CSE 341
Lecture 22

Macros; extending Scheme's syntax

slides created by Marty Stepp
http://www.cs.washington.edu/341/

Macros

- **macro**: A rule or pattern for text substitution.
  - macros are *expanded* to convert one text string to another

- macro systems are found in programming languages
  - rules for rewriting programs
  - a pre-pass before evaluation

- macros have a bad reputation in the PL community
  - considered to be a "hack" in many languages

How are macros implemented?

- *Sublexical*: Replace `car` with `hd`, `cart` with `hdt`.
  - No macro system does this; so a macro-expander must know how to break programs into tokens.

- *Pre-parsing (token-based)*: Replace `add(x,y)` with `x + y` (where x and y stand for expressions)
  - can cause errors in complex expressions

- *Pre-binding*: Replacing `car` with `hd` would turn:
  - `(let* ((hd 0) (car 1)) hd)` into:
    `(let* ((hd 0) (hd 1)) hd)`

The C preprocessor

- **preprocessor** : Part of the C language's compilation process; modifies source code before it is compiled

function	description
`#include <`***`filename`***`>`	insert a library file's contents into this file
`#include "`***`filename`***`"`	insert a user file's contents into this file
`#define` ***`name [value]`***	create a preprocessor symbol ("variable")
`#if` ***`test`***	`if` statement
`#else`	`else` statement
`#elif` ***`test`***	`else if` statement
`#endif`	terminates an `if` or `if/else` statement
`#ifdef` ***`name`***	`if` statement; `true` if ***name*** is defined
`#ifndef` ***`name`***	`if` statement; `true` if ***name*** is *not* defined
`#undef` ***`name`***	deletes the given symbol name

Constants

```
#define NUM_STUDENTS  100
#define DAYS_PER_WEEK 7
...
double grades[NUM_STUDENTS];
int six_weeks = DAYS_PER_WEEK * 6;   // 42
printf("Course over in %d days", six_weeks);
```

- When preprocessor runs before compilation, 7 is literally inserted into the code wherever DAYS_PER_WEEK is seen
  - name DAYS_PER_WEEK does not exist in eventual program

```
int six_weeks = 7 * 6;   // 42
```

Debugging code

```
#define DEBUG
...
#ifdef DEBUG
    // debug-only code
    printf("Size of stack = %d\n", stack_size);
    printf("Top of stack  = %p\n", stack);
#endif
    stack = stack->next;    // normal code
```

- How is this different from a `bool`/`int` named DEBUG?

Preprocessor macros

- macros are like functions, but injected before compilation

```
#define SQUARED(x)   x * x
#define ODD(x)       x % 2 != 0

int a = 3;
int b = SQUARED(a);
if (ODD(b))
    printf("%d is an odd number.\n", b);
```

  - The above literally converts the code to the following:

```
int b = a * a;
if (b % 2 != 0) { ...
```

  - (C++ was originally implemented as a set of C macros.)

Subtleties of C macros

- preprocessor is dumb; it just replaces tokens

```
#define foo 42
int food = foo;          // int food = 42;
int foo = foo + foo;     // int 42 = 42 + 42;
```

- preprocessor can be used to do stupid/evil things

```
#define + -
#define 0 1
#define < >
```

Caution with macros

- since macros are injected directly, strange things can happen if you pass them complex values:

```
#define ODD(x)        x % 2 != 0
...
if (ODD(1 + 1)) {
    printf("It is odd.\n");    // prints!
}
```

  - The above literally converts the code to the following:

```
if (1 + 1 % 2 != 0) {
```

More macro bugs

```
#define SUMSQUARES(a, b)  a*a + b*b
...
// what goes wrong in the expressions below?
int a = 4, b = 3;
int c = SUMSQUARES(a, b);
int d = SUMSQUARES(a + 1, b - 1);
int e = d * SUMSQUARES(a, b);
int f = SUMSQUARES(a++, --b);

// int d = a + 1*a + 1 + b - 1*b - 1;
```

Correcting the macro

- the ODD macro is better written as:

```
#define ODD(x)        ((x) % 2 != 0)
...
if (ODD(1 + 1)) {
    printf("It is odd.\n");
}
```

  - Now the above literally converts the code to the following:

```
if (((1 + 1) % 2 != 0)) {
```

  - *Always* surround macro parameters in parentheses.
    - *(And you thought Scheme had too many!)*

Hygienic macros

- The problem is that C's macro system is really just a hack bolted onto the language after the fact.

- **hygienic macro**: One whose evaluation has predictable and expected results, and whose expansion is guaranteed not to cause collisions with existing symbol definitions.
  - Scheme features a powerful hygienic macro system.

Defining macros

```
(define-syntax name
    (syntax-rules (keywords)
        (pattern expr)))
```

- defines a new piece of syntax that uses the given keywords in the given pattern, and evaluates them to produce the given expression

- macros can be used to delay/avoid evaluation

Macro example

```
(define-syntax if2
  (syntax-rules ()
    ((if2 test e1 e2)
     (cond (test e1)
           (else e2)))))
```

- The above macro defines a new expression `if2` that behaves like Scheme's `if` expression
  - `if2` is implemented in terms of `cond`
  - as with the real `if`, `if2` evaluates only one of `e1`/`e2`
    - (if we had written `if2` as a procedure, this would not be so)

Macro example

```
(define-syntax if3
  (syntax-rules (then else)
    ((if3 e1 then e2 else e3)
      (if e1 e2 e3))))
```

- The above macro defines a new expression `if3` that adds new keywords `then` and `else`
  - Example:

    ```
    (if3 (< 2 3) then 42 else (+ 5 9))
    ```

Macros with redundancy

```
; produces a number twice as big as x!
(define-syntax double
  (syntax-rules ()
    ((double x) (+ x x))))
```

- problem: redundant (evaluates x twice)
  - How can we improve it?

Macro with local variable

```
; produces a number twice as big as x!
(define-syntax double
  (syntax-rules ()
    ((double x)
       (let ((temp x))
            (+ temp temp)))))
```

- by capturing x's result as temp, it is evaluated just once

A silly variation

```
; produces a number twice as big as x!
(define-syntax double
  (syntax-rules ()
    ((double x)
       (let* ((temp2 0) (temp x))
             (+ temp temp temp2)))))
```

- consider the following version of `double`...
  - works the same way, but uses a new useless variable `zero`

A potential problem situation

```
> (define temp 17)
> (double temp)
```

- if Scheme didn't have hygienic macros, would become:

```
((let* ((temp 0) (temp2 temp))
          (+ temp temp temp2))))
```

  - It would equal 0 every time!

- Scheme macros carefully rename any local variables to avoid any chance of conflict with the surrounding code.

How hygienic macros work

- Internally, a hygienic macro system:
  - gives fresh names to local variables in macros on each use
  - binds free variables in macros where the macro is defined
- Without hygiene, macro programmers:
  - get very creative with local-variable names in macros
  - get creative with helper-function names too
  - avoid local variables, which cause unpredictable effects
- Hygiene is a big idea for macros, but sometimes is not what you want. (Sometimes you just want text replace!)

Macro exercises

- Define a macro named `neither` that accepts two boolean expressions and returns `#t` if both are *false*.
  - Example: `(neither (> 5 9) (= 1 4))` returns `#t`
  - *(Use short-circuit evaluation.)*
- Define a macro `let1` that is like `let` but defines only one symbol, and uses only one set of parentheses.
  - Example:

```
(define (x+y^2 x y)
  (let1 (sum (+ x y))
        (* sum sum)))
```

Macro exercise solutions

```
(define-syntax neither
  (syntax-rules ()
    ((neither expr1 expr2)
     (cond (expr1 #f)
           (expr2 #f)
           (else #t)))))

(define-syntax let1
  (syntax-rules ()
    ((let1 (name value) expr)
     (let ((name value)) expr))))
```

More macro exercises

- Define a macro `maybe` that takes an expression and has a 50/50 chance of evaluating it; else it does nothing.
  - Example: `(maybe (display "hello!"))`

- Solution:

```
(define-syntax maybe
  (syntax-rules () ((maybe expr)
     (if (< 0.5 (random)) expr '()))))
```

Macros for streams

- Streams are hard to use. Macros/helpers make it easier!

```
(define-syntax scons (syntax-rules ()
    ((scons x y) (cons x (delay y)))))
(define scar car)
(define (scdr stream) (force (cdr stream)))
```

- Example:

```
(define ones (scons 1 ones))
(define (ints-from n) (scons n (ints-from (+ n 1))))
(define nat-nums (ints-from 1))
```

Including files

```
(load "filename")
```

- includes the given file's code in your program

```
> (include "utility.scm")
```