Genuine, Full-power, Hygienic Macro System for a Language with Syntax

David A. Moon

International Lisp Conference

March 23, 2009

Outline:

Introduction

Programming language

Goal of this talk

Objects

Syntax

Macros

Genuine, Full-Power, Hygienic, Easy to Use

Examples

Code Walking

Programming Language for Old Timers

A hobby, not a serious implementation

Trying to do things "right"

> You know it is right when both simplicity and power are maximized, while at the same time confusion and kludges are minimized. This is hard!

Cleanliness, Flexibility, Orthogonality, Extensibility

A dialect of Lisp, even if it doesn't look like it

Goal of this talk

Show it is possible to have good macros in a language with syntax. “Good” means:

Genuine = structural, not string substitution

Full-power = macros can compute, can accept any syntax, can do anything that the built-in syntax can do

Hygienic = no unintentional name clashes

PLOT Objects

Data defined by **classes**

- Slots, inheritance, constructor
- No magic “primitive types,” only class instances
- Multi-valued slots instead of magic array objects

Behavior defined by **function methods**

- Call a function with arguments
- Dispatches to most specific applicable method

Classification of data defined by **types**

- A type is a dichotomy over all objects
- Type = class, integer range, union, protocol

PLOT Syntax

Infix syntax for operators and function calls

Uniform syntax

- Unify expressions, statements, and declarations
- Operator is a definition, not an inherent property

Minimize punctuation

- BCPL: Omit semicolon at end of line. If line ends with operator it continues on the next
- Python: Indentation, not brackets, for nesting

Operator macros

- (is a macro with function on lhs, args on rhs

PLOT Syntax Examples

```
if x < 3 foo(x) else bar(x, y)

while f1(x, precise: true) < 3
   f2(x)
   f3(x)

for i from 0 below b.length, j downfrom k
   a[j] := b[i]

block exit: return
   traverse([x, y => if x > y return(x, y)], tree)
```

PLOT Syntax Examples (continued)

```
def pi = 3.14159

def fib(x) fib(x - 1) + fib(x - 2)
def fib(x is integer(infinity, 1)) 1

defclass point
   x is number
   y is number

defmacro print ?expr => `write(stdout, ?expr)`
```

Genuine, Full-Power, Hygienic, Easy to Use Macros

Structural like Lisp
works on Abstract Syntax Trees
not string substitution like C

This means that everything nests properly.

You can write macro-defining macros.

You can do code walking.

Genuine, Full-Power, Hygienic, Easy to Use Macros

PLOT achieves genuineness by implementing macros as functions that execute inside the compiler (or IDE).

A macro parses from a stream that yields tokens and/or Abstract Syntax Trees.

A macro produces an Abstract Syntax Tree object or a token list to be re-parsed.

Genuine, Full-Power, Hygienic, Easy to Use Macros

Abstract Syntax Tree = object-oriented S-expr

Types:

literal	conditional
quotation	definition
name	block-expression
invocation	assignment *etc.*

Genuine, **Full-Power**, Hygienic, Easy to Use Macros

1. Arbitrary computations can be executed during macro expansion.

Macros can:

- communicate with each other
- operate on already-parsed code
- be aware of scopes and definitions
- do file I/O

Genuine, **Full-Power**, Hygienic, Easy to Use Macros

2. Macros can accept any syntax that is possible to parse.

Not limited to a fixed set of predefined forms
e.g. ***foo*** and ***foo(arg, arg, ...)*** in C
e.g. ***(foo ...)*** in Lisp and Scheme

Users are free to use whatever is most expressive for their purposes.

Genuine, **Full-Power**, Hygienic, Easy to Use Macros

3. Macros can do anything the language's built-in syntax can do (there is no magic).

Thus user-defined domain-specific languages can be syntactically compatible with the base language.

Macros can call the same syntactic type parsers that the built-in syntax calls.

Genuine, **Full-Power**, Hygienic, Easy to Use Macros

PLOT achieves full power by implementing macros as functions that execute inside the compiler (or IDE). Thus macros can do anything that any function can do.

A macro entirely controls its parsing.

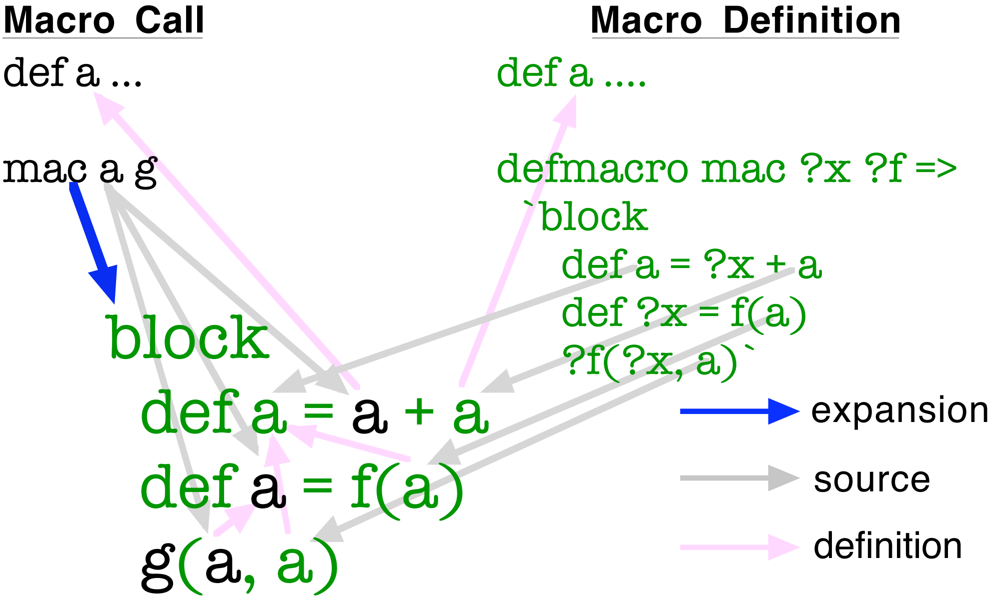
PLOT has no built-in syntax! Everything is a macro, exported by a predefined module.

Genuine, Full-Power, **Hygienic**, Easy to Use Macros

No unintentional name clashes:

- a name introduced by a macro call means what it means at the call site.
- a name introduced by a macro definition means what it means at the definition site.
- a macro can expand into local definitions, invisible to caller.
- deliberately visible local definitions allowed, caller can supply name or can be anaphoric.

Genuine, Full-Power, **Hygienic**, Easy to Use Macros



Genuine, Full-Power, **Hygienic**, Easy to Use Macros

PLOT achieves hygiene by storing a "context" in a slot of each *name* object in an Abstract Syntax Tree.

A name introduced by a macro definition has a context that allows it to see only definitions introduced by the same macro expansion, plus definitions in scope at the macro definition.

Genuine, Full-Power, **Hygienic**, Easy to Use Macros

A name introduced by a macro call does not have the same context as one introduced by a macro definition, so it cannot see invisible local definitions in the macro expansion.

This works for macro-defining macros too. There are three relevant contexts for names: end-user call, defining-macro call, defining-macro definition.

Genuine, Full-Power, Hygienic, **Easy to Use** Macros

Writing parsing code can be very tedious and also error-prone.

Constructing an Abstract Syntax Tree one node at a time can be tedious.

Keeping track of the contexts for hygiene is a burden on macro writers.

Genuine, Full-Power, Hygienic, **Easy to Use** Macros

Solution: Domain-specific languages for macros
- Patterns for parsing
- Templates for constructing expansion
- Macros are not *required* to use these

Declarative is easier than imperative.

Parsing can call library syntactic type parsers.

No extra work to be hygienic.

No extra work for visible local definitions with caller-supplied name.

Patterns

defmacro if
{ ?test [then] ?then is block & elseif }+
[else ?else is block] => ...

? introduces a pattern variable
? *name* is *syntactic type* (default = expression)
{ ... }+ means repeat one or more times
& introduces separator between repeats
[...] means optional

Templates

```
`def  loop({ ?vars is ?types &, }*)
        if { ?tests & and }+
          ?body
          loop({ ?steps &, }*)`
```

` is the template macro
? introduces a substitution variable
{ ... }* means repeat zero or more times
& introduces separator between repeats

Macros' place in the compiler

Extension of parse phase (*NOT* transform phase!).
Compiler calls ***parse-expression***.
When ***parse-expression*** sees a name defined as a macro, it calls the macro's parse function.
Extensible syntax requires LL(1) recursive descent parsing rather than grammar compiler.
Macro returns AST or token list.
Macro expanding into a macro call means next macro could see AST as a token.

Example Macro: print

```
def print = macro(
  [tokens is token-stream, previous-context =>
   invocation(#write,
              #stdout,
              parse-expression(tokens,
                               previous-context, true))])

print x + y * z
            =>
              write(stdout, x + y * z)
```

Example Macro: if

```
;; A simplified version of if
defmacro if ?test ?then [ else ?else ] =>
   conditional(test, then, else or quotation(false))
```

Example Macros: if

```
;; This is the real definition of if
defmacro-block if
              { ?test [then] ?then is block & elseif }+
              [ else ?else is block ] =>
  reduce-right(conditional,
                   else or quotation(false),
                   test,
                   then)
```

Example Macro: for

```
defmacro for ?var is name from ?from to ?to
                ?:body =>
  `block
     def start = ?from
     def limit = ?to
     def loop(?var)
        if ?var <= limit
          ?body
          loop(?var + 1)
     loop(start)`
```

Example Macro: defmacro

```
defmacro defmacro
        ?:name { ^ ?:pattern \=> ?:block }+ =>
  def msg = sequence-to-string(
        collect-pattern-starts(pattern), ", ", " or ")
  def err = `wrong-token-error(?=tokens, ?msg)`
  def expander = reduce-right(
        translate-pattern(`?=tokens`, _, _, _),
        err, pattern, block)
```

continued on next slide

Example Macro: defmacro (pg 2)

```
`def ?name = macro([name: ?name,
                    ?=tokens is token-stream,
                    ?=previous-context =>
  def ?=macro-context = unique-macro-context()
  def ?=source-file, ?=source-line =
        source-location(?=tokens)
  ?expander ])`
```

Example Macro: App-specific

```
defmacro def-character-class
            ?:name = { ?expr }+ [ size: ?size] =>
  ;; Functions to convert input to character code ranges
  def range(x is invocation) code(x.args[0]) : code(x.args[1])
  def range(x is anything) code(x) : code(x)
  def code(x is integer) x
  def code(x is character) char-code(x)
  ;; Build the bit vector at compile time
  def bits = bit-vector#(size or 256)
  for x in expr
    for code in range(x)
      bits[code] := 1
```

continued on next slide

Example Macro: App-specific (pg 2)

```
;; Define name to be that constant bit vector
def constant = quotation(bits)
`def  ?name = ?constant`


;; Example uses of the macro

def-character-class whitespace = ' '   '\t'   10   13

def-character-class letters = ‘A’ : ‘Z’          ; Majuscules
                              ‘a’ : ‘z’          ;miniscules
```

Code Walking

Traditionally, code walking has required ad hoc code to understand every “special form.”

It is better to have a well-defined, object-oriented interface to the Abstract Syntax Tree, scopes, and definitions. This is why objects are better than S-expressions as a representation for program source code.

Code Walking Protocol

Collecting code walk

```
;; reduce(function, initial-value, subexpressions(e))
require walk(f is function, e is expression, s is scope,
              initial-value is anything,
             result: new-value)
```

Replacing code walk

```
;; Replace each subexpression of e with f(sub,scope)
require walk(f is function, e is expression, s is scope,
             result: new-expression is expression)
```

Code Walking Example

```
def free-variables(e is expression,
            optional: s is scope = local-scope(),
                      vars is collection = stack())
  walk(free-variables, e, s, vars)

def free-variables(n is name,
            optional: s is scope = local-scope(),
                      vars is collection = stack())
  if lookup(s, n) or member?(n, vars) then vars
  else add(n, vars)
```

Another Code Walking Example

Stick *trace* in front of an expression to trace all function calls inside it.

```
defmacro trace ?expr =>
  add-tracing(expr, get-local-compiler-scope())

;; Default method just walks over subexpressions
def add-tracing(e is expression, s is scope)
  walk(add-tracing, e, s)
```

Another Code Walking Example (pg 2)

```
;; This method adds tracing to a function call
def add-tracing(e is invocation, s is scope)
  def fcn = add-tracing(e.function, s)
  def args = map(walk(add-tracing, _, s),
                 e.arguments)
  def macro-context = unique-macro-context()
  def temps =
        for n from 1 to args.length
          collect name("temp-?n", macro-context)
```

continued on next slide

Code Walking Example (pg 3)

```
parse-expression(token-sequence-stream(
  `do
    def fcn = ?fcn
    { def ?temps = ?args & ^ }*
    def results = values-list(fcn( { ?temps &, }* ))
    write(*trace-output*,
          " " + fcn + "(" { + ?temps }* +
          ") = " + results + "\n")
     values(results...)`),
  false, true)
```

For More Information

For lots of expository text and larger examples, see

http://users.rcn.com/david-moon/PLOT/index.html

Questions?

Maybe some answers.