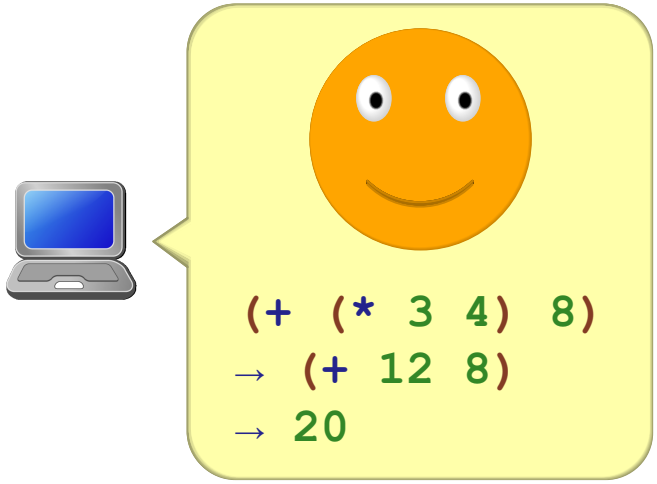
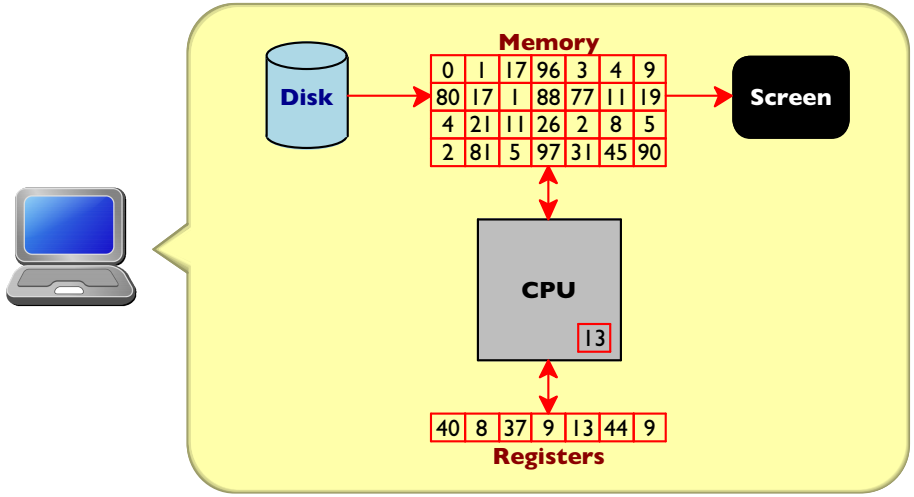
Models of Computation

Racket



minimacket6.zip

C



Mixing Languages

Call C functions from Racket?

- You can't write everything from scratch; real programs use libraries
- *Many* libraries are written in C

... or they provide a C interface

Calling C from Racket

C is the least-common denominator

By "C," we really mean the machine model and conventions for how functions are called

To mix Racket and C, you have to think in terms of how Racket is implemented

It's the same for mixing any language with C

Libraries

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv) {
  printf("%f", log10(atof(argv[1])));


}
```

```
% gcc log.c -lc -lm
```

Libraries

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc, char **argv) {
  printf("%f", log10(atof(argv[1])));


}
```

```
% gcc log.c -lc -lm
% ldd a.out
linux-gate.so.1 =>  (0x00553000)
libc.so.6 => /lib/libc.so.6 (0x00a22000)
libm.so.6 => /lib/libm.so.6 (0x00210000)
/lib/ld-linux.so.2 (0x00f17000)
```

Dynamic Libraries

Programs normally link to ***dynamic libraries***

also known as ***shared libraries***, ***dynamically linked libraries***, ***DLLs***, or ***shared objects***

Common shared-library file existensions are `.so`, `.dll`, and `.dylib`

Roughly, a shared library provides

- machine code, and
- a mapping from C names to addresses

Loading Dynamic Libraries in Racket

```
(require ffi/unsafe)

; Load a library:
(define math-lib (ffi-lib "libm"))

; Get the address of the log10() function:
(get-ffi-obj "log10" math-lib _pointer)
```

Calling Dynamic Libraries in Racket

```
(require ffi/unsafe)
(define math-lib (ffi-lib "libm"))

(define log10
  (get-ffi-obj "log10"
               math-lib
               (_fun _double -> _double)))
(log10 100.0)
```

Calling Dynamic Libraries in Racket

```
(require ffi/unsafe)
(define math-lib (ffi-lib "libm"))

(require ffi/unsafe/define)
(define-ffi-definer define-math math-lib)

(define-math log10 (_fun _double -> _double))
(log10 100.0)
```

libncurses

Taking control of a terminal:

- **man ncurses**
- **man initscr**

```
WINDOW *initscr(void);
int endwin(void);

int waddstr(WINDOW *win, const char *str);

int wrefresh(WINDOW *win);
```

libncurses

```
(define-ffi-definer define-curses (ffi-lib "libncurses"))

(define _WINDOW-pointer (_cpointer 'WINDOW))

(define-curses initscr (_fun -> _WINDOW-pointer))
(define-curses endwin (_fun -> _int))

(define-curses waddstr (_fun _WINDOW-pointer _string -> _int))

(define-curses wrefresh (_fun _WINDOW-pointer -> _int))
```

libncurses

Getting a password:

```
int wgetch(WINDOW *win);

int cbreak(void);
int noecho(void);

int wmove(WINDOW *win, int y, int x);
int getcurx(WINDOW *win);
int getcury(WINDOW *win);
```

libncurses

```
(define-curses wgetch (_fun _WINDOW-pointer -> _int))

(define-curses noecho (_fun -> _int))
(define-curses cbreak (_fun -> _int))

(define-curses wmove (_fun _WINDOW-pointer _int _int -> _int))
(define-curses getcurx (_fun _WINDOW-pointer -> _int))
(define-curses getcury (_fun _WINDOW-pointer -> _int))
```

libncurses

Mouse input:

```
typedef unsigned long mmask_t;
mmask_t mousemask(mmask_t newmask,
                  mmask_t *oldmask);
#define BUTTON1_CLICKED 0004

#define KEY_MOUSE 0631
int keypad(WINDOW *win, bool bf);

typedef struct {
  short id;
  int x, y, z;
  mmask_t bstate;
} MEVENT;
int getmouse(MEVENT *event);
```

libncurses

```
(define _mmask_t _ulong)
(define-curses mousemask (_fun _mmask_t _pointer -> _mmask_t))
(define BUTTON1_CLICKED 4)

(define KEY_MOUSE #o631)
(define-curses keypad (_fun _WINDOW-pointer _bool -> _int))

(define-cstruct _MEVENT ([id _short]
                         [x _int]
                         [y _int]
                         [z _int]
                         [bstate _mmask_t]))
(define-curses getmouse (_fun _MEVENT-pointer -> _int))
```

libmenu

libmenu builds on **libncurses**:

```
ITEM *new_item(const char *name,
               const char *desc);
MENU *new_menu(ITEM **items);

int set_menu_win(MENU *menu, WINDOW *win);
int post_menu(MENU *menu);
```

libmenu

```
(define-ffi-definer define-menu (ffi-lib "libmenu"))

(define _ITEM-pointer (_cpointer 'ITEM))
(define _MENU-pointer (_cpointer 'MENU))

(define-menu new_item (_fun _bytes _bytes -> _ITEM-pointer))
(define-menu new_menu (_fun (_list i (_or-null _ITEM-pointer))
                            -> _MENU-pointer))

(define-menu set_menu_win (_fun _MENU-pointer _WINDOW-pointer
                                -> _int))
(define-menu post_menu (_fun _MENU-pointer -> _int))
```

libmenu

```
int menu_driver(MENU *menu, int c);

#define REQ_LEFT_ITEM  (KEY_MAX + 1)
#define REQ_RIGHT_ITEM (KEY_MAX + 2)
#define REQ_UP_ITEM    (KEY_MAX + 3)
#define REQ_DOWN_ITEM  (KEY_MAX + 4)
...

ITEM *current_item(const MENU *menu);
```

libmenu

```
(define-menu menu_driver (_fun _MENU-pointer _int -> _int))

(define REQ_LEFT_ITEM (+ KEY_MAX 1))
(define REQ_RIGHT_ITEM (+ KEY_MAX 2))
....

(define-menu current_item (_fun _MENU-pointer -> _ITEM-pointer))
```

Strings, Byte Strings, and Allocation

For

```
ITEM *new_item(const char *name,
               const char *desc);
```

use

```
(define-menu new_item
  (_fun _bytes _bytes -> _ITEM-pointer))
```

or

```
(define-menu new_item
  (_fun _string _string -> _ITEM-pointer))
```

?

Strings, Byte Strings, and Allocation

With

```
(define-menu new_item
  (_fun _string _string -> _ITEM-pointer))
```

then

```
(define items
  (for/list ([s (in-list '("First"
                           "Second"
                           "Third"
                           "Fourth"))])
    (define i (new_item s "item"))
    (collect-garbage)
    i))
```

probably crashes...

Strings, Byte Strings, and Allocation

From the **`new_item`** documentation (emphasis added):

> The function **`new_item`** allocates a new item and initializes it from the name and description pointers. Please notice that the item stores only the pointers to the name and description. *Those pointers must be valid during the lifetime of the item.*

This requirement does not mesh with a copying garbage collector

Strings, Byte Strings, and Allocation

`(malloc .... 'raw)` allocates using C's **`malloc`**:

```
(define (copy s)
  (define bstr (string->bytes/utf-8 s))
  (define new (malloc (add1 (bytes-length bstr))
                      'raw))
  (memcpy new bstr (add1 (bytes-length bstr)))
  new)
```

Strings, Byte Strings, and Allocation

Build the copying protocol into a “C type”:

```
(define _copied-string
  (make-ctype
   _pointer
   ; Racket to C:
   (lambda (s)
     (define bstr (string->bytes/utf-8 s))
     (define new (malloc (add1 (bytes-length bstr))
                         'raw))
     (memcpy new bstr (add1 (bytes-length bstr)))
     ; s is live ⇒ new is live
     (register-finalizer s (lambda (s) (free new)))
     new)
   ; C to Racket:
   (lambda (p)
     (cast p _pointer _string))))
```

then use it for **`new_item`**...

Strings, Byte Strings, and Allocation

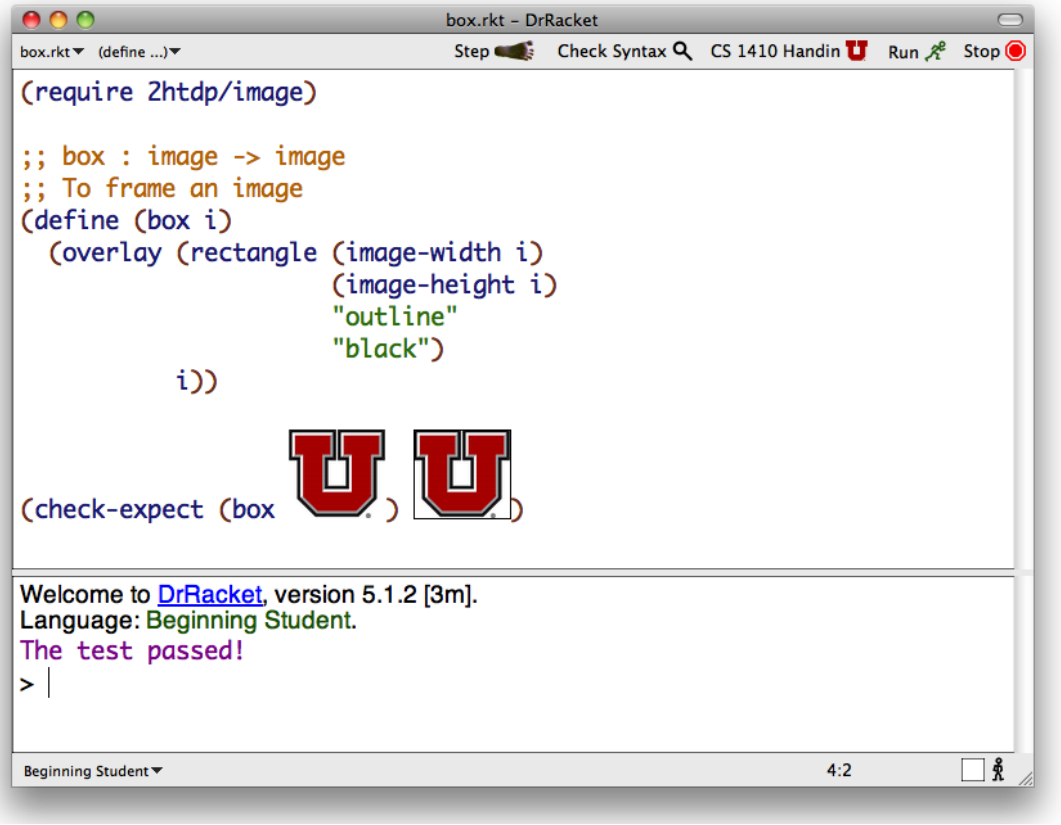
First cut:

```
(define-menu new_item
  (_fun _copied-string _copied-string -> _ITEM-pointer))
```

Even better, connect string lifetimes to the item:

```
(define-menu new_item
  (_fun (s1 : _copied-string) (s2 : _copied-string)
        -> (i : _ITEM-pointer)
        ; Extra arrow makes a wrapper:
        -> (begin
             ; i is live ⇒ s1 & s2 are live
             (register-finalizer i (lambda (i) s1))
             (register-finalizer i (lambda (i) s2))
             i)))
```

A Big Example



Uses **libssl**, **libglib**, **libcairo**, **libpango**, **libpng**, **libjpeg**, **Cocoa**, **libgtk**, **Win32**, ...

Foreign-Function Interface

A ***foreign-function interface*** must define

- translation rules for primitive datatypes

  What if a C function has a function argument?

- memory-management interaction

  When can a GC occur? Can the address of a value change?

- control, reentrancy, and concurrency

  What if a C function calls back to Racket?

Non-Racket examples: JNI (Java), Managed vs. Unmanaged .NET, Simplified Wrapper and Interface Generator (SWIG)