The Racket Way

Matthew Flatt

PLT and University of Utah

Racket

- A dialect of Lisp and a descendant of Scheme

```
#lang racket

; Grep stdin for "Racket":
(for ([line (in-lines)])
  (when (regexp-match? #rx"Racket" line)
    (displayln line)))
```

Racket

- A dialect of Lisp and a descendant of Scheme
- Optimizing bytecode+JIT compiler

```
....
(+ x y)
....
```

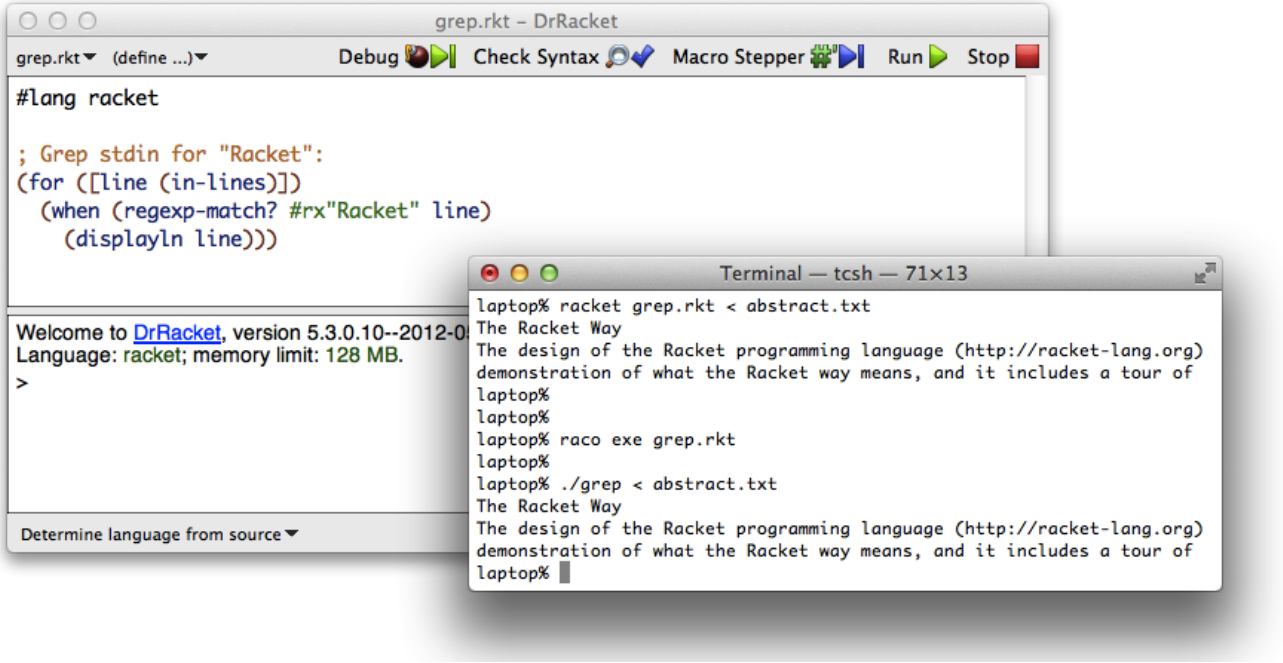
⇒

```
....
test $0x1, eax
je   not_fixnum_plus
test $0x1, ecx
je   not_fixnum_plus
xor  $0x1, eax
add  ecx, eax
....
```

performance comparable to Clojure, Go, OCaml

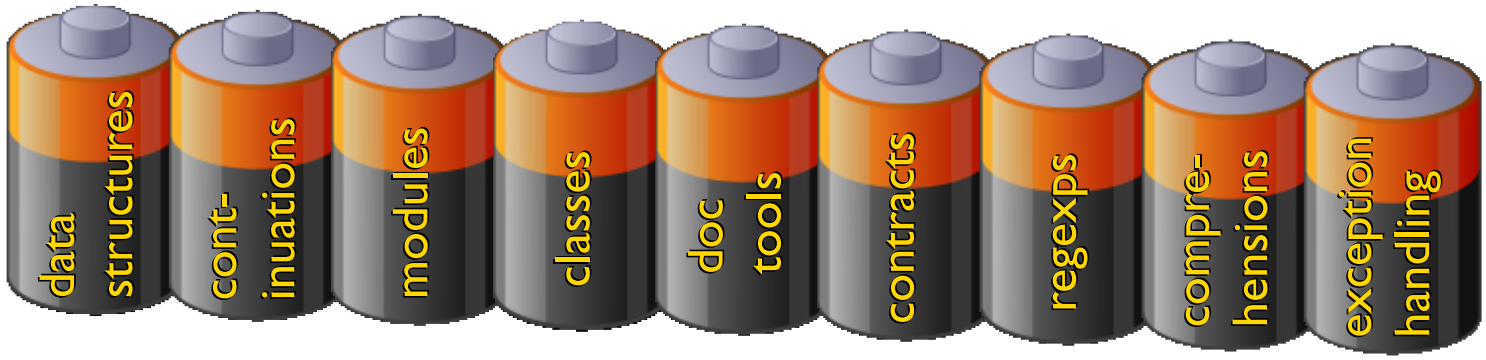
Racket

- A dialect of Lisp and a descendant of Scheme
- Optimizing bytecode+JIT compiler
- Command-line tools plus DrRacket IDE



Racket

- A dialect of Lisp and a descendant of Scheme
- Optimizing bytecode+JIT compiler
- Command-line tools plus DrRacket IDE
- Batteries included

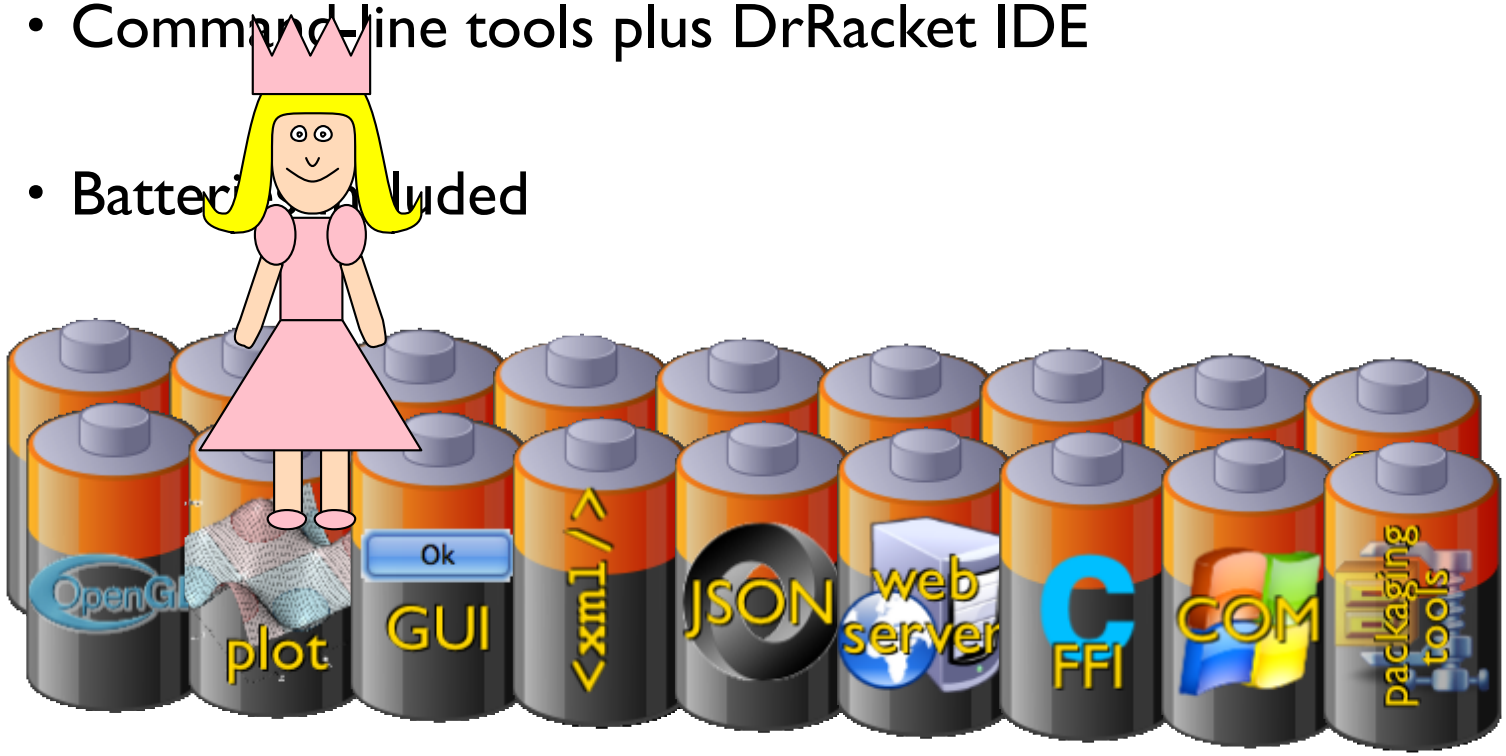


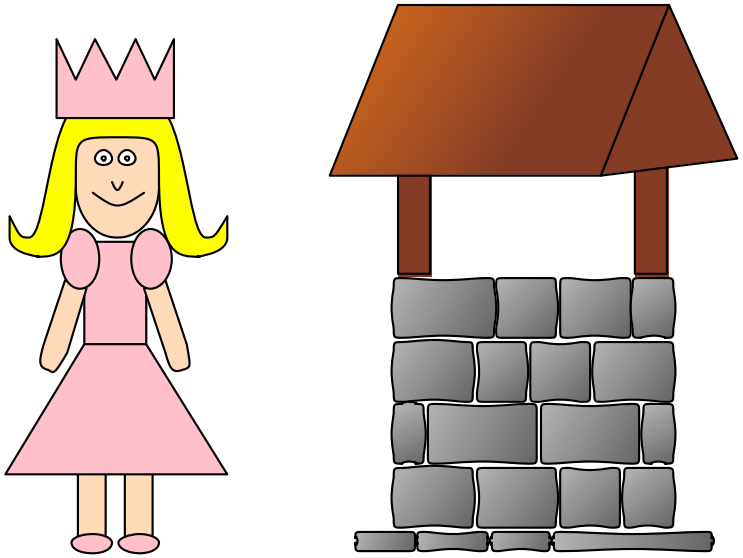
Racket

- A dialect of Lisp and a descendant of Scheme
- Optimizing bytecode+JIT compiler
- Command-line tools plus DrRacket IDE
- Batteries included

Racket

- A dialect of Lisp and a descendant of Scheme
- Optimizing bytecode+JIT compiler
- Command-line tools plus DrRacket IDE
- Batteries included

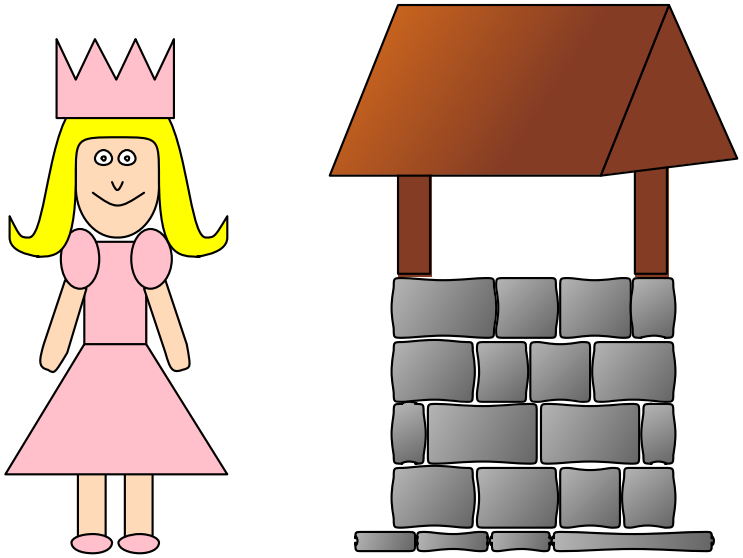


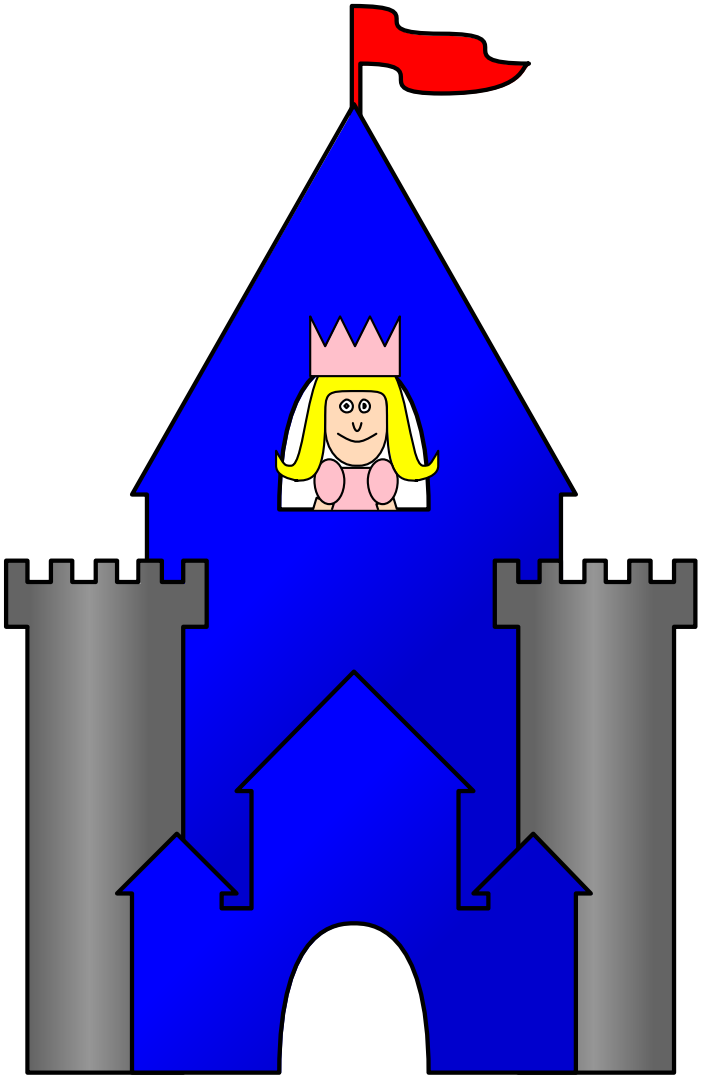


Wishing Well

1

wish per princess

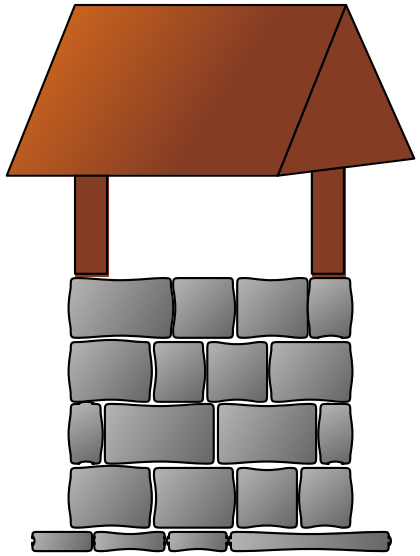




Wishing Well

1

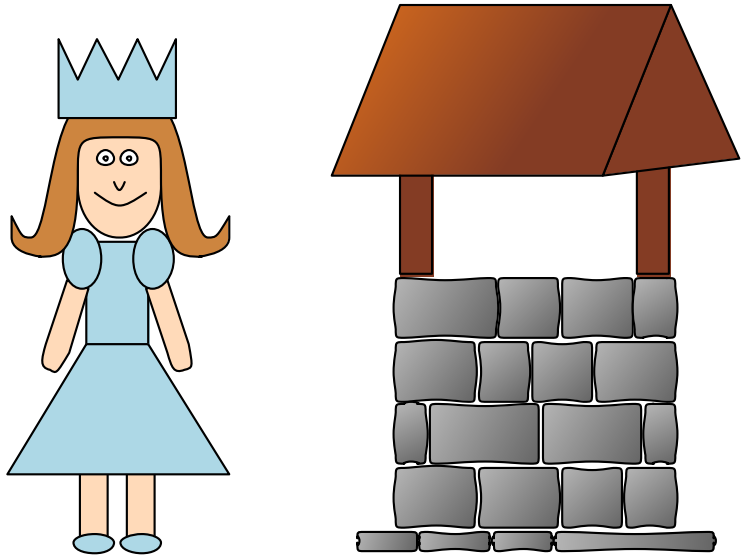
wish per princess

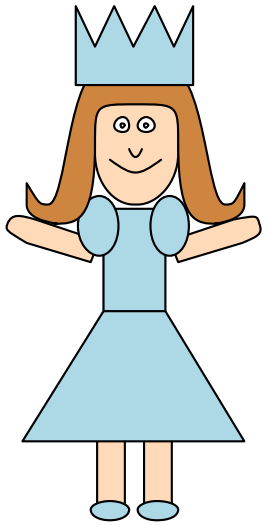


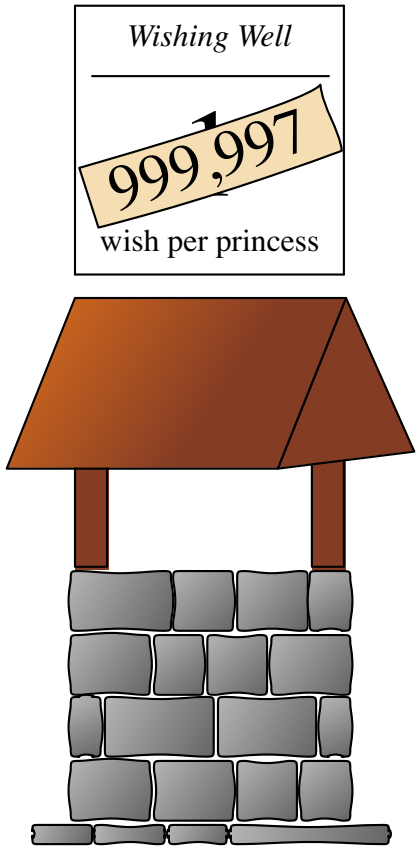
Wishing Well

1

wish per princess



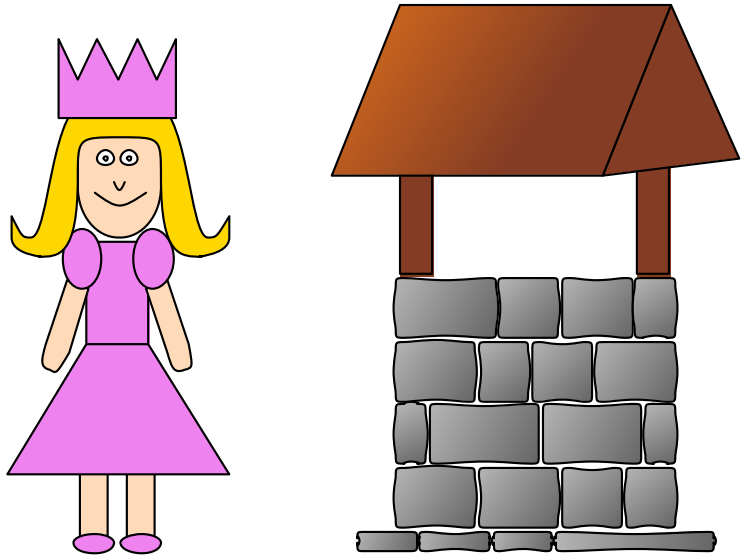



Wishing Well
1
999,997
wish per princess

Wishing Well

1

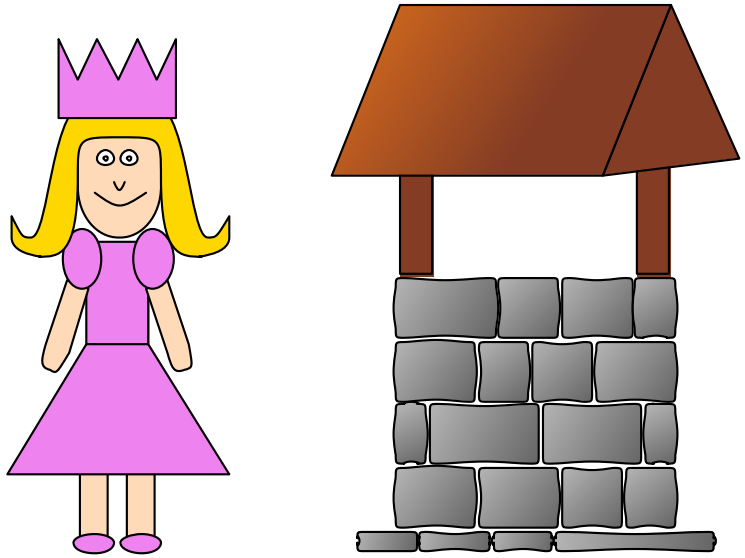
wish per princess

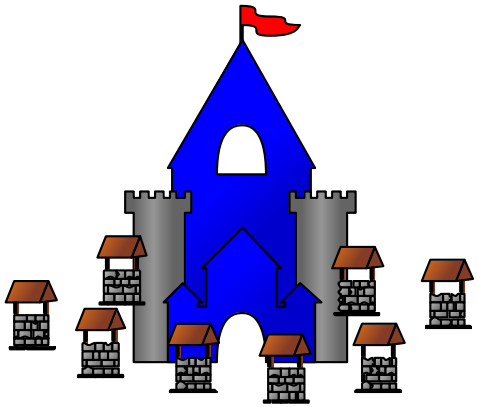


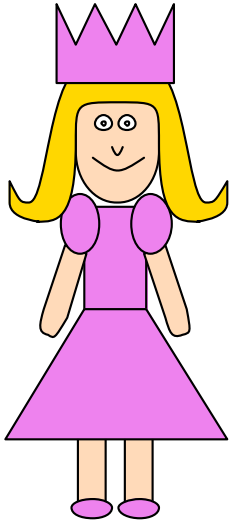
Wishing Well

1

wish per princess



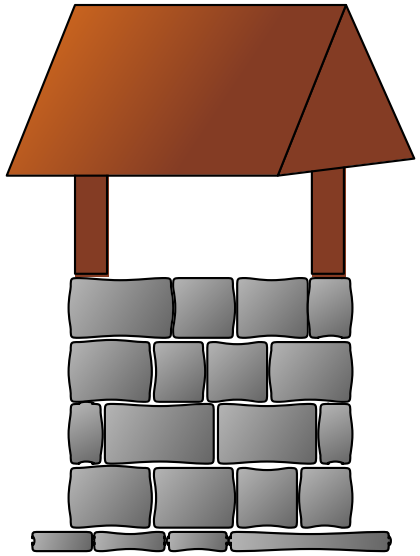


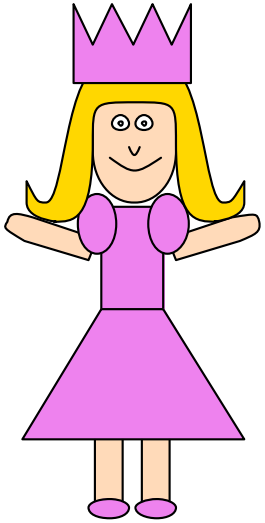


Wishing Well

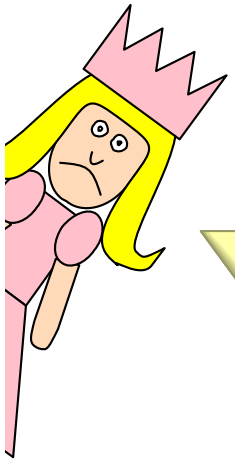
1

wish per princess



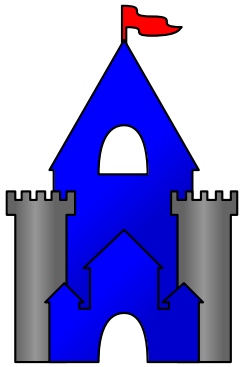


The End

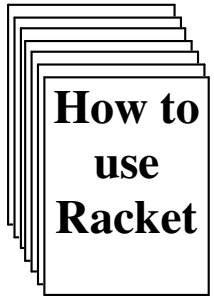


I'm confused... are we still talking about **Racket**?

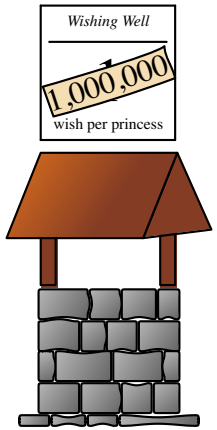
The End



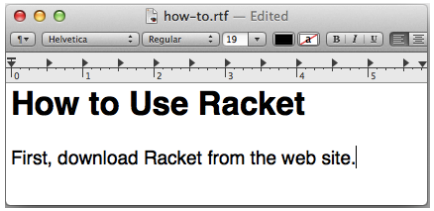
=



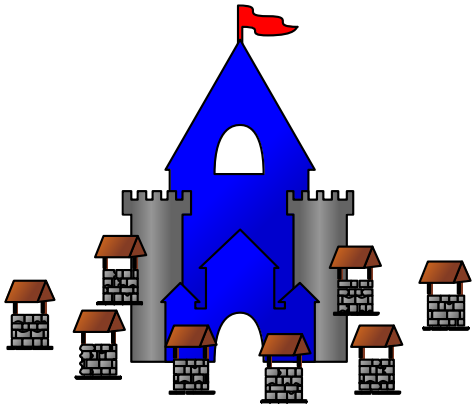
document



=



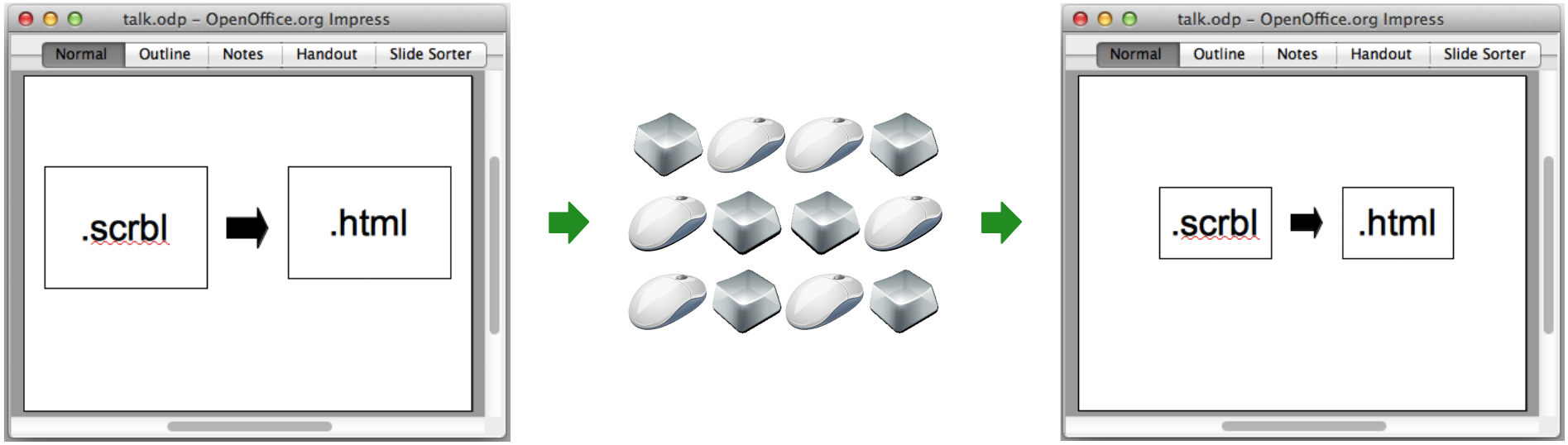
tool
for
documents



=

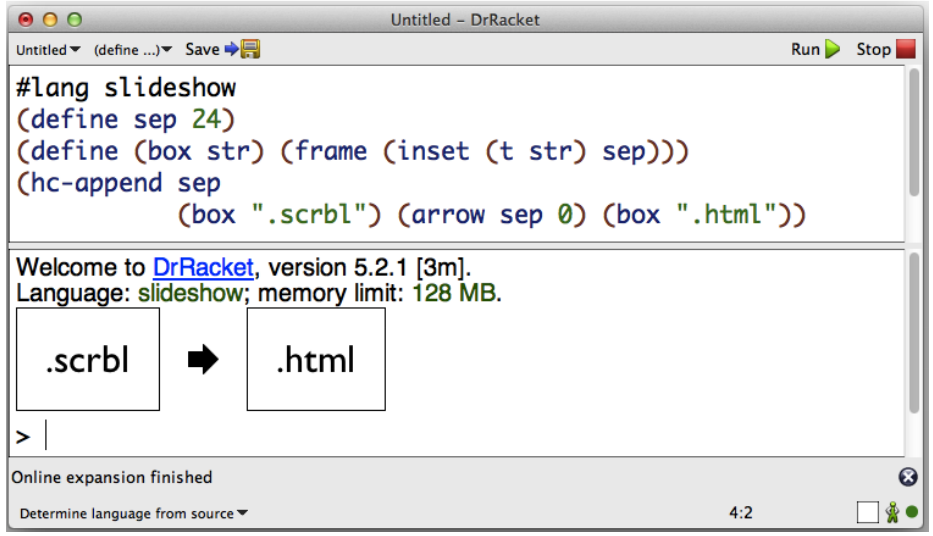
desktop
of
tools

What a Programmer Wants



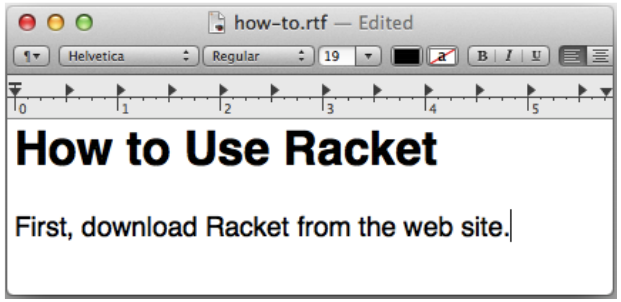
Programmer

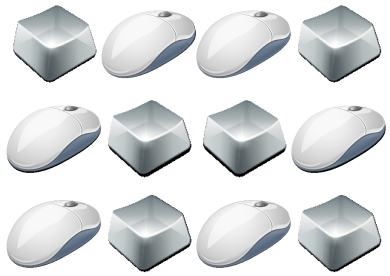
What a Programmer Wants



Programmer

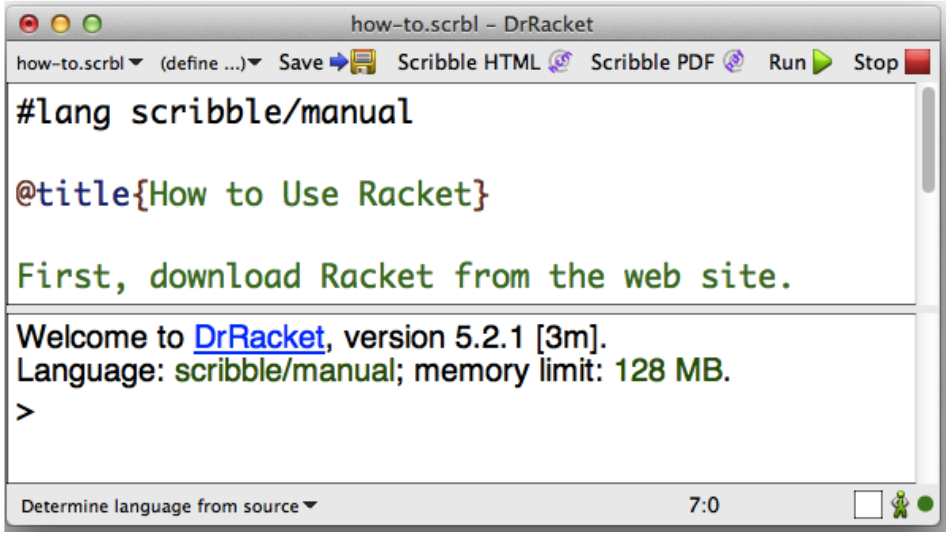
What a Programmer Wants



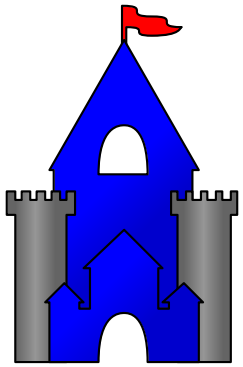


Programmer

What a Programmer Wants

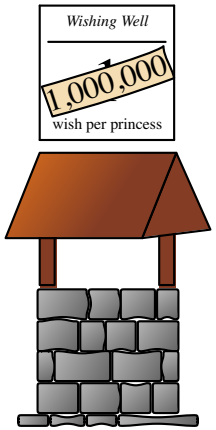


Programmer

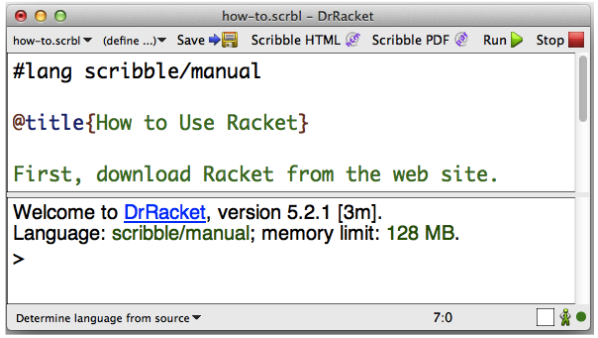


=

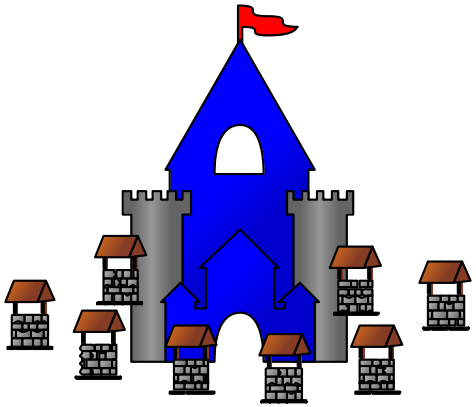
document



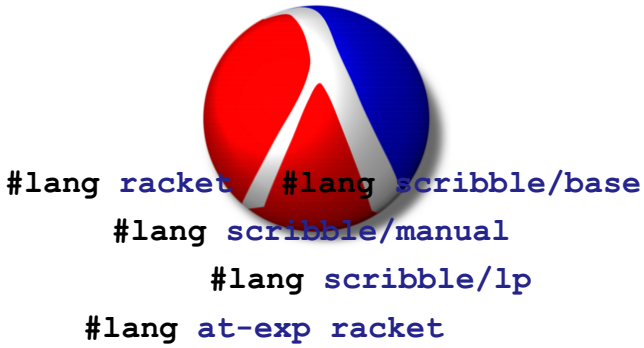
=



language
for
documents

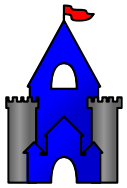


=

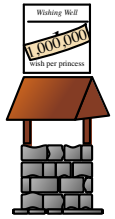


language
for
languages

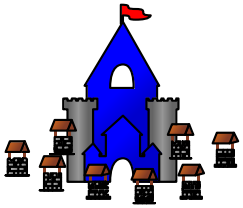
The Racket Way



Everything is a program

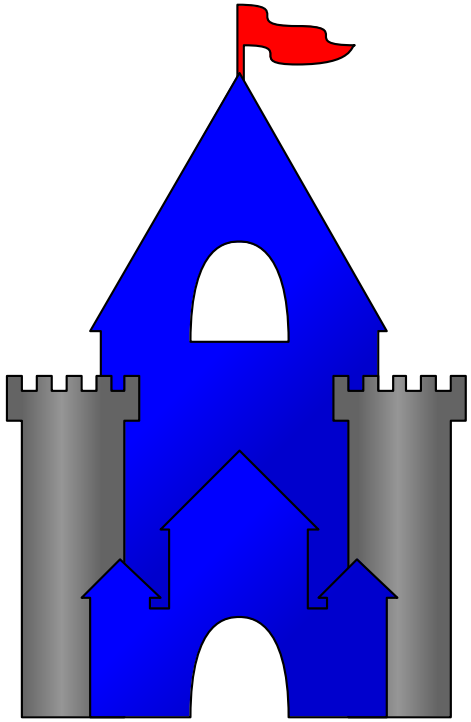


Concepts are programming language constructs

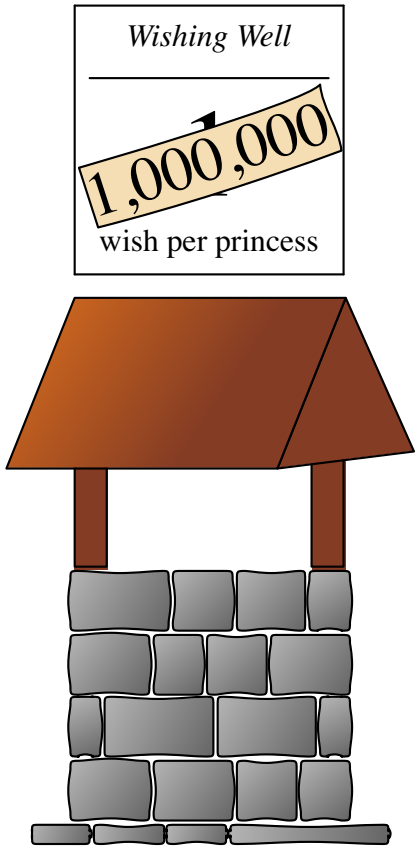


The programming language is extensible

Everything is a Program

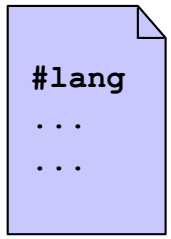


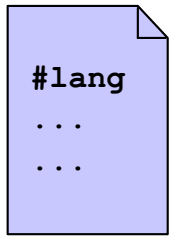
Concepts as Language Constructs



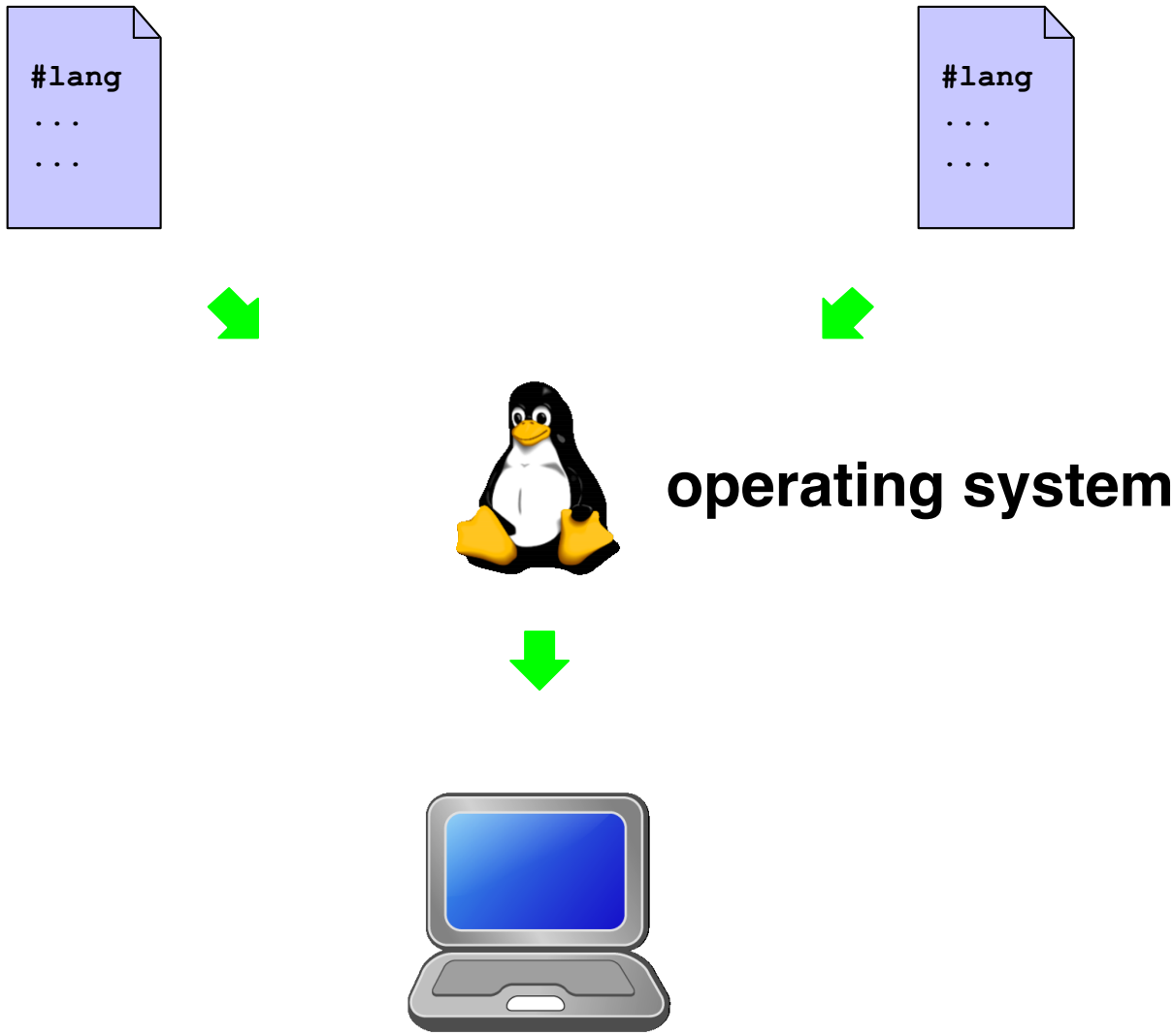
Processes

Processes

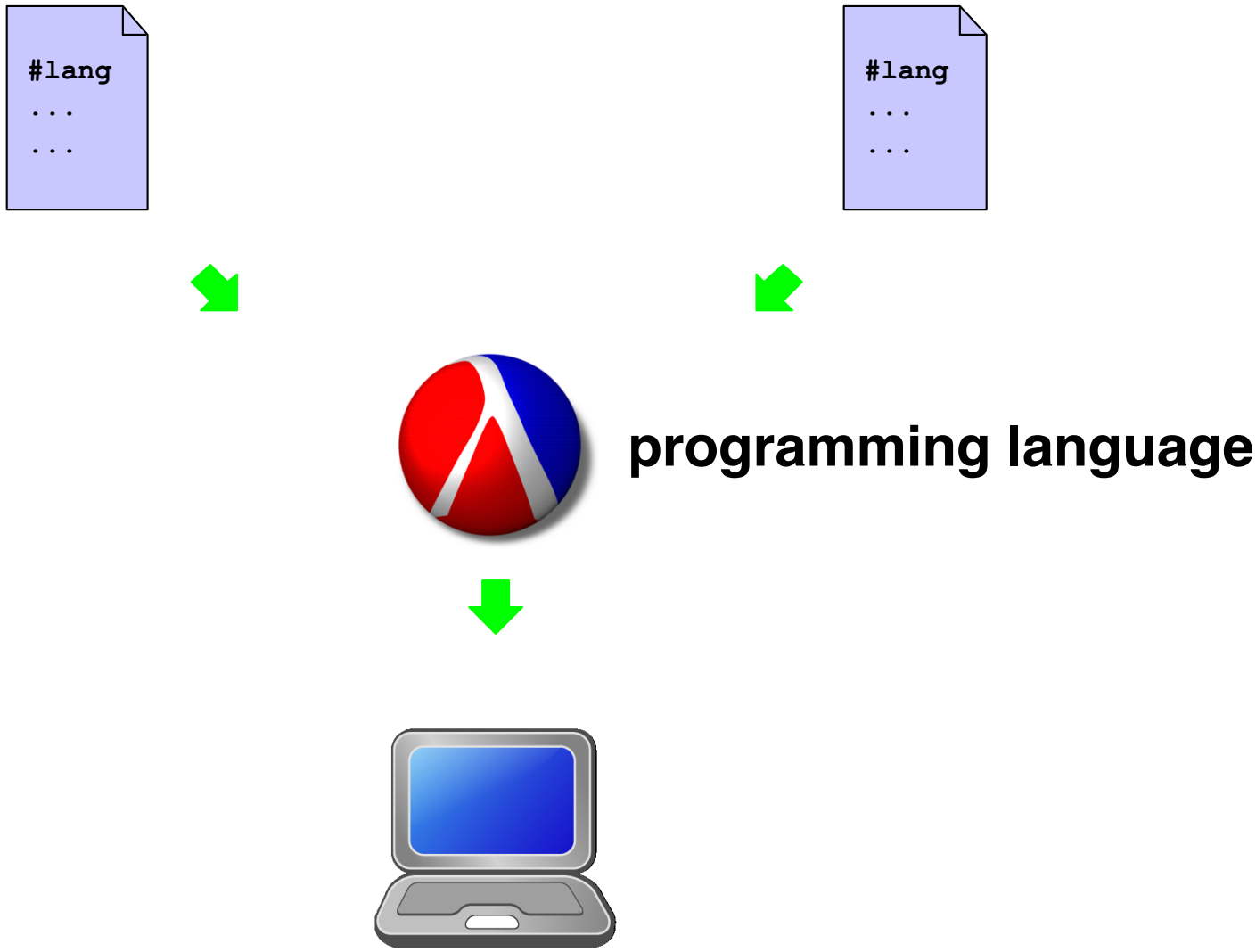




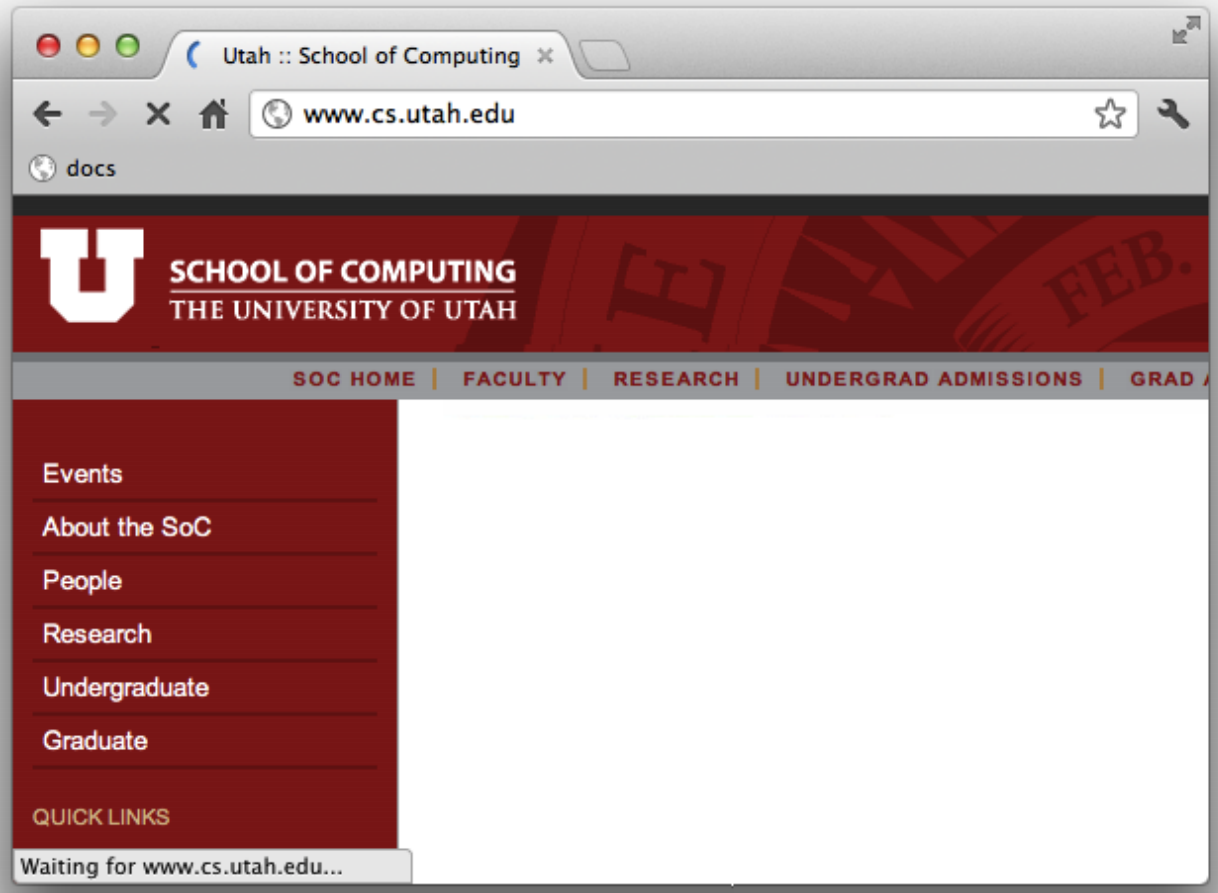
Processes



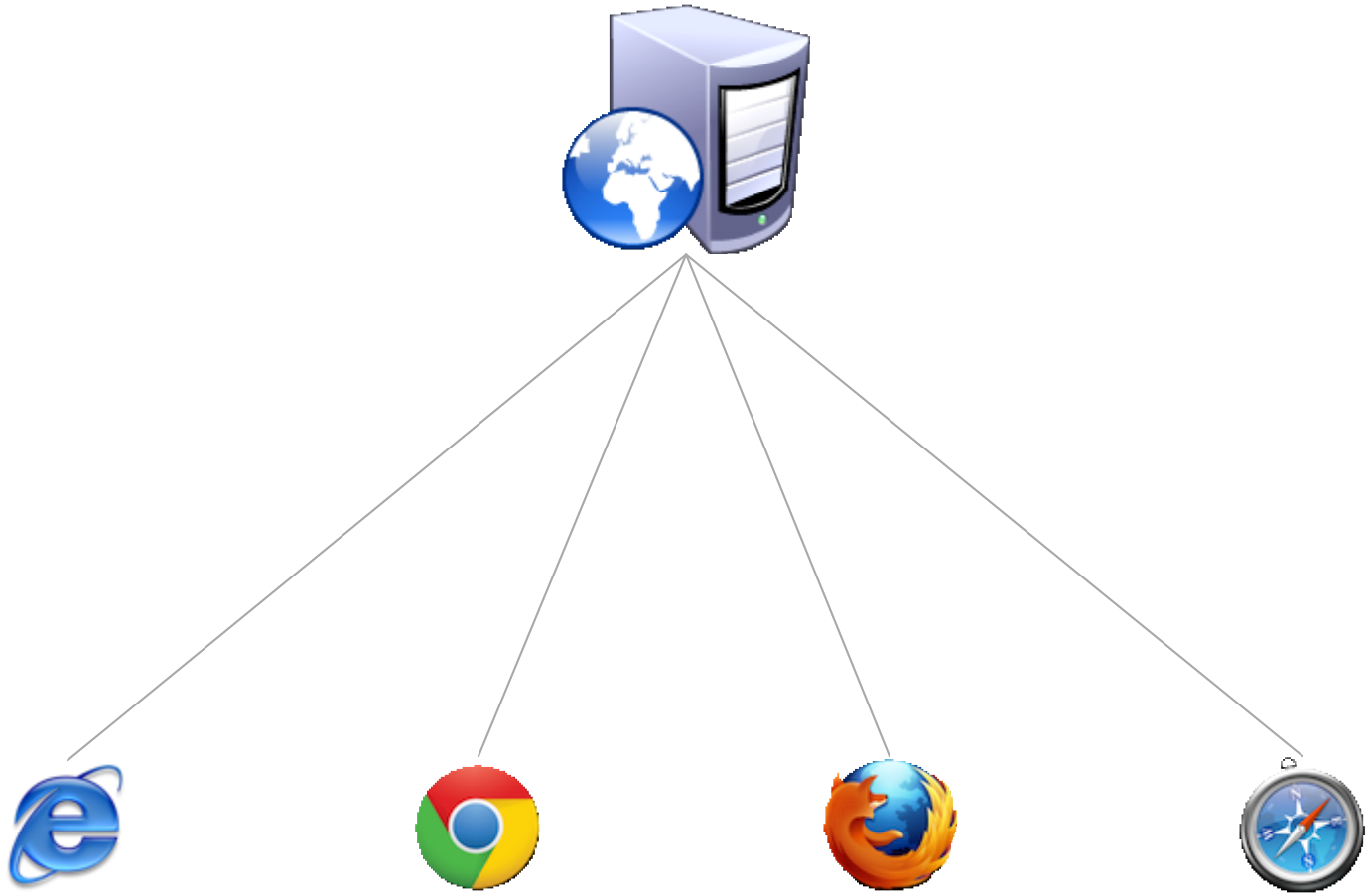
Processes



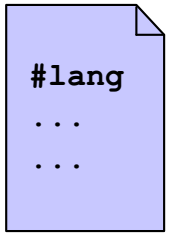
Process Examples



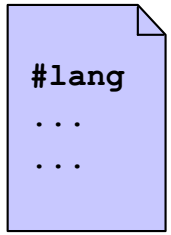
Process Examples



Process Examples



DrRacket



user's program

Run DrRacket

Process Concepts

- Threads
- Process-specific state (e.g., current directory)
- Graphical event loops
- Debugging capabilities
- Resource accounting
- Terminate a process and reclaim resources

Threads

```
(require "spin-display.rkt")
```
eval

```
(define (spin)
  (rotate-a-little)
  (sleep 0.1)
  (spin))
```

```
(define spinner (thread spin))
```
eval

```
(kill-thread spinner)
```
eval

Parameters – Thread-local State

```
(printf "Hello\n")
(fprintf (current-output-port) "Hola\n")
(fprintf (current-error-port) "Goodbye\n")
(error "Ciao")
```

eval

```
(parameterize ([current-error-port (current-output-port)])
  (error "Au Revoir"))
```

eval

```
(parameterize ([current-error-port (current-output-port)])
  (thread
   (lambda ()
     (error "再见"))))
```

eval

Eventspaces – Concurrent GUIs

```
(thread (lambda () (message-box "One" "Hi")))
(thread (lambda () (message-box "Two" "Bye")))
```

eval

```
(thread (lambda () (message-box "One" "Hi")))
(parameterize ([current-eventspace (make-eventspace)])
  (thread (lambda () (message-box "Two" "Bye"))))
```

eval

Custodians – Termination and Clean-up

```
(define c (make-custodian))
(parameterize ([current-custodian c])
  ....)
```

eval

```
(custodian-shutdown-all c)
```

eval

Custodians – Resource Limits

```
(define (run-away)
  (cons 1 (run-away)))

(custodian-limit-memory c 2000000)

(parameterize ([current-custodian c])
  ....
  (thread run-away))
```

eval

Building a Programming Environment

RacketEsq, a mini DrRacket

GUI – Frame

```
(define frame
  (new frame%
       [label "RacketEsq"]
       [width 400]
       [height 175]))

(send frame show #t)
```

eval

GUI – Reset Button

```
(new button%
     [label "Reset"]
     [parent frame]
     [callback (lambda (b e)
                 (reset-program))])
```

eval

GUI – Interaction Area

```
(define repl-display-canvas
  (new editor-canvas%
       [parent frame]))
```

eval

GUI – Interaction Buffer

```
(define esq-text%
  (class text% .... (evaluate str) ....))

(define repl-editor (new esq-text%))
(send repl-display-canvas set-editor repl-editor)
```

eval

Evaluator

```
(define (evaluate expr-str)
  (thread
   (lambda ()
     (print (eval (read (open-input-string expr-str))))
     (newline)
     (send repl-editor new-prompt))))
```

eval

Evaluator Output

```
(define user-output-port
  (make-output-port .... repl-editor ....))

(define (evaluate expr-str)
  (parameterize ([current-output-port user-output-port])
    (thread
     (lambda ()
       ....)))))
```

eval

Evaluating GUIs

```
(define user-eventspace (make-eventspace))

(define (evaluate expr-str)
  (parameterize ([current-output-port user-output-port]
                 [current-eventspace user-eventspace])
    (thread
     (lambda ()
       ....))))
```

eval

Custodian for Evaluation

```
(define user-custodian (make-custodian))

(define user-eventspace
  (parameterize ([current-custodian user-custodian])
    (make-eventspace)))

(define (evaluate expr-str)
  (parameterize ([current-output-port user-output-port]
                 [current-eventspace user-eventspace]
                 [current-custodian user-custodian])
    (thread
     (lambda ()
       ....))))
```

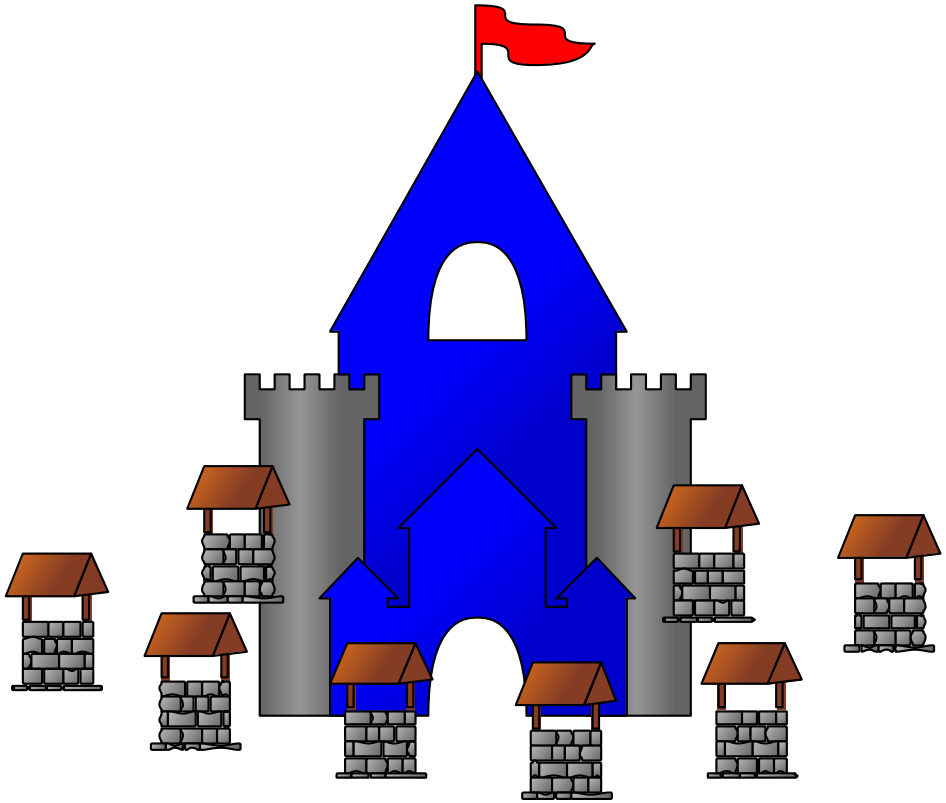
eval

Reset Evaluation

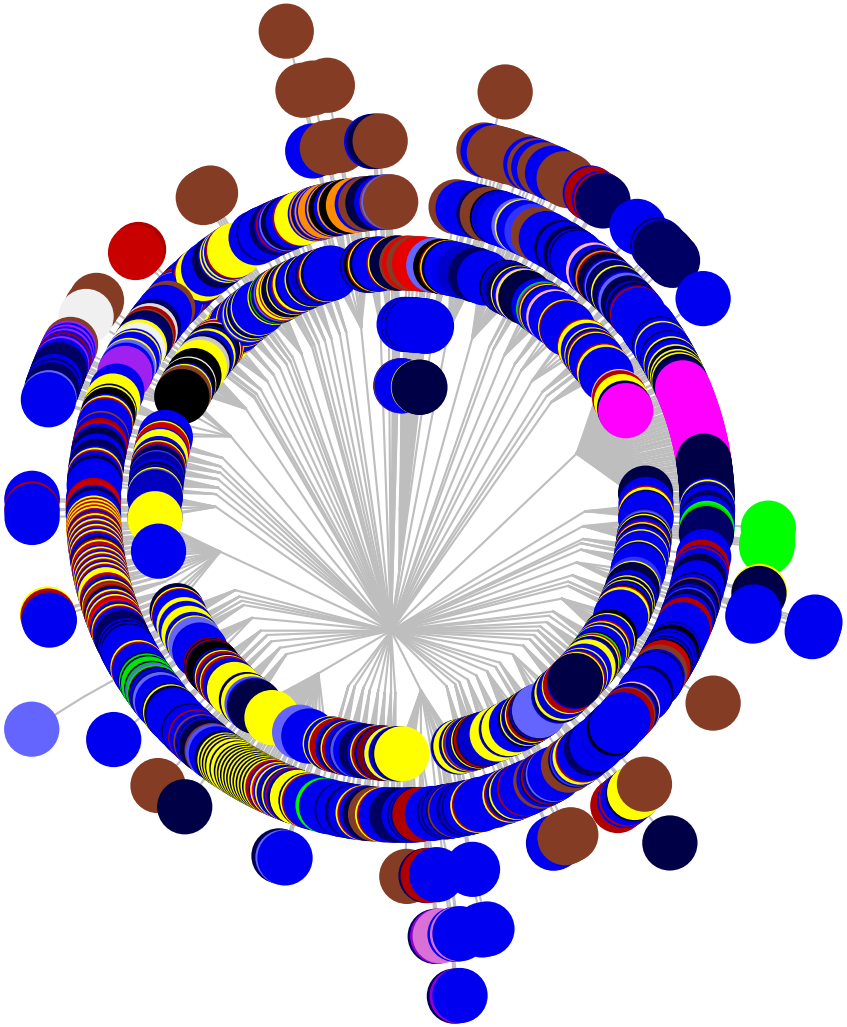
```
(define (reset-program)
  (custodian-shutdown-all user-custodian)

  (set! user-custodian (make-custodian))
  (parameterize ((current-custodian user-custodian))
    (set! user-eventspace (make-eventspace))
    (send repl-editor reset)))
```

eval

Language Extensibility



Languages in the Racket Distribution



- mzscheme
- racket
- racket/private
- racket/unit
- racket/private/base
- #%kernel
- racket/load
- racket/base
- racket/private/provider
- racket/signature
- slideshow
- racket/gui
- at-exp scheme/base
- at-exp racket/base
- scribble/doc
- scribble/manual
- scribble/base/reader
- scribble/lp
- deinprogramm/DMdA
- htdp/asl
- htdp/isl+
- htdp/bsl
- frtime/lang-utils
- frtime/frtime-lang-only
- frtime
- syntax/module-reader
- web-server/insta
- meta/web
- web-server
- srfi/provider
- typed/racket
- typed-scheme/minimal
- r6rs
- r5rs
- setup/infotab
- everything else

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.
>
```

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 i ansitive verbs
  - place-loca bs
  - thing-spec

Objects?

Methods?

Need not only serialize, but save & restore variables

Must convert between string command and method call

Adventure Game Data

```
; A place is
;  (place symbol list-of-thing dict-of-verb-to-function)
(struct place (desc [things #:mutable] actions))

; A thing is
;  (thing symbol any dict-of-verb-to-function)
(struct thing (name [state #:mutable] actions))

; A verb is
;  (verb list-of-symbol string boolean)
(struct verb (aliases desc transitive?))
```

all code at "Creating Languages in Racket," *ACM Queue*, November 2011

Version 0: Longhand

```
(define north (verb (list 'north 'n) "go north" #f))
(record-element! 'north north)

(define south (verb (list 'south 's) "go south" #f))
(record-element! 'south south)
```

.....

```
(define get (verb (list 'get 'grab 'take) "take" #t))
(record-element! 'get get)

(define put (verb (list 'put 'drop 'leave) "drop" #t))
(record-element! 'put put)
```

.....

Version 0: Longhand

```
(define door
  (thing 'door
         #f
         (list
          (cons open
                (lambda ()
                  (if (have-thing? key)
                      (begin
                        (set-thing-state! door 'open)
                        "The door is now unlocked and open.")
                      "The door is locked.")))
          (cons close
                (lambda ()
                  (begin
                    (set-thing-state! door #f)
                    "The door is now closed.")))
          (cons knock
                (lambda ()
                  "No one is home.")))))
(record-element! 'door door)
```

Version 1: Syntactic Abstraction

define a pattern-based macro

definition

```
(define-syntax-rule (define-thing id
                      [vrb expr] ...)
  (begin
    (define id
      (thing 'id #f (list (cons vrb (lambda () expr)) ...)))
    (record-element! 'id id)))
```

---

use

```
(define-thing door
  [close (if (have-thing? key)
             (begin
               (set-thing-state! door 'open)
               "The door is now unlocked and open.")
             "The door is locked.")]
  [open (begin
          (set-thing-state! door #f)
          "The door is now closed.")]
  [knock "No one is home."])
```

Version 1: Syntactic Abstraction

pattern

definition

```
(define-syntax-rule (define-thing id
                      [vrb expr] ...)
  (begin
    (define id
      (thing 'id #f (list (cons vrb (lambda () expr)) ...)))
    (record-element! 'id id)))
```

---

use

```
(define-thing door
  [close (if (have-thing? key)
             (begin
               (set-thing-state! door 'open)
               "The door is now unlocked and open.")
             "The door is locked.")]
  [open (begin
          (set-thing-state! door #f)
          "The door is now closed.")]
  [knock "No one is home."])
```

Version 1: Syntactic Abstraction

definition

```
(define-syntax-rule (define-thing id
                      [vrb expr] ...)
  (begin
    (define id
      (thing 'id #f (list (cons vrb (lambda () expr)) ...)))
    (record-element! 'id id)))
```

template

---

use

```
(define-thing door
  [close (if (have-thing? key)
             (begin
               (set-thing-state! door 'open)
               "The door is now unlocked and open.")
             "The door is locked.")]
  [open (begin
          (set-thing-state! door #f)
          "The door is now closed.")]
  [knock "No one is home."])
```

Version 1: Syntactic Abstraction

definition

```
(define-syntax-rule (define-thing id
                      [vrb expr] ...)
  (begin
    (define id
      (thing 'id #f (list (cons vrb (lambda () expr)) ...)))
    (record-element! 'id id)))
```

---

use

```
(define-thing door
  [close (if (have-thing? key)
             (begin
               (set-thing-state! door 'open)
               "The door is now unlocked and open.")
             "The door is locked.")]
  [open (begin
          (set-thing-state! door #f)
          "The door is now closed.")]
  [knock "No one is home."])
```

Version 1: Syntactic Abstraction

definition

```
(define-syntax-rule (define-thing id
                      [vrb expr] ...)
  (begin
    (define id
      (thing 'id #f (list (cons vrb (lambda () expr)) ...)))
    (record-element! 'id id)))
```

---

use

```
(define-thing door
  [close (if (have-thing? key)
             (begin
               (set-thing-state! door 'open)
               "The door is now unlocked and open.")
             "The door is locked.")]
  [open (begin
          (set-thing-state! door #f)
          "The door is now closed.")]
  [knock "No one is home."])
```

Version 1: Syntactic Abstraction

definition

```
(define-syntax-rule (define-thing id
                      [vrb expr] ...)
  (begin
    (define id
      (thing 'id #f (list (cons vrb (lambda () expr)) ...)))
    (record-element! 'id id)))
```

---

use

```
(define-thing door
  [close (if (have-thing? key)
             (begin
               (set-thing-state! door 'open)
               "The door is now unlocked and open.")
             "The door is locked.")]
  [open (begin
          (set-thing-state! door #f)
          "The door is now closed.")]
  [knock "No one is home."])
```

Version 1: Syntactic Abstraction

definition

```
(define-syntax-rule (define-verbs all-id
                      [id spec ...] ...)
  (begin
    (define-one-verb id spec ...) ...
    (record-element! 'id id) ...
    (define all-id (list id ...))))

(define-syntax define-one-verb
  (syntax-rules (= _)
    [(define-one-verb id (= alias ...) desc)
     (define id (verb (list 'id 'alias ...) desc #f))]
    [(define-one-verb id _ (= alias ...) desc)
     (define id (verb (list 'id 'alias ...) desc #t))]))
```

---

use

```
(define-verbs all-verbs
  [north (= n) "go north"]
  [south (= s) "go south"]
  .....
  [get _ (= grab take) "take"]
  [put _ (= drop leave) "drop"]
  .....)
```

no _: intransitive

has _: transitive

Version 1: Syntactic Abstraction

definition

```
(define-syntax-rule (define-verbs all-id
                      [id spec ...] ...)
  (begin
    (define-one-verb id spec ...) ...
    (record-element! 'id id) ...
    (define all-id (list id ...))))

(define-syntax define-one-verb
  (syntax-rules (= _)
    [(define-one-verb id (= alias ...) desc)
     (define id (verb (list 'id 'alias ...) desc #f))]
    [(define-one-verb id _ (= alias ...) desc)
     (define id (verb (list 'id 'alias ...) desc #t))]))
```

helper macro

---

use

```
(define-verbs all-verbs
  [north (= n) "go north"]
  [south (= s) "go south"]
  .....
  [get _ (= grab take) "take"]
  [put _ (= drop leave) "drop"]
  .....)
```

Version 1: Syntactic Abstraction

definition

```
(define-syntax-rule (define-verbs all-id
                      [id spec ...] ...)
  (begin
    (define-one-verb id spec ...) ...
    (record-element! 'id id) ...
    (def                    (list id ...))))
```

define a macro

```
(define-syntax define-one-verb
  (syntax-rules
    [(define-one-verb id (= alias ...) desc)
     (define id (verb (list 'id 'alias ...) desc #f))]
    [(define-one-verb id _ (= alias ...) desc)
     (define id (verb (list 'id 'alias ...) desc #t))]))
```

match multiple patterns

---

use

```
(define-verbs all-verbs
  [north (= n) "go north"]
  [south (= s) "go south"]
  .....
  [get _ (= grab take) "take"]
  [put _ (= drop leave) "drop"]
  .....)
```

Version 1: Syntactic Abstraction

definition

```
(define-syntax-rule (define-verbs all-id
                      [id spec ...] ...)
  (begin
    (define-one-verb id spec ...) ...
    (record-element! 'id id) ...
    (define all-id (list id ...))))

(define-syntax define-one-verb
  (syntax-rules (= _)
    [(define-one-verb id (= alias ...) desc)
     (define id (verb (list 'id 'alias ...) desc #f))]
    [(define-one-verb id _ (= alias ...) desc)
     (define id (verb (list 'id 'alias ...) desc #t))]))
```

literals

---

use

```
(define-verbs all-verbs
  [north (= n) "go north"]
  [south (= s) "go south"]
  .....
  [get _ (= grab take) "take"]
  [put _ (= drop leave) "drop"]
  .....)
```

Version 1: Syntactic Abstraction

definition

```
(define-syntax-rule (define-verbs all-id
                      [id spec ...] ...)
  (begin
    (define-one-verb id spec ...) ...
    (record-element! 'id id) ...
    (define all-id (list id ...))))

(define-syntax define-one-verb
  (syntax-rules (= _)
    [(define-one-verb id (= alias ...) desc)
     (define id (verb (list 'id 'alias ...) desc #f))]
    [(define-one-verb id _ (= alias ...) desc)
     (define id (verb (list 'id 'alias ...) desc #t))]))
```

---

use

```
(define-verbs all-verbs
  [north (= n) "go north"]
  [south (= s) "go south"]
  .....
  [get _ (= grab take) "take"]
  [put _ (= drop leave) "drop"]
  .....)
```

Version 2: Syntactic Extension

```
#lang racket
(provide define-verbs define-thing
         define-place define-everywhere
         .....)
.....

(define-syntax-rule (define-thing id
                      [vrb expr] ...)
  (begin
    (define id
      (thing 'id #f
             (list (cons vrb (lambda ()
                               expr))
                   ...)))
    (record-element! 'id id)))

.....
```

not exported, works anyway: lexical scope

txtadv.rkt

```
#lang racket
(require "txtadv.rkt")

(define-verbs .....)
(define-everywhere .....)
(define-thing .....) ...
(define-place .....) ...
```

world.rkt

Version 3: Module Language

```
#lang racket
(provide define-verbs define-thing
         define-place define-everywhere
         .....

         cons first rest lambda .....

         #%module-begin)
.....
```

txtadv.rkt

specify language

```
#lang s-exp "txtadv.rkt"

(define-verbs .....)
(define-everywhere .....)
(define-thing .....) ...
(define-place .....) ...
```

world.rkt

Version 3: Module Language

still using parentheses

```
#lang racket
(provide define-verbs define-thing
         define-place define-everywhere
         .....

         cons first rest lambda .....

         #%module-begin)
.....
```

txtadv.rkt

```
#lang s-exp "txtadv.rkt"

(define-verbs .....)
(define-everywhere .....)
(define-thing .....) ...
(define-place .....) ...
```

world.rkt

Version 3: Module Language

only **txtadv.rkt** exports available

```
#lang racket
(provide define-verbs define-thing
         define-place define-everywhere
         .....

         cons first rest lambda .....

         #%module-begin)
.....
```

txtadv.rkt

```
#lang s-exp "txtadv.rkt"

(define-verbs .....)
(define-everywhere .....)
(define-thing .....) ...
(define-place .....) ...
```

world.rkt

Version 3: Module Language

```
#lang racket
(provide define-verbs define-thing
         define-place define-everywhere
         .....
         cons first rest lambda .....

         #%module-begin)
.....
```

re-export from **racket**

txtadv.rkt

```
#lang s-exp "txtadv.rkt"

(define-verbs .....)
(define-everywhere .....)
(define-thing .....) ...
(define-place .....) ...
```

world.rkt

Version 3: Module Language

```
#lang racket
(provide define-verbs define-thing
         define-place define-everywhere
         .....

         cons first rest lambda .....

         #%module-begin)
.....
```

macro for whole module body

txtadv.rkt

```
#lang s-exp "txtadv.rkt"

(define-verbs .....)
(define-everywhere .....)
(define-thing .....) ...
(define-place .....) ...
```

world.rkt

Version 4: Types

```
(define-place room
  "You're in the house."
  [trophy desert]
  ([out house-front]))
```

~~*runtime error: message not understood*~~

syntax error: `desert' does not have type `thing'

Version 4: Types

start compile-time code

```
(begin-for-syntax
 (struct typed (id type) #:property prop:procedure .....))

(define-syntax check-type
  (lambda (stx)
    (syntax-case stx ()
      [(check-type id type)
       (let ([t (syntax-local-value #'id)])
         (unless (and (typed? t)
                      (equal? (syntax-e #'type) (typed-type t)))
           (raise-type-error))
         #'id)])))
```

```
(define-syntax-rule (define-thing id .....)
  (define-syntax id (typed #'gen-id "thing"))
  (define gen-id (thing .....))
  .....)

(define-syntax-rule (define-place id desc (thng ...) .....)
  ..... (place desc (list (check-type thng "thing") ...) .....)
  .....)
```

Version 4: Types

a compile-time record declaration

```
(begin-for-syntax
 (struct typed (id type) #:property prop:procedure .....))

(define-syntax check-type
  (lambda (stx)
    (syntax-case stx ()
      [(check-type id type)
       (let ([t (syntax-local-value #'id)])
         (unless (and (typed? t)
                      (equal? (syntax-e #'type) (typed-type t)))
           (raise-type-error))
         #'id)])))
```

```
(define-syntax-rule (define-thing id .....)
  (define-syntax id (typed #'gen-id "thing"))
  (define gen-id (thing .....))
  .....)

(define-syntax-rule (define-place id desc (thng ...) .....)
  ..... (place desc (list (check-type thng "thing") ...) .....)
  .....)
```

Version 4: Types

```
(begin-for-syntax
 (struct typed (id type) #:property prop:procedure .....))

(define-syntax check-type
  (lambda (stx)
    (syntax-case stx ()
      [(check-type id type)
       (let ([t (syntax-local-value #'id)])
         (unless (and (typed? t)
                      (equal? (syntax-e #'type) (typed-type t)))
           (raise-type-error))
         #'id)])))
```

bind *id* to a compile-time record

```
(define-syntax-rule (define-thing id .....)
  (define-syntax id (typed #'gen-id "thing"))
  (define gen-id (thing .....))
  .....)

(define-syntax-rule (define-place id desc (thng ...) .....)
  ..... (place desc (list (check-type thng "thing") ...) .....)
  .....)
```

Version 4: Types

```
(begin-for-syntax
 (struct typed (id type) #:property prop:procedure .....))

(define-syntax check-type
  (lambda (stx)
    (syntax-case stx ()
      [(check-type id type)
       (let ([t (syntax-local-value #'id)])
         (unless (and (typed? t)
                      (equal? (syntax-e #'type) (typed-type t)))
           (raise-type-error))
         #'id)])))
```

```
(define-syntax-rule (define-thing id .....)
  (define-syntax id (typed #'gen-id "thing"))
  (define gen-id (thing .....))
  .....)

(define-syntax-rule (define-place id desc (thng ...) .....)
  ..... (place desc (list (check-type thng "thing") ...) .....)
  .....)
```

check type of *thng*

Version 4: Types

```
(begin-for-syntax
 (struct typed (id t ....))

(define-syntax check-type
  (lambda (stx)
    (syntax-case stx ()
      [(check-type id type)
       (let ([t (syntax-local-value #'id)])
         (unless (and (typed? t)
                      (equal? (syntax-e #'type) (typed-type t)))
           (raise-type-error))
         #'id)])))
```

bind to a compile-time function (i.e., macro)

```
(define-syntax-rule (define-thing id .....)
  (define-syntax id (typed #'gen-id "thing"))
  (define gen-id (thing .....))
  .....)

(define-syntax-rule (define-place id desc (thng ...) .....)
  ..... (place desc (list (check-type thng "thing") ...) .....)
  .....)
```

Version 4: Types

```
(begin-for-syntax
 (struct typed (id type) #:property prop:procedure .....))

(define-syntax check-type
  (lambda (stx)
    (syntax-case stx ()
      [(check-type id type)
       (let ([t (syntax-local-value #'id)])
         (unless (and (typed? t)
                      (equal? (syntax-e #'type) (typed-type t)))
           (raise-type-error))
         #'id)])))
```

lookup compile-time value

```
(define-syntax-rule (define-thing id .....)
  (define-syntax id (typed #'gen-id "thing"))
  (define gen-id (thing .....))
  .....)

(define-syntax-rule (define-place id desc (thng ...) .....)
  ..... (place desc (list (check-type thng "thing") ...) .....)
  .....)
```

Version 4: Types

```
(begin-for-syntax
 (struct typed (id type) #:property prop:procedure ....))
```

```
(define-syntax check-type
  (lambda (stx)
    (syntax-case stx ()
      [(check-type id type)
       (let ([t (syntax-local-value #'id)])
         (unless (and (typed? t)
                      (equal? (syntax-e #'type) (typed-type t)))
           (raise-type-error))
         #'id)])))
```

```
(define-syntax-rule (define-thing id .....)
  (define-syntax id (typed #'gen-id "thing"))
  (define gen-id (thing ....))
  .....)
```

= compile time

= run time

= bridge

```
(define-syntax-rule (define-place id desc (thng ...) .....)
  ..... (place desc (list (check-type thng "thing") ...) ....)
  .....)
```

Version 5: New Language

import character-level parser...

```
#lang reader "txtadv-reader.rkt"

===VERBS===
north, n
"go north"

south, s
"go south"

....

===THINGS===

---cactus---
get
 "Ouch!"
....
```

world.rkt

Version 5: New Language

parses into a module that imports **`txtadv.rkt`**

```
#lang reader "txtadv-reader.rkt"

===VERBS===
north, n
"go north"

south, s
"go south"

....

===THINGS===

---cactus---
get
 "Ouch!"
....
```

`world.rkt`

Version 5: New Language

```
#lang reader "txtadv-reader.rkt"

===VERBS===
north, n
"go north"

south, s
"go south"

....

===THINGS===

---cactus---
get
 "Ouch!"
....
```

world.rkt

```
#lang racket
(require syntax/readerr)
(provide read-syntax)

(define (read-syntax src in)
  ...
  (datum->syntax
   #f
   `(module world "txtadv.rkt"
      .....)))
```

txtadv-reader.rkt

Version 6: Environment Support

installed with **raco link**

```
#lang txtadv

===VERBS===
north, n
"go north"

south, s
"go south"

....

===THINGS===

---cactus---
get
 "Ouch!"
....
```

world.rkt

The Core Racket Grammar?

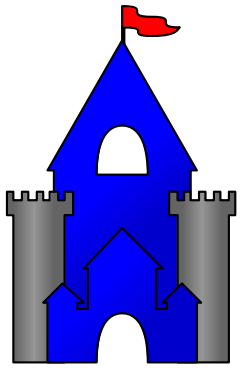
⟨module⟩ **`::=`** **`#lang`** ⟨module-name⟩ ⟨any⟩

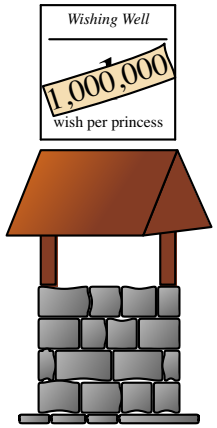
... plus a mapping from ⟨module-name⟩s to ⟨module⟩s

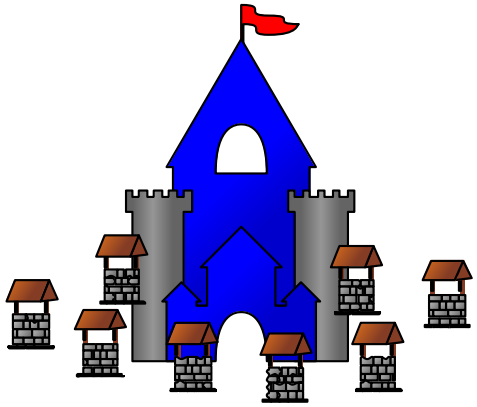
... plus one pre-defined ⟨module-name⟩

Racket is...

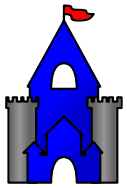
- ... a programming language
- ... a family of programming languages
- ... a set of programming tools
- ... a *way* of programming



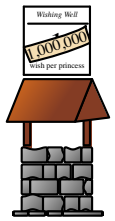




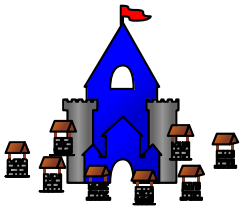
The Racket Way



Everything is a program



Concepts are programming language constructs



The programming language is extensible

racket-lang.org