip in a b) (let ([tmp (unbox b)])
                            (set-box! b (unbox a))
                            (set-box! a tmp))]]

    [[(flip a b) (swap a b)]])

(flip in xb yb)  ⇒  (let ([tmp (unbox yb)])
                       (set-box! yb (unbox xb))
                       (set-box! xb tmp))

(flip x y)  ⇒  (swap x y)
Matching Sequences

Some macros need to match sequences

\((\text{rotate } x \ y)\)

\((\text{rotate } \text{red} \ \text{green} \ \text{blue})\)

\((\text{rotate } \text{front-left} \ \text{rear-right} \ \text{front-right} \ \text{rear-left})\)
Matching Sequences

(define-syntax rotate
  (syntax-rules ()
    [(rotate a) (void)]
    [(rotate a b c ...) (begin
                           (swap a b)
                           (rotate b c ...))])))

• ... in a pattern: multiple of previous sub-pattern

   (rotate x y z w) ⇒ c is z w

• ... in a template: multiple instances of previous sub-template

   (rotate x y z w) ⇒ (begin
                        (swap x y)
                        (rotate y z w))
Matching Sequences

\[
\text{(define-syntax rotate}
\begin{array}{c}
\text{(syntax-rules ()}
\begin{array}{c}
\text{[(rotate a c ...)} \\
\text{(shift-to (c ... a) (a c ...)))]})
\end{array}
\end{array}
\text{)}
\]

\[
\text{(define-syntax shift-to}
\begin{array}{c}
\text{(syntax-rules ()}
\begin{array}{c}
\text{[(shift-to (from0 from ...)(to0 to ...))}
\text{(let ([tmp from0])}
\text{\begin{array}{c}
\text{(set! to from) ...}
\text{(set! to0 tmp)) ]})
\end{array}
\end{array}
\end{array}
\text{)}
\]

- ... maps over same-sized sequences
- ... duplicates constants paired with sequences
[insert programming here]

complete game implementation
Macro Scope

(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))

What if we swap a variable named tmp?

(let ([tmp 5] [other 6])
  (swap tmp other))

(let ([tmp 5] [other 6])
  (set! other tmp)
  (set! tmp tmp)))
Macro Scope

(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))

What if we swap a variable named tmp?

(let ([tmp 5] other 6)
  (swap tmp other))

This expansion would break scope
Macro Scope

(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))

What if we swap a variable named tmp?

(let ([tmp 5] [other 6])
  (swap tmp other)) → (let ([tmp 5] [other 6])
  (let ([tmp₁ other])
    (set! other tmp₁)
    (set! tmp tmp₁)))

Rename the introduced binding
Macro Scope: Local Bindings

Macro scope means that local macros work, too:

```scheme
(define (f x)
  (define-syntax swap-with-arg
    (syntax-rules ()
      [(swap-with-arg y) (swap x y)]))

(let ([z 12]
       [x 10])
  ; Swaps z with original x:
  (swap-with-arg z))
)
How Macro Scope Works

\[\text{define-syntact-rule (swap a b)}\]
\[\text{(let ([tmp b])}\]
\[\text{(set! b a)}\]
\[\text{(set! a tmp))}\]

Seems obvious that \text{tmp} can be renamed...
How Macro Scope Works

```
(define-syntax-rule (swap a b)
  (let-one [tmp b]
    (set! b a)
    (set! a tmp)))
```
How Macro Scope Works

```
(define-syntax-rule (swap a b)
  (let-one [tmp b]
    (set! b a)
    (set! a tmp)))
```

Can rename `tmp`:

```
(define-syntax-rule (let-one (x v) body)
  (let ([x v]) body))
```
How Macro Scope Works

\[
\text{(define-syntax-rule (swap a b)} \\
\quad \text{(let-one [tmp b]} \\
\quad \quad \text{(set! b a)} \\
\quad \quad \text{(set! a tmp)))}
\]

\textit{Cannot} rename \texttt{tmp}:

\[
\text{(define-syntax (let-one (x v) body)} \\
\quad \text{(list 'x v body))}
\]
How Macro Scope Works

\[(\text{define-syntax-rule (swap a b)}\]
\[(\text{let-one [tmp b]}\]
\[(\text{set! b a)}\]
\[(\text{set! a tmp)})\]

\textit{Cannot} rename \texttt{tmp}:

\[(\text{define-syntax (let-one (x v) body)}\]
\[(\text{list 'x v body)})\]

Track identifier introductions, then rename only as binding forms are discovered
How Macro Scope Works

```
(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))
```

Tracking avoids capture by introduced variables

```
(let ([tmp 5] [other 6])
  (swap tmp other))  =>  (let ([tmp 5] [other 6])
                           (let^1 ([tmp^1 other])
                             (set!^1 other tmp)
                             (set!^1 tmp tmp^1)))
```

^1 means introduced by expansion

tmp^1 does not capture tmp
How Macro Scope Works

(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))

Tracking also avoids capture of introduced variables

(let ([set! 5] [let 6])
  (swap set! let))  =>  (let ([set! 5] [let 6])
                         (let¹ ([tmp¹ let])
                           (set!¹ let set!)
                           (set!¹ set! tmp¹)))

set! does not capture set!¹

et does not capture let¹
[insert programming here]

modular game implementation
Implicit Syntactic Forms

To change functions:

\[(\text{define-syntact-rule (lambda ..... ) .....})\]

To change function calls?

\[(\text{define-syntact-rule (#%app ..... ) .....})\]

\[(\text{expr}_1 \ldots \text{expr}_N)\]

is implicitly

\[(#%app \text{expr}_1 \ldots \text{expr}_N)\]
Implicit Syntactic Forms

\[
\texttt{#lang s-exp path} \\
form_1 \\
\ldots \\
form_N
\]

is implicitly

\[
\texttt{#lang s-exp path} \\
(\texttt{#%module-begin} \\
form_1 \\
\ldots \\
form_N)
\]
[insert programming here]

game module language
Transformer Definitions

In general, \texttt{define-syntax} binds a transformer procedure:

\begin{verbatim}
(define-syntax swap
  (syntax-rules ....))
\end{verbatim}

\begin{verbatim}
⇒

(define-syntax swap
  (lambda (stx)
    use syntax-object primitives to match \texttt{stx} and generate result ))
\end{verbatim}
Representing Code

```clojure
{ mark ,
  rename tmp\textsuperscript{2} to tmp\textsubscript{1},
  lambda = lambda @ kernel,
  ...
}
```
Representing Code

```
#'(lambda (x) x)
```

```
syntax-object
  datum
  srcloc
  context

(file.ss:1:12)

{ mark ^, 
  rename tmp^2 to tmp^1, 
  lambda = lambda @ kernel, 
  ... }
Expressions, Bindings, and Phases

(define-syntax three
  (lambda (stx) #'3))

(+ 1 (three))
Expressions, Bindings, and Phases

\[
\begin{align*}
(\text{define-syntax} & \ \text{three} \\
& (\text{lambda} \ (\text{stx}) \ #\,3)) \\
(+ & 1 \ (\text{three}))
\end{align*}
\]
Expressions, Bindings, and Phases

```scheme
(define-syntax three
  (lambda (stx) #'3))

(+ 1 (three))
```
Expressions, Bindings, and Phases

```scheme
(require (for-syntax "roman-numerals.rkt"))

(define-syntax three
  (lambda (stx)
    `(+ 1 ,,(roman->number "II")))))

(+ 1 (three))
```
Expressions, Bindings, and Phases

(require (for-syntax "roman-numerals.rkt"))

(define-syntax three
  (lambda (stx)
    #'(+ 1 #,(roman->number "II")))))

(+ 1 (three))
Expressions, Bindings, and Phases

(begin-for-syntax
  (define (roman->number str) ....))

(define-syntax three
  (lambda (stx)
    `(+ 1 #,(roman->number "II"))))

(+ 1 (three))
Expressions, Bindings, and Phases

(begin-for-syntax
  (define (roman->number str) ....))

(define-syntax three
  (lambda (stx)
    `(+ 1 #, (roman->number "II"))))

(+ 1 (three))
Matching Syntax and Having It, Too

`syntax-case` and `#` combine patterns and computation

```
(syntax-case stx-expr ()
  [pattern result-expr]
  ... 
  [pattern result-expr])

#'template
```
Matching Syntax and Having It, Too

```
(define-syntax-rule (swap a b)
  (let ([tmp b])
    (set! b a)
    (set! a tmp)))

⇒

(define-syntax swap
  (lambda (stx)
    (syntax-case stx ()
      [(swap a b) #'(let ([tmp b])
                       (set! b a)
                       (set! a tmp))])))
```
Matching Syntax and Having It, Too

Check for identifiers before expanding:

```
(define-syntex swap
  (lambda (stx)
    (syntax-case stx ()
      [(swap a b)
        (if (and (identifier? #'a)
                 (identifier? #'b))
          #'(let ([tmp b])
             (set! b a)
             (set! a tmp))
          (raise-syntex-error
           'swap "needs identifiers" stx))]))
```
[insert programming here]

“typed” game language
Parsing

Source → \( \ldots \)(\ldots)\ldots \) → S-expression → function
args
body
→ AST
Parsing

#lang path

Source → S-expression → AST

(\ldots (\ldots) \ldots)

function
args
body
Parsing

#lang path

(module name path ...)

Source → S-expression → AST

function

args

body
Parsing

#lang path

(module name path ...)

Source

S-expression

AST

(function
  args
  body)

(require path)
Parsing

#lang path

(module name path ...)

(... (....) ...)

S-expression

(require path)

function
args
body

Source

#reader path

AST
Parsing

#lang scheme
(define (hi) "Hello")

(module m scheme
  (define (hi) "Hello"))

(define
  hi function
  ()
  "Hello"

#lang scribble/doc
@((require
    scribble/manual)
@bold{Hi})

(module m doclang
  (require
    scribble/manual)
  (bold "Hi"))

(define
  doc
doc
  apply
  bold
  ("Hi")

#lang honu
1+2;

(module m honu
  (1 + 2 |;|))

(define
  print
  apply
  apply
  (1 2)
Parsing

Source ➔ (⋯ (⋯⋯⋯) ⋅⋅⋅) ➔ S-expression ➔ function
                                          args
                                          body ➔ AST
Read layer provides absolute control

\[(+ 1 2) \quad \text{@bold\{Hi\}} \quad 1+2\]
Expand layer can delay “inside” until after “outside”

```
(define-place start ....
  ([north house-front]
    [south desert]))

(define-place house-front ....
  ([in room]
    [south start]))

int is_odd(int x) {
  ... is_even(x-1);
}

int is_even(int x) {
  ... is_odd(x-1);
}
```
[insert programming here]

non-S-expression game language
Environment Support

Support at S-expression level is free

• Error source locations
• Check Syntax

Source-editing support requires more

• On-the-fly coloring
[insert demo here]

DrRacket editor support
Languages in Racket

#lang path

(module name path ...)

(...
    (...)
    ...
)

S-expression

(requires path)

function
  args
  body

AST

Source

)reader path

#reader path