CS342: Organization of Prog. Languages

Topic 6: Syntax Extension

- Syntax Extension via Macros
- `define-syntax` for transformation rules
- Example
- Syntax Extension via Macros: "..."
- Syntax Extension via Macros: hygiene
- Syntax Extension via Macros: Conclusion

Syntax Extension via Macros

- We have described a number of forms in Scheme as being "equivalent to" other forms.
- These can be expressed formally using syntax transformation rules called "macros"
- Programmers can introduce their own syntax extensions using macros
- Unlike, e.g. C, these are tree-to-tree transformations, so we do not worry about mis-grouping.

```
#define baad_times(a,b)  a * b
#define good_times(a,b)  ((a) * (b))

...
baad_times(1+2, 3+4)     /* Gives 11 */
good_times(1+2, 3+4)     /* Gives 21 */
...
```

`define-syntax` for transformation rules

- Transformations, such as we gave to write `do` loops as recursive functions, can be specified in Scheme using `define-syntax`.
- The basic form is:

```
(define-syntax id
   (syntax-rules (literals)
       (pattern1 construction1)
       (pattern2 construction2)
       ...  ))
```

- These define tree transformations.
- The patterns give names to the parts of the argument structures, and are used to place them in the result constructions.
- The *literals* give words that must be matched exactly.

Example

```
(define-syntax setf!
  (syntax-rules (car cdr string-ref vector-ref)
    ((_ (car p) e)           (set-car! p e))
    ((_ (cdr p) e)           (set-cdr! p e))
    ((_ (vector-ref v i) e) (vector-set! v i e))
    ((_ (string-ref s i) e) (string-set! s i e))
    ((_ a e)                 (set! a e)) ))
```

Syntax Extension via Macros: ...

- The special pattern "..." causes the *preceding* pattern to be reproduced zero or more times.
- *E.g.* Our `do` loop equivalence can be given as the following:

```
(define-syntax my-do (syntax-rules ()

   ( (_ ((var1 init1 step1) ...)
        (test fini1 ...)
        body1 ...)

     (letrec
        ((my-do-fn  (lambda (var1 ...)
           (if test
              (begin (if #f #f) fini1 ...)
              (begin
                 body1 ...
                 (my-do-fn step1 ...))))))
       (my-do-fn init1 ...)) ) ))
```

- This is only mildly simplified from the real Scheme definition: we require that $step_i$ be given.

Syntax Extension via Macros: hygiene

- Note that a new identifier, `my-do-fn` was introduced in the example.
- Scheme syntax rules are "hygienic" – new names introduced are made not interfere with the scoped names in the source code.
- E.g. `my-do-fn` below does not mess up `my-do`.

```
(define (my-do-fn x) (write x) (+ x 1))

(my-do ((i 1 (my-do-fn i))) ((> i 3) 99) (write "Yip"))

"Yip"1"Yip"2"Yip"3
99
```

- The Scheme Report says:
  - If a macro transformer inserts a binding for an identifier (variable or keyword), the identifier will in effect be renamed throughout its scope to avoid conflicts with other identifiers. Note that a define at top level may or may not introduce a binding; see section 5.2.
  - If a macro transformer inserts a free reference to an identifier, the reference refers to the binding that was visible where the transformer was specified, regardless of any local bindings that may surround the use of the macro.

Syntax Extension via Macros: Conclusion

- Packages exist to build the syntax target constructs with full-blown programs.
- Macros in Scheme are simple and powerful (*e.g.* compared to C) because
  - Scheme has a uniform representation for source trees (lists).
  - Transformations are based on *trees*, not *character streams*
  - Different transformations can be applied in different situations, as determined by patterns and programs.
  - The language is *expression-based* with forms for *local bindings* so the macro results can be complex and self-contained.
  - Macros are *hygienic* so new bindings do not interfere with old
  - (for those who are curious) Arbitrary computation may be done in the re-writing done by Scheme macros.