Macro Expansion/ Macro Languages

Lecture 26

Lecture Outline

- Languages can be extended by text transformations
- C preprocessor is a weak example
- Lisp macro definitions are stronger
- Examples

C Preprocessor

- #include "*filename*"
- #define YES 1
- #define *id1(id2, id3…) token-string*
- #undef *identifier*
- #if *constant-expression*
- #ifdef *identifier*
- #ifndef *identifier*
- #else
- #endif
- #line *constant id* /* renumber the next line re errors */

C Preprocessor Example

- We could do this

```
#define then
#define begin {
#define end ;}
if (i>0) then begin a=1; b=2 end
```

Becomes

```
if (i>0)  {a=1;b=2;}
```

C Preprocessor Strength

- It does certain useful things to make up for deficiencies in the language. Conditional compilation subject to compile-time flags def/ifdef
- (Java doesn't have a preprocessor though!)

C Preprocessor Weaknesses

- It is a DIFFERENT language from C.
- For example

  #define foo(a,b) 2*a+b

  #define foo (a,b) 2*a+b

  mean entirely different things

  Scope? We don't need no stinking scope. Or do we?

Other Macro / string processors

- M6, came with original UNIX
- TeX (typesetting program from D. Knuth)
- Emacs Elisp
- (perhaps stretching the form)
  - VB macros for Microsoft Word, etc. -- not just string processing
  - Perl, Tcl are "scripting" languages with possible substitution usages

Lisp Macro Definitions: the idea

- It is part of the same language:
- Change the interpreter, before calling a function (foo a b c); ask: is foo defined as a macro, if so, replace it by its expansion. Then continue.
- For the compiler, before compiling a function call (foo a b) ask: is foo a macro…

What do we change to put if* then elseif else in Common Lisp?

in lisp now: (if a b c).

This is clumsy. What if you want to change the code to do (if a {b1 b2} {c1 c2 c3}). You have to rewrite as

(if a (progn b1 b2)(progn c1 c2 c3)).

Harder to edit: must insert (progn ..b2) not just b2.

Proposal. A NEW language syntax

(if* a then b1 b2 else c1 c2) or even

(if* a then b1 b2 elseif c1 c2 then d else e)

violates spirit of lisp BUT if you are willing to agree that b1, b2, etc are never then elseif...etc we can write a macro to transform to "normal" lisp.

A definition of if* (including checking ....)

```
(defvar if*-keyword-list '("then" "thenret" "else" "elseif"))

(defmacro if* (&rest args)      ;;; not for human consumption during lecture...
   (do ((xx (reverse args) (cdr xx))
    (state :init)
    (elseses nil)
    (totalcol nil)
    (lookat nil nil)
    (col nil))
       ((null xx)
    (cond ((eq state :compl)
           `(cond ,@totalcol))
          (t (error "if*: illegal form ~s" args))))
       (cond ((and (symbolp (car xx))
             (member (symbol-name (car xx))
                     if*-keyword-list
                     :test #'string-equal))
          (setq lookat (symbol-name (car xx)))))

       (cond ((eq state :init)  .... more on next slide....
```

A definition (including checking ....)

```
..... continued from previous...
(cond (lookat (cond ((string-equal lookat "thenret")
                                (setq col nil
                                      state :then))
                               (t (error
                                   "if*: bad keyword ~a" lookat))))
            (t (setq state :col
                      col nil)
               (push (car xx) col))))
       ((eq state :col)
        (cond (lookat
               (cond ((string-equal lookat "else")
                      (cond (elseseen
                             (error
                              "if*: multiples elses")))
                      (setq elseseen t)
                      (setq state :init)
                      (push `(t ,@col) totalcol))
                     ((string-equal lookat "then")
                      (setq state :then))
                     (t (error "if*: bad keyword ~s"
                                       lookat))))
            (t (push (car xx) col))))
... even more... see http://www.franz.com/~jkf/ifstar.txt
```

Lisp Macro Implementation: Interpreter

```
(defun interp (x &optional env)
  "Interpret (evaluate) the expression x
    in the environment env.
  This version handles macros."
  (cond
    ((symbolp x) (get-var x env))
    ((atom x) x)
    ((is-it-a-macro (first x))
     (interp (macro-expand x) env))
    ((case (first x)
...
```

Rest of interpreter changes

```
(defun macro-expand (x)
  "Macro-expand this Lisp expression."
  (if (and (listp x) (is this-a-macro (first x)))
      (macro-expand
        (apply (is-this-a-macro (first x)) (rest x)))
      x))

(defun is-this-a-macro(name) (get name 'macro))

(defmacro def-macro (name parmlist &body body)
  "Define a lisp  macro."
  `(setf (get ',name 'scheme-macro)
         #'(lambda ,parmlist .,body)))
```

Lisp Macro Definitions permeate language

- Used routinely, even for parts of the base language! let, let*, and, or, cond, case.
- THERE IS NO AND in LISP

  (macroexpand-1 '(and a b c)) ➔

  (cond ((not a) nil) ((not b) nil) (t c))
- Used for defining NEW constructs in the language, with nearly complete freedom of semantics.
- Maintains similar parenthesized appearance.

Lisp Macro Definition : downside

- It can become complex to define a macro that is as general as possible and correct.
- If macros are repeatedly expanded (as they are in the interpreter), they can slow down operation.
- Expanding macros "once, in place" is possible though.

Lisp Macro Definitions don't change lexical model

- If you wish to change the LEXICAL MODEL of Lisp, e.g. allow x:=a+b, no spaces, to be 5 tokens, this cannot be done with the lisp macroexpansion.
- There is however a lisp "reader macro" and readtable facility that is character-oriented.
- By using a character macro we can switch surface syntax, change lexer, parser, etc... so we can embed MJ code in lisp
- (+ 23 %{int x; method f(){return x+1;}%) ☺

You have already done macro expansions by hand for MJ. And Lisp

We don't need AND and OR .. We can use IF. But then….
We don't need IF .. We can use COND
Do we have to make special checks for these operations, or can we make general facilities.

Of course we can..

```
(defmacro and(a b) (list 'if a b))   ;; or using ` notation…
(defmacro and(a b) `(if ,a ,b))

(defmacro myor (a b) `(if ,a t ,b))
(defmacro if (pred th el) `(cond ((,pred th)(t ,el))))
```

The magic and the mystery

From time to time we have used common lisp program segments usually without comment, like

```
(incf x)
  ;; this computes y=x+1, stores it in x and returns y.
```

Can we write a function to do this?

```
(defun myincf (x)(setf x (+ x 1)))  ;; no good.
(setf z 4)
(myincf z) --> 5  but z is still 4. We don't have access to z inside myincf.
```

Expansion in place

The trick is to expand in place the form (incf z) and
replace it with (setf z (+ z 1))

```
(defmacro myincf(z)(list 'setq z (list '+ z 1)))  ;; or more succinctly
(defmacro myincf(z)`(setf ,z (+ ,z 1)))
```

(macroexpand '(incf x)) → (setq x (+ x 1))

;; in fact, this is what the common lisp system does too:

(macroexpand '(incf x)) →
(setq x (+ x 1))

What do we expand, exactly?

```
;; it is not always so simple.
(macroexpand '(myincf (aref z (incf a)))) →
(setq (aref z 3) (+ (aref z 3) 1)) ;; looks OK      z[3]:=z[3]+1

(macroexpand '(myincf (aref z (incf a))))
(setq (aref z (incf a)) (+ (aref z (incf a)) 1))  ;; NOPE z[a+1]:=z[a+2]+1

;; Here's a "better" version
(macroexpand '(incf (aref z (incf a)))) ;; produces something like this
(let* ((temp1 z)
       (temp2 (incf a))
       (temp3 (+ (aref temp1 temp2) 1)))
  (internal-set-array temp3 temp1 temp2))  ;;   z[a+1]:=temp3
```

Problems with free variables rebound

```
;; It can't do EXACTLY that because look what happens here

(defparameter temp1 43)

(macroexpand '(incf (aref z (+ temp1 1))))

(let* ((temp1 z)
       (temp2 (+ temp1 1))  ;; supposed to be 44 but is not
       (temp3 (+ (aref temp1 temp2) 1)))
  (internal-set-array temp3 temp1 temp2))

;; LAMBDA CALCULUS EXPLAINS WHY THIS IS AN ERROR ABOVE.
```

Problems with free variables rebound:solved

```
;; It can't do EXACTLY that because look what happens here

(defparameter temp1 43)

(macroexpand '(incf (aref z (+ temp1 1))))


;;actually, to avoid this kind of problem
;; expansion generates new, unique, labels.

(let* ((#:g23729 z)
       (#:g23730 (+ temp1 1))
       (#:g23731 (+ (aref #:g23729 #:g23730) 1)))
  (excl::.inv-s-aref #:g23731 #:g23729 #:g23730))
```

LOOPs

;; LOOPs don't have to be compiled because they are macro-expanded away.

```
 (macroexpand '(loop (f x) (incf x)(print x)))

(block nil
  (tagbody
   #:g23661 (progn (f x) (incf x) (print x))
            (go #:g23661)))
```

LOOPs

```
;; a fancier loop macro
(macroexpand '(loop for i from 1 by 2 to lim collect i))

(let ((i 1)
      (#:g23666 lim))
  (declare (type real #:g23666) (type real i))
  (excl::with-loop-list-collection-head (#:g23667 #:g23668)
    (block nil
      (tagbody
       excl::next-loop (when (> i #:g23666) (go excl::end-loop))
              (excl::loop-collect-rplacd (#:g23667 #:g23668) (list i))
              (excl::loop-really-desetq i (+ i 2))
              (go excl::next-loop)
       excl::end-loop (return-from nil (excl::loop-collect-answer
#:g23667))))))
```

SETF

SETF is a fairly subtle kind of program.

```
 (macroexpand '(setf (cadr x) 'hello))
(let* ((#:g23663 (cdr x))
       (#:g23662 'hello))
  (excl::.inv-car #:g23663 #:g23662))

 (macroexpand '(setf (aref (cadar x) 43) 'hello))
(let* ((#:g23664 (cadar x))
       (#:g23665 'hello))
   (excl::.inv-s-aref #:g23665 #:g23664 43))
```

IF*, like IF but with key words

```
;; in case you dislike the lack of key-words in an if, use if*

 (macroexpand '(if* a then b c else d e)) →
(if a (progn b c) (cond (t d e)))

(macroexpand '(if* a then b c elseif d then e else f)) →
(if a (progn b c) (cond (d e) (t f)))

(macroexpand '(dotimes (i 10 retval) (f i))) →
(let ((i 0))
  (declare (type (integer 0 10) i))
  (block nil
    (tagbody
     #:Tag608 (cond ((>= i 10) (return-from nil (progn retval))))
              (tagbody (f i))
              (psetq i (1+ i))
              (go #:Tag608))))
```

While, Unless

```
(macroexpand '(while x y))

(block nil
  (tagbody
   #:g23670 (progn (unless x (return)) y)   ;;; what is unless??
            (go #:g23670)))

(macroexpand '(unless x y))
(if (not x) (progn nil y) nil)
```

Push

```
(macroexpand '(push x y)) ;;; simple version
(setq y (cons x y))

(macroexpand '(push x (aref stackarray 1))) ;;full version
(let* ((#:g23678 x)
       (#:g23676 stackarray)
       (#:g23677 (cons #:g23678 (aref #:g23676 1))))
  (excl::.inv-s-aref #:g23677 #:g23676 1))
```

Pop

... reminder .... here's how it works...
(setf x '(a b c d))
(a b c d)

(pop x)
a
x → (b c d)
.....
could we do (pop x) this way?

```
(let ((ans (car x)))
   (setf x (cdr x))
   ans)
;;or in lisp idiom
  (prog1 (car x)(setf x (cdr x)))
```

Pop, continued

```
;; The real pop does this…
 (macroexpand '(pop x))
(let* ((#:g23682 nil))
  (setq #:g23682 x)
  (prog1 (car #:g23682)
      (setq #:g23682 (cdr #:g23682))
      (setq x #:g23682)))
```

The moral: Each object must be evaluated at most once.

How far can macroexpansion be pushed?

What more can be done? 100,000 lines of series code originally by Richard Waters..

```
(macroexpand '(collect-sum (choose-if #'plusp (scan '(1 -2 3 -4)))))
```

This means essentially the same as this..

```
(let ((sum 0)) ; reference
  (dolist (i '(1 -2  3 -4) sum)
    (when (plusp i) (setq sum (+ sum i)))))
```

but uses "series". Macroexpansion of the original translates series into loops like this..

Series code as Loop

How does it work?
(series 'b 'c) --> #z(b c b c ...) ;; self evaluating

(scan (list 'a 'b 'c)) --> #z(a b c)

(scan-range :upto 3)
(scan-range :from 1 :by -1 :above -4)

scan returns a series containing the elements of its sequence in order

optional type, e.g. (scan 'string "BAR") --> #Z(#\B #\A #\R)

Series code as Loop

```
(macroexpand '(collect-sum (choose-if #'plusp (scan '(1 -2 3 -4)))))

(LET*       (#:ELEMENTS-198793
               (#:LISTPTR-198794 '(1 -2 3 -4))
               (#:SUM-198769 0))
  (DECLARE (TYPE LIST #:LISTPTR-198794) (TYPE NUMBER
#:SUM-198769))
  (TAGBODY
   #:LL-198859 (IF (ENDP #:LISTPTR-198794) (GO SERIES::END))
         (SETQ #:ELEMENTS-198793 (CAR #:LISTPTR-198794))
         (SETQ #:LISTPTR-198794 (CDR #:LISTPTR-198794))
         (IF (NOT (PLUSP #:ELEMENTS-198793)) (GO #:LL-198859))
         (SETQ #:SUM-198769 (+ #:SUM-198769 #:ELEMENTS-
198793))
         (GO #:LL-198859) SERIES::END)
  #:SUM-198769)
```

Series code as Loop

Series expressions are transformed into loops by pipelining them -the computation is converted from a form where entire series are computed one after the other to a form where the arguments to series functions are incrementally referenced/ computed in parallel. In the resulting loop, each individual element is computed just once, used, and then discarded before the next element is computed.

For this pipelining to be possible, a number of restrictions have to be satisfied, basically requiring that you can fix the amount of the "input" pipeline needed to compute the first element and subsequent elements of the "output" pipeline.

Language definition/ extension…

- Much more in CS 263/ 264
- Vast literature on language extension techniques