Introduction to Racket

CS 442/642: Principles of Programming Languages

Instructor: Prabhakar Ragde

Marking scheme:
40% assns, 20% midterm, 40% final

(See course Web page for more details.)

Required textbook: Pierce, Types and Programming Languages

Initial readings: chapters 1 and 2.

Themes:

- Expressivity
- Meaning
- Guarantees
- Implementations


```
; Racket

; filter:
; (X -> boolean) (listof X) -> (listof X)

(define (filter p lst)
  (cond
    [(empty? lst) empty]
    [(p (first lst))
       (cons (first lst)
             (filter p (rest lst)))]
    [else (filter p (rest lst))]))

(filter (lambda (x) (> 3 x)) '(1 2 3 4))
```


Racket:

- Basics of functional programming
- Dynamic typing
- Functions as values
- Higher-order functions
- Macros and continuations
- Associated formalisms: lambda calculus, recursive function theory, combinators, semantics


```
(* OCaml *)

(* filter : ('a -> bool) -> 'a list -> 'a list *)
let rec filter p lst =
 match lst with
   [] -> []
 | (h :: t) ->
   if p h then h :: filter p t
          else filter p t
```

```
; Racket

(define (filter p lst)
  (match lst
    ['() '()]
    [(cons h t)
       (if (p h)
           (cons h (filter p t))
           (filter p t))]))
```


OCaml:

- Static typing
- Type inference
- Algebraic data types
- Parametric polymorphism
- Associated formalisms: typed lambda calculus, type judgments, progress and preservation


```
-- Haskell

filter :: (a -> Boolean) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
 | p x       = x : filter p xs
 | otherwise = filter p xs
```


Haskell:

- Purity
- Laziness
- Monads
- Type classes
- Associated formalisms: higher-order polymorphism

Racket

Review tutorial:
Teach Yourself Racket
`www.cs.uwaterloo.ca/~plragde/tyr`

Some historical milestones:

λ-calculus (Church 1936)

Recursive function theory (Kleene 1940)

LISP (McCarthy 1958)

Scheme (Steele and Sussman, 1975)

Standards and revisions (1978, 1985, 1986, 1990, 1998, 2007)

PLT Scheme (1995 – 2010)

Racket (2010 – present)

A minimal subset of Racket

Constant definition
Lambda abstraction
Function application

```
(define life 42)
(define add-life
  (lambda (x) (+ x life)))
(add-life 4)
```


Conditional expressions
Lists
Structures

```
(struct point (x y))
(define (dist p1 p2)
  (sqrt
   (+
    (sqr (- (point-x p1)
            (point-x p2)))
    (sqr (- (point-y p1)
            (point-y p2))))))
```


Local definitions

```
(define (dist p1 p2)
  (define dx (- (point-x p1)
                (point-x p2)))
  (define dy (- (point-y p1)
                (point-y p2)))
  (sqrt (+ (sqr dx) (sqr dy))))
```

A Racket program is a sequence of definitions and expressions.

Conceptual model:
The expressions are reduced to values by substitution.


```
(+ (* 3 4) (- 6 5))
=> (+ 12 (- 6 5))
=> (+ 12 1)
=> 13
```


Some language features can be replaced by the use of others.

Syntactic sugar
“Desugaring”

```
(define (dist p1 p2)
  ((lambda (dx dy)
     (sqrt (+ (sqr dx) (sqr dy))))
    (- (point-x p1) (point-x p2))
    (- (point-y p1) (point-y p2))))
```



```
(define (dist p1 p2)
  (let ([dx (- (point-x p1)
               (point-x p2))]
        [dy (- (point-y p1)
               (point-y p2))])
    (sqrt (+ (sqr dx) (sqr dy)))))
```



```
; Implementation of let
;  using a macro

(define-syntax-rule
 (let ([x e] ...) body)
    ((lambda (x ...) body) e ...))
```


Homoiconicity

Core language can easily be extended by programmers

How minimal is minimal?

A **truly** minimal subset of Racket:

Lambda abstraction
Function application

The lambda calculus

Reading:
Pierce, chapter 5
(chapter 3 as needed)

Origin:

Alonzo Church's successful attempt (1936) to show that there is no procedure for deciding statements in first-order logic.

Church's work was the first uncomputability result, a few months prior to Alan Turing's work.

Why study the lambda calculus today?

Can formalize reduction by substitution.

Can formalize scope.

Can prove properties of the system.

Many proofs generalize when we put back or add features.

Abstract syntax:

```
t ::= x
      λ x . t
      t t
```


Concrete syntax:

$$
\begin{array}{lcl}
\langle expr \rangle & ::= & \langle var \rangle \\
 & | & \langle abs \rangle \\
 & | & \langle app \rangle \\
\langle abs \rangle & ::= & (\lambda \ \langle var \rangle \ . \ \langle expr \rangle \ ) \\
\langle app \rangle & ::= & (\ \langle expr \rangle \ \langle expr \rangle \ )
\end{array}
$$

We use **term** and **expression** interchangeably.

In $\lambda x.y$, *x* is the **variable**, *y* the **body**.

In *x y*, *x* is the **rator** and *y* the **rand**.

We parenthesized the rules for $\langle abs \rangle$ and $\langle app \rangle$ to eliminate ambiguity.

We avoid excessive parentheses with some conventions.

Function application is left-associative: $x\ y\ z$ means $((x\ y)\ z)$.

Abstractions extend as far right as possible: $\lambda x.yz$ means $\lambda x.(yz)$ and not $(\lambda x.y)z$.

A function of n variables ($n > 1$) will be represented by a function of one variable that produces a function of $n - 1$ variables.

Such functions are called **curried** (after Haskell Curry).

Some texts permit $\lambda xyz.w$ to represent $\lambda x.\lambda y.\lambda z.w$.

A **denotational semantics** for the lambda calculus would associate a mathematical object with each expression.

Our intuitive interpretation of expressions as functions turns out to be difficult to formalize.

Instead, we define an **operational semantics**, where expressions are acted on by an abstract machine.

In fact, we use a **reduction semantics**, which is a formalization of the substitution model from CS 115/135/145.

Free and bound variables

In an abstraction $\lambda x.t$, we see a *binding occurrence* of x.
There may be one or more *bound occurrences* of x in t.

There may also be free variables: y is free in $\lambda x.y$.

We can recursively define the set of bound and free variables of an expression, using the recursive definition of an expression.

$$\begin{aligned} BV[x] &= \phi \\ BV[\lambda x.t] &= BV[t] \cup \{x\} \\ BV[t_1 t_2] &= BV[t_1] \cup BV[t_2] \end{aligned}$$

$$\begin{aligned} FV[x] &= \{x\} \\ FV[\lambda x.t] &= FV[t] \setminus \{x\} \\ FV[t_1\ t_2] &= FV[t_1] \cup FV[t_2] \end{aligned}$$

Note that a variable may be both free and bound in an expression: $x \lambda x.x$.

But a single occurrence of a variable is either free or bound, not both.

Intuitively, the name in a binding occurrence does not matter: $\lambda x.x$ and $\lambda y.y$ are the same.

We formalize this notion as α-equivalence.

For a term t and variables x, y, we say $\lambda x.t =_\alpha \lambda y.[x \mapsto y]t$ if $y \notin FV[t]$.

Here $[x \mapsto y]t$ means t with y substituted for x. We call this an α-conversion. (In some texts: $t[y/x]$.)

We are working with an intuitive definition of substitution for now, to be made precise shortly.

We want to extend α-equivalence to rewriting of **subterms**, so that $z\lambda x.x =_\alpha z\lambda y.y$.

Pierce uses **inference rules** to accomplish such extensions.

$$\frac{t_1 =_\alpha t_1'}{t_1 t_2 =_\alpha t_1' t_2}$$

$$\frac{t_2 =_\alpha t_2'}{t_1 t_2 =_\alpha t_1 t_2'}$$

$$\frac{t_1 =_\alpha t_1'}{\lambda x.t_1 =_\alpha \lambda x.t_1'}$$

Inference rules come from logic, where they are used to formalize proofs (we will see an example of this in the next lecture module).

The proof of a statement forms a tree (branching if a rule has more than one premise above the line).

$$\frac{\lambda x.x =_\alpha \lambda y.y}{z \lambda x.x =_\alpha z \lambda y.y}$$

Substitution is at the heart of the semantics of the lambda calculus.

Intuitively, we want $(\lambda x.zx)y \to zy$, because y should be substituted for x in the abstraction.

We call this β-reduction, denoted $\to_\beta$.

$\to_\beta$ is a **relation** on terms, because several substitutions may be possible.

$(\lambda x.(\lambda y.x)z)w \to_\beta (\lambda y.w)z$

$(\lambda x.(\lambda y.x)z)w \to_\beta (\lambda x.x)w$

We can define the **transitive closure** $\rightarrow_\beta^*$ as follows:

$t_1 \rightarrow_\beta^* t_2$ iff $t_1 = t_2$, or there exists t_3 such that $t_1 \rightarrow_\beta t_3$ and $t_3 \rightarrow_\beta^* t_2$.

Formal definition of $\rightarrow_\beta$.

For terms t_1, t_2, and a variable x, $(\lambda x.t_1)t_2 \rightarrow_\beta [x \mapsto t_2]\ t_1$.

Once again, we extend this to rewriting of subterms.

A β-reducible expression (β-redex, or just **redex**) is a subterm of the form $(\lambda x.t_1)t_2$, which we rewrite as $[x \mapsto t_2]\ t_1$ in one step of β-reduction.

An expression is in β-normal form (or just **normal form**) if it contains no redices.

Computation in the lambda-calculus consists of reducing an expression to normal form.

But we still need a proper definition of substitution.

Pierce gives several intuitive but incorrect definitions of substitution, followed by the correct definition.

The same problem arises in predicate logic (CS 245).

Both incorrect and correct definitions are recursive, following the recursive definition of a term.

First try (incorrect)

$$
\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \\
[x \mapsto s](\lambda y.t) &= \lambda y.([x \mapsto s]t) \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)([x \mapsto s]t_2)
\end{aligned}
$$

Problem:

$[x \mapsto y]\lambda x.x = \lambda x.y$

We should not substitute for bound variables.

Second try (incorrect)

$$
\begin{aligned}
[x \mapsto s]x &= E \\
[x \mapsto s]y &= y \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)([x \mapsto s]t_2) \\
[x \mapsto s](\lambda x.t) &= \lambda x.t \\
[x \mapsto s](\lambda y.t) &= \lambda y.([x \mapsto s]t)
\end{aligned}
$$

Problem: inadvertent capture of free variables.

$(\lambda x.\lambda y.x)yw \to_\beta (\lambda y.y)w \to_\beta w$

$(\lambda x.\lambda z.x)yw \to_\beta (\lambda z.y)w \to_\beta y$

It shouldn't matter whether we use $\lambda y.x$ or $\lambda z.x$ in that inner abstraction.

Solution:

α-convert an abstraction being substituted into if its variable is free in the substituted term.

$$\begin{aligned}
[x \mapsto s]x &= s \\
[x \mapsto s]y &= y \\
[x \mapsto s](t_1\ t_2) &= ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) \\
[x \mapsto s](\lambda x.t) &= \lambda x.t \\
[x \mapsto s](\lambda y.t) &= \lambda y.([x \mapsto s]t), y \notin FV[s] \\
[x \mapsto s](\lambda y.t) &= \lambda z.([x \mapsto s][y \mapsto z]t), \\
& \quad y \in FV[s] \\
& \quad z \text{ a fresh variable}
\end{aligned}$$

Confluence

Not every term has a normal form.

Consider $(\lambda x.xx)(\lambda x.xx)$.

One step of β-reduction recreates it.

We say that $\to_\beta^*$ **diverges** in this case.

Can a term have more than one normal form?

No, as a consequence of the following confluence theorem.

The Church-Rosser theorem

Thm: If E_1, E_2, E_3 are terms such that $E_1 \to_\beta^* E_2$ and $E_1 \to_\beta^* E_3$, then there exists E_4 such that $E_2 \to_\beta^* E_4$ and $E_3 \to_\beta^* E_4$.

$$
\begin{tikzcd}
 & E_1 \arrow[dl, "\beta^*"'] \arrow[dr, "\beta^*"] & \\
E_2 \arrow[dr, dashed, "\beta^*"'] & & E_3 \arrow[dl, dashed, "\beta^*"] \\
 & E_4 &
\end{tikzcd}
$$

Idea of proof:
Suppose we could prove this for one step of β-reduction.

$$
\begin{tikzcd}
 & E_1 \arrow[dl, "\beta"'] \arrow[dr, "\beta"] & \\
E_2 \arrow[dr, "\beta"'] & & E_3 \arrow[dl, "\beta"] \\
 & E_4 &
\end{tikzcd}
$$

Then we could “fill in the parallelogram”.

(Formally: double induction on lengths of sides.)

Problem:
The diamond property does not hold.

$$(\lambda x.x\ x\ x)((\lambda y.y)(\lambda z.z))$$

We can define a restricted “parallel” version of β-reduction for which the diamond property holds.

We can also prove that the transitive closure of the parallel reduction is the same as the transitive closure of β-reduction.

Neither the statement nor the proof of the confluence theorem indicates how a normal form can be found.

Full β-reduction, as we have seen, is not deterministic.

Thm: (Curry/Feys) Repeatedly reducing the leftmost, outermost redex will find a normal form, if it exists.
Normal Order Reduction (NOR) is the name given to this evaluation strategy, in which $\rightarrow_\beta$ becomes a partial function.

NOR is not used in programming languages, because it requires rewriting of lambda abstractions.

Call by name reduces the leftmost, outermost redex, but not within the body of abstractions.

Call by name was used as early as Algol 60 (1960), and an optimized version, **call-by-need**, is used in Haskell.

Inference rules for these are somewhat messy (Exercise 5.3.6).

Call by value reduces the leftmost, innermost redex, but not within the body of abstractions.

Call by value is used in Racket, OCaml, and most imperative programming languages (also throughout Pierce).

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}$$

$$\frac{t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'}$$

We'll now see how to simulate familiar programming constructs in the lambda calculus.

Although there is no notion of `define`, we can for our convenience define abbreviations, with the understanding that simple textual substitution is meant.

id $= \lambda x.x$

Booleans

true $= \lambda x.\lambda y.x$
false $= \lambda x.\lambda y.y$

A conditional test then applies the tested value to the two alternatives.

if B **then** T **else** $F = B\ T\ F$

Given **if**, we can easily define **and**, **or**, **not**.

Note: NOR assumed here (to avoid evaluating both alternatives).

Lists

cons consumes two arguments, f and r, and produces a function consuming a message.

$$
\begin{aligned}
\textbf{cons} &= \lambda f.\lambda r.\lambda m.\ \textbf{if}\ m\ \textbf{then}\ f\ \textbf{else}\ r \\
&= \lambda f.\lambda r.\lambda m.\ m\ f\ r \\
\textbf{first} &= \lambda p.p\ \textbf{true} \\
\textbf{rest} &= \lambda p.p\ \textbf{false}
\end{aligned}
$$

empty? must produce **false** when given a **cons**.

empty must produce **true** when **empty**? is applied.

$$
\begin{aligned}
\textbf{cons} &= \lambda f.\lambda r.\lambda m.\ m\ f\ r \\
\textbf{empty}? &= \lambda p.\ p\ \lambda f.\lambda r.\ \textbf{false} \\
\textbf{empty} &= \lambda m.\ \textbf{true}
\end{aligned}
$$

Natural numbers

Idea 1: "n" is a list of length n.

Benefits: successor, predecessor, zero test are easy.

Drawbacks: other arithmetic functions (e.g. addition) require a general recursion mechanism.

Idea 2 (Pierce): Church numerals.

n is represented by a function $\mathbf{c_n}$ that consumes two arguments, a successor s and a zero z.

$$
\begin{aligned}
\mathbf{c_0} &= \lambda s.\lambda z.\ z \\
\mathbf{c_1} &= \lambda s.\lambda z.\ s\ z \\
\mathbf{c_2} &= \lambda s.\lambda z.\ s\ (s\ z) \\
&\dots \\
\mathbf{succ} &= \lambda n.\lambda s.\lambda z.\ s\ (n\ s\ z) \\
\mathbf{plus} &= \lambda m.\lambda n.\lambda s.\lambda z.\ m\ s\ (n\ s\ z) \\
\mathbf{mult} &= \lambda m.\lambda n.\ m\ (\mathbf{plus}\ n)\ \mathbf{c_0}
\end{aligned}
$$

pred is tricky (see Pierce) but then **sub** follows.

expt has a very slick definition.

General recursion

Suppose we try to compute factorials.

$$\textbf{fact} = \lambda n.\ \textbf{if}\ (\textbf{iszero}\ n)\ \textbf{then}\ 1\ \textbf{else mult}\ n\ (?\ (\textbf{pred}\ n))$$

What do we fill in for the question mark?

We make the question mark into a variable.

$$\textbf{pfact} = \lambda r.\lambda n.\ \textbf{if}\ (\textbf{iszero}\ n)\ \textbf{then}\ 1\ \textbf{else mult}\ n\ (\ r\ (\textbf{pred}\ n))$$

We need **fact** such that **fact** $=$ **pfact fact**.

fact is a *fixed point* of **pfact**.

Suppose **fact** $=$ **Y pfact**.

Then **Y pfact** $=$ **pfact** (**Y pfact**).

In general, we want **Y** $f = f$ (**Y** f) for all f.

This is known as the **Y combinator**.

We can extract it from **pfact** with a bit of work.

$$\textbf{pfact} = \lambda r.\lambda n.\ \textbf{if}\ (\textbf{iszero}\ n)\ \textbf{then}\ 1\ \textbf{else mult}\ n\ (\ r\ (\textbf{pred}\ n))$$

Suppose we try **pfact pfact** as a candidate for **fact**.

This doesn't work, as the recursive call is wrong.

$\textbf{pfact}' = \lambda r.\lambda n.$ **if** ($\textbf{iszero}\ n$) **then** 1
else
$\textbf{mult}\ n\ (\ (r\ r)\ (\textbf{pred}\ n))$

Now $\textbf{fact} = \textbf{pfact}'\ \textbf{pfact}'$.

But we want **Y** to work on **pfact**.

$\textbf{pfact}' = \lambda r.(\lambda g.\lambda n.$
if ($\textbf{iszero}\ n$)
then 1 **else**
$\textbf{mult}\ n\ (\ g\ (\textbf{pred}\ n)))$
$(r\ r)$

And there's **pfact** again.

$\textbf{pfact}' = \lambda r.\ \textbf{pfact}\ (r\ r)$

But the answer is $\textbf{pfact}'\ \textbf{pfact}'$, and we want **Y** to work when given a general **pfact**-like f.

$\textbf{Y} = \lambda f.((\lambda r.f\ (r\ r))(\lambda r.f\ (r\ r)))$

This works in NOR, but not in call-by-value (why?).

$\textbf{Y}_\textbf{v} =$
$\lambda f.((\lambda r.f\ (\lambda y.r\ r\ y))(\lambda r.f\ (\lambda y.r\ r\ y)))$

Replacing f with $\lambda x.f\ x$ is known as η-expansion.

Its inverse, η-reduction, is used with NOR to ensure certain desirable properties.

η-expansion is used in call-by-value to avoid or delay certain reductions.

de Bruijn indices

de Bruijn (1972), in implementing a proof checker, used a nameless representation for terms, where a bound occurrence of a name is replaced by its lexical depth relative to its binding occurrence.

$\lambda x.x$ becomes $\lambda.0$.

$\lambda x.\lambda y.y$ becomes $\lambda.\lambda.0$.

$\lambda x.\lambda y.x$ becomes $\lambda.\lambda.1$.

Free variables are bound with a "naming context" (Pierce) or left unconverted (other texts).

Using deBruijn indices, α-conversion is unnecessary.

Substitution does not require fresh variables, but some indices need to be "shifted". (Details in Pierce, in particular, exercise 6.2.7.)

deBruijn indices are a convenient internal representation in implementations, but are not very human-readable.

Combinators

Another approach to simplifying the mechanics of the lambda calculus does away with λ entirely, by using only sets of combinators.

This approach was pioneered by Haskell Curry in the 1940's and 1950's (and played a role in the development of Haskell and its compilation strategies).

We will take a brief look at a simple combinator system, called the **SKI calculus**.

$$
\begin{array}{rcl}
\langle expr \rangle & ::= & \langle var \rangle \\
& | & \mathbf{S} \\
& | & \mathbf{K} \\
& | & \mathbf{I} \\
& | & \langle expr \rangle \ \langle expr \rangle
\end{array}
$$

We use the same conventions for parentheses as the lambda calculus.

Intuitively, the letters represent the following:

$$\begin{aligned} \mathbf{I} &= \lambda x.x \\ \mathbf{K} &= \lambda x.\lambda y.x \\ \mathbf{S} &= \lambda x.\lambda y.\lambda z.xz(yz) \end{aligned}$$

Formally, we use the following reduction rules (here X, Y, Z stand for terms):

$$\begin{aligned} \mathbf{I}X &\to X \\ \mathbf{K}XY &\to X \\ \mathbf{S}XYZ &\to XZ(YZ) \end{aligned}$$

For example: $\mathbf{S}(\mathbf{SK})xy \to \mathbf{SK}y(xy) \to \mathbf{K}(xy)(y(xy)) \to xy$

Substitution is a lot simpler, since all occurrences of variables are free.

$$\begin{aligned} [x \mapsto E]x &= E \\ [x \mapsto E]y &= y \\ [x \mapsto E](MN) &= ([x \mapsto E]M)([x \mapsto E]N) \end{aligned}$$

We can prove the SKI calculus equivalent to the lambda calculus by simulating abstraction.

We define $[x].M$ so that $([x].M)N \to^* [x \mapsto N]M$.

$$\begin{aligned} [x].E &= \mathbf{K}E \text{ if } x \text{ does not occur in } E; \\ [x].x &= \mathbf{I} \\ [x].E_1E_2 &= \mathbf{S}([x].E_1)([x].E_2) \end{aligned}$$

Enriching the lambda calculus

Pierce discusses adding Booleans and numbers as primitives, resulting in λ**NB**.

Landin (1970) defines **ISWIM**, a family of languages.

An ISWIM language extends the lambda calculus with a set of constants, a set of primitive functions, and a δ-function or δ-rules that describe how the primitive functions apply to constants.

Lisp, Scheme, and Racket are ISWIM languages with a lot of syntactic sugar.

Landin observed that ISWIM could be used to define the semantics of programming languages.

He defined an abstract machine (SECD) as an evaluator for ISWIM, an early result in what is now called operational semantics.

Denotational semantics (modelling computation using mathematical logic) arose from the work of Dana Scott and Christopher Strachey in the 1960's.

Another computational model that influenced Lisp (and thus Scheme and Racket) was that of partial recursive functions.

Recursive function theory

We define the set of **primitive recursive** functions consuming and producing natural numbers.

The base set of primitive recursive functions is:

- The constant function 0;
- The successor or "plus one" function S;
- The projection functions Π^n_k, where $\Pi^n_k(m_1, \ldots, m_n) = m_k$.

We build more by composition and recursion.

Composition:

If $\psi, \chi_1, \ldots, \chi_p$ are primitive recursive, then so is $\phi(m_1, \ldots, m_n) = \psi(\chi_1(m_1, \ldots, m_n), \ldots, \chi_p(m_1, \ldots, m_n))$.

Recursion:

If ψ, χ are primitive recursive, then so is ϕ defined as:

$$\phi(0, m_1, \ldots, m_n) = \psi(m_1, \ldots, m_n)$$

$$\phi(k+1, m_1, \ldots, m_n) = \chi(k, \phi(k, m_1, \ldots, m_n), m_1, \ldots, m_n)$$

Addition and subtraction:

$$
\begin{aligned}
+(0, n) &= \Pi^1_1(n) \\
+(k+1, n) &= S(\Pi^3_2(k, +(k, n), n)) \\
P(0, n) &= 0 \\
P(k+1, n) &= \Pi^2_1(k, P(k)) \\
-(0, n) &= \Pi^1_1(n) \\
-(k+1, n) &= P(\Pi^3_2(k, -(k, n), n))
\end{aligned}
$$

$-(m, n)$ computes $\max\{0, n-m\}$.

All primitive recursive functions are total.

We can show that there are total functions with natural recursive definitions that are not primitive recursive.

Ackermann's function
(form due to Rosza Peter):

$$
\begin{aligned}
A(0, n) &= n + 1 \\
A(m, 0) &= A(m-1, 1) \\
A(m, n) &= A(m-1, A(m, n-1))
\end{aligned}
$$

The partial recursive functions

A function ϕ is **partial recursive** if there exists a primitive recursive χ such that $\phi(m_1, \ldots, m_k)$ is the least k such that $\chi(m_1, \ldots, m_n, k) = 0$, or undefined if no such k exists.

The partial recursive functions are equal to the functions expressible in the lambda calculus, which are equal to the functions computable on a Turing machine.

Mutation in Racket

“Mutation” refers to changing the value bound to a name, or stored in a data structure.

It is used sparingly in Scheme and Racket, slightly more in Lisp and OCaml, and, as we will see, not at all in Haskell.

Why not mutation?

In the absence of mutation, programs exhibit **referential transparency** (an expression may be replaced with one of equal value).

This makes it much easier to reason about, transform, and optimize programs.

But mutation is occasionally useful for expressivity or efficiency.

Warning:

Do not use mutation for assignment questions unless it is explicitly permitted.

Elementary mutation (rebinding)

```
(define x 3)
(set! x 4)
x
=> 4
```

`set!` produces the special value `#<void>`.

Its **side effect** is more important.


```
(define counter 0)
(define (count)
  (begin
    (set! counter (add1 counter))
    counter))
```

There is an implicit `begin` at the beginning of each function body and `cond` answer.

Elementary mutation can be added to ISWIM and modelled using techniques similar to those discussed here for the lambda calculus.

Semantics Engineering with PLT Redex by Felleisen et al. contains a complete treatment of the subject.

Intermediate mutation: boxes

A box is a Racket value that contains a Racket value.

`(box v)` creates a box containing value `v`.

`(unbox b)` produces the value in box `b`.

`(set-box! b newv)` replaces the old value in box `b` with the new value `newv`.

Boxes are provided as primitives in Racket.

We can simulate boxes using elementary mutation.

We can simulate mutable structures and lists (also provided in Racket) using boxes.

These simulations mean that more advanced forms of mutation can be viewed as syntactic sugar over elementary mutation.


```
(define (make-counter) (box 0))
(define (count ctr)
  (begin
    (set-box! ctr (add1 (unbox ctr)))
    (unbox ctr)))

(define ctr1 (make-counter))
(define ctr2 (make-counter))
(count ctr1) => 1
(count ctr1) => 2
(count ctr2) => 1
```

```
(define (make-box v)
  (local [(define val v)]
    (lambda (msg)
      (cond
        [(symbol=? msg 'unbox) val]
        [(symbol=? msg 'set-box)
           (lambda (newv) (set! val newv))]))))

(define (my-set-box! b v) ((b 'set-box) v))
(define (my-unbox b) (b 'unbox))
```

This can be generalized to a complete object/class system.

Lists in Racket are immutable, in contrast to Scheme and Lisp. Racket provides a separate mutable list type.

Structures in Racket can be made mutable by adding the `#:mutable` keyword to the `struct` definition.

Racket also has mutable arrays (called vectors), hash tables, and iteration comprehensions.

Racket has a rich set of I/O primitives.

The REPL (Interactions window) lets us mostly avoid them.

It is sometimes useful to instrument code, during development, to print out intermediate values.

```
(printf "Value so far: ~a\n" x)
```

This produces `#<void>` but has the side effect of printing a line as soon as it is evaluated.

Some other important features of Racket (pattern matching, modules, macros, continuations) will be discussed later.

DrRacket's Help Desk and the references on the Web page are a useful source of further information.