# *List Comprehension*

Simple
and Efficient Compilation of
List Comprehension
in Common Lisp

Mario Latendresse
Bioinformatics Research Group
SRI International

International Lisp Conference, 1-4 April 2007
Clare College, Cambridge, UK

# *This Talk Could Have Been Titled ...*

- Recursion in Common Lisp is not like in Scheme.
- Optimization declarations in Common Lisp is not always portable.
- The Loop Facility has a nice semantics; it is just there to be discovered.
- The right solution is often time the simplest; here is another example.

# *The Context*

The design of a language to query metabolic pathways and genomic databases (PGDBs) on the Web (*Pathway Tools*).

The result is *BioVelo*, a typed language, based on List Comprehension:

Example

```
[ (x^name, x^left, x^right) :
          x <- ecoli^^reactions,
          #x^right > 3 ]
```

Let see `www.biocyc.org`, the `Advanced Database Search` in the left menu.

# BioVelo Structured Web Interface

http://www.biocyc.org/query.html

http://www.biocyc.org/query.html

http://www.biocyc.org/que...

The old Advanced Query Page

## Structured Advanced Query Page

Switch to the Free Form Advanced Query Page

Advanced Query documentation

### Specify your query below

In database Escherichia coli (ECOLI) search for -Polypeptides (let's call them x1) remove the entire condition below

Where for each object ... in CATALYZES of type Enzymatic-Reactions (let's call them x2)

we have x2 NAME contains the substring Switch to variable entry

add a condition

add a condition

insert a new search component here

### Specify the contents of the output of your query below

| Column 1 | Column 2 | add a column |
|---|---|---|
| Sort based on this column | Sort based on this column | |
| NAME | FRAME-ID | |

Select your desired output format: HTML Tab Delimited Text (columns are separated by tabs)

Submit Query Reset Query

# BioVelo Free Form Web Interface

http://www.biocyc.org/query.html

http://www.biocyc.org/query.html

Google

http://www.biocyc.org/que...

The old Advanced Query Page

## Free Form Advanced Query Page

Switch to the Structured Advanced Query Page

Advanced Query documentation

**Enter your query below according to the *BioVelo* language syntax.**

**Below are some query examples.**

1. All the proteins of *E. coli*.

   `[x : x <- ecoli^^proteins]`

   This can be read as follows: return all `x` such that `x` is an object in the class `proteins` in the database `ecoli`. In *BioVelo*, the database name must be an identifier, as defined in the *BioVelo* language, not containing any spaces or special characters.

   The colon ':' delimits the head of the query, which in this case is `x`, and the rest of the query, which are *qualifiers*, that is,

The following selectors are provided as reference documentation to help you write your query above — they do not have to be used to create your query. For some browsers, when you select an option from one of these selectors and then double click it, the option value will be copied in the text area above. For some browsers, a tooltip (i.e., some text in a yellow box) appears for some selectors when you hover the mouse pointer over them.

| Databases | Classes | Attributes | Operators |
|---|---|---|---|
| Acidithiobacillus ferrooxidans (AFER243159) | THING | NAME | [ ... ] |

Select your desired output format: ⊙ HTML ○ Tab Delimited Text (columns are separated by tabs)

Submit Query  Reset Query

# *BioVelo Example*

Find all enzymes of *E. coli* that catalyze two reactions in the same pathway.

```
[(es, p, r1, r2) :
  p <- ecoli^^pathways,
  r1 <- p^reaction-list,
  r2 <- p^reaction-list,
  r1 != r2,
  es:=(r1^enzymatic-reaction**r2^enzymatic-reaction),
  #es > 0]
```

# BioVelo Example, Web Result

Advanced Query Results

http://www.biocyc.org/query?submit=Submit+Query&object=%28%22HTML%22+%22%0

Google

Advanced Query Results

## Advanced Query Results

Your query in BioVelo is " [([y^enzyme:y<-es], p, r1, r2) : p <- ecoli^^pathways, r1 <- p^reaction-list, r2 <- p^reaction-list, r1 != r2, es := (r1^enzymatic-reaction ** r2^enzymatic-reaction), #es > 0]".

This query resulted in a single table of 14 rows.

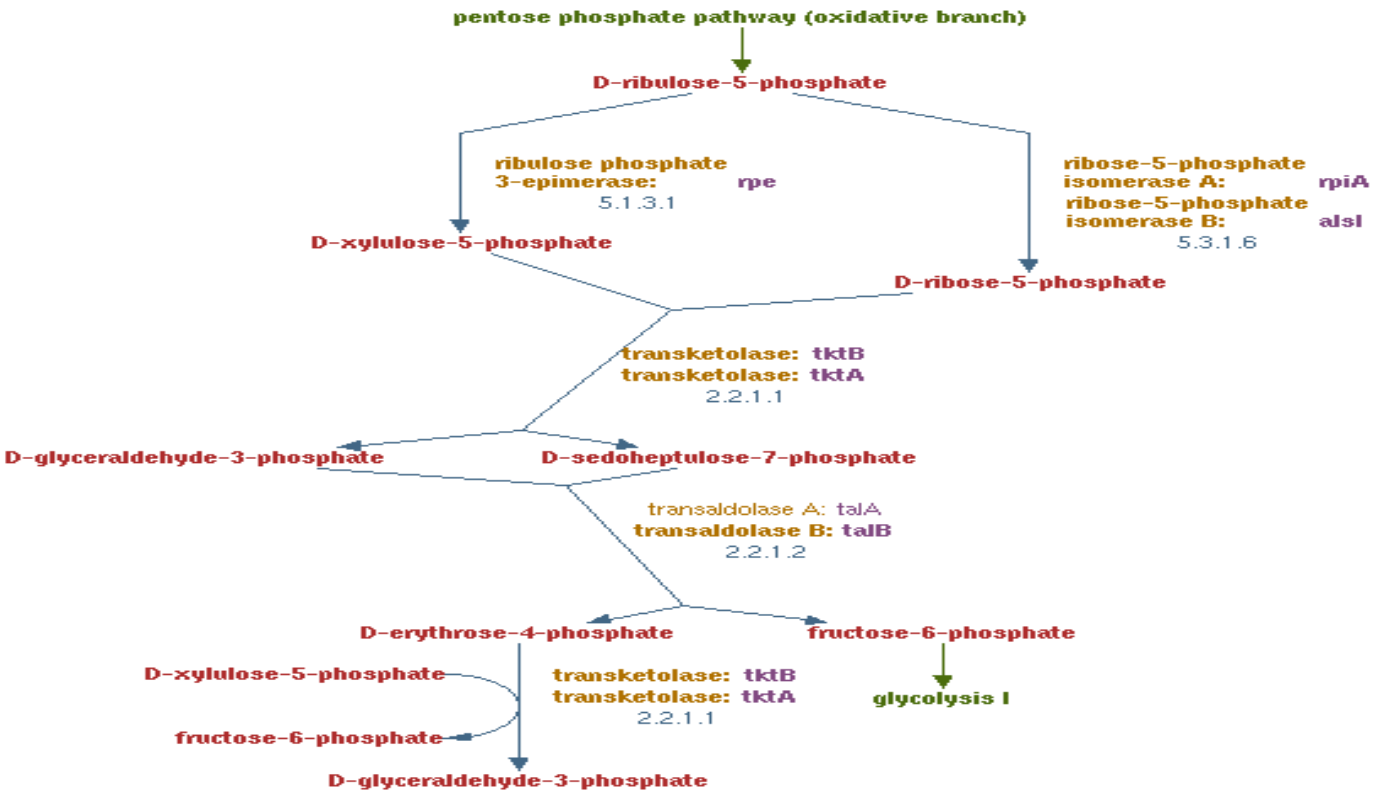
| Column 1 for [(y ^ enzyme) : y <- es] | Column 2 for p | Column 3 for r1 | Column 4 for r2 |
|---|---|---|---|
| TRANSKETOI-CPLX | NONOXIPENT-PWY | 1TRANSKETO-RXN | 2TRANSKETO-RXN |
| TRANSKETOI-CPLX | NONOXIPENT-PWY | 2TRANSKETO-RXN | 1TRANSKETO-RXN |
| CPLX0-235 | KETOGLUCONMET-PWY | 1.1.1.215-RXN | YIAE1-RXN |
| CPLX0-235 | KETOGLUCONMET-PWY | 1.1.1.215-RXN | YIAE2-RXN |
| CPLX0-235 | KETOGLUCONMET-PWY | YIAE1-RXN | 1.1.1.215-RXN |
| CPLX0-235 | KETOGLUCONMET-PWY | YIAE1-RXN | YIAE2-RXN |
| CPLX0-235 | KETOGLUCONMET-PWY | YIAE2-RXN | 1.1.1.215-RXN |
| CPLX0-235 | KETOGLUCONMET-PWY | YIAE2-RXN | YIAE1-RXN |
| PDXK-CPLX | PLPSAL-PWY | PYRIDOXKIN-RXN | PNKIN-RXN |
| PDXK-CPLX | PLPSAL-PWY | PNKIN-RXN | PYRIDOXKIN-RXN |
| EG11081-MONOMER | NONMEVIPP-PWY | ISPH2-RXN | RXN0-884 |
| EG11081-MONOMER | NONMEVIPP-PWY | RXN0-884 | ISPH2-RXN |
| MYRISTOYLACYLTRAN-MONOMER | KDO-NAGLIPASYN-PWY | MYRPALMTRAN-RXN | MYRISTOYLACYLTRAN-RXN |
| MYRISTOYLACYLTRAN-MONOMER | KDO-NAGLIPASYN-PWY | MYRISTOYLACYLTRAN-RXN | MYRPALMTRAN-RXN |

Quick Search Go Query Page Advanced Query Page BioCyc Home Report Errors or Provide Feedback

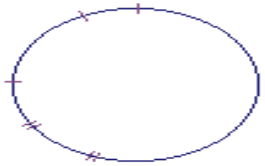
Page generated by SRI International Pathway Tools version 11.0 on Mon Apr 2, 2007, biocyc07.
EcoCyc version 11.0.

# BioVelo links to Web Database

More Detail | Less Detail | Cross-Species Comparison | Download Genes | BioPAX format




Locations of Mapped Genes:



# *The Question*

How to compile List Comprehension in Common Lisp?

This should be straightforward. Or is it?

# *Searching for List Comprehension Compilation*

In March 2006, searching Google with '`translating List Comprehension Common Lisp`'

It delivers Guy Lapalme's 1991 paper in Lisp Pointers:
*Implementation of a "Lisp comprehension" macro*

The translation is based on Wadler's "three rules". The macro is short and elegant. It can be used directly for *BioVelo*. A good start ...

# *A Bit of Syntax*

A list comprehension will have the form

( $h$ $q_1 \ldots q_n$ )

where $h$ is the *head* and the $q_i$ are either

- a *generator* of the form ( $x$ <- $X$ )
- a *filter*, i.e., a Boolean expression.

Example:

```
((list x y) (x <- '(1 2 3 4)) (< x 4)
            (y <- '(a b)))
```

gives

```
((1 a)(1 b)(2 a)(2 b)(3 a)(3 b))
```

# Lapalme's List Comprehension Macro

```
(defmacro comp ((e &rest qs) l2)
  (if (null qs) '(cons ,e ,l2)              ; rule A
    (let ((q1 (car qs))
          (q  (cdr qs)))
     (if (not (eq (cadr q1) '<-))           ; rule B
         '(if ,q1 (comp (,e ,@q) ,l2) ,l2)
       (let ((v   (car q1))                 ; rule C
             (l1  (third q1))
             (h   (gentemp "H-"))
             (us  (gentemp "US-"))
             (us1 (gentemp "US1-")))
         '(labels
            ((,h (,us)          ; a generator
                (if (null ,us) ,l2
                 (let ((,v   (car ,us))
                       (,us1 (cdr ,us)))
                   (comp (,e ,@q) (,h ,us1))))))
            (,h ,l1)))))))
```

# A Translation of comp

Compiling

```
(x (x <- i50000) (y <- '(a b c)) (> 0 x))
```

translates to

```
(labels
  ((h-1000 (us-1001)
     (if (null us-1001)
        nil
      (let ((x (car us-1001))
            (us1-1002 (cdr us-1001)))
        (labels ((h-1003 (us-1004)
            (if (null us-1004)
               (h-1000 us1-1002)
              (let ((y (car us-1004))
                    (us1-1005 (cdr us-1004)))
                 (if (> 0 x)
                   (cons x (h-1003 us1-1005))
                  (h-1003 us1-1005))))))
         (h-1003 '(a b c)))))))
 (h-1000 i50000))
```

# *Problems ...*

But, two major problems:

- stack overflows. To control it, you need to control the optimization declarations of your implementations.
- difficult to debug when a break occurs due to the obscure translation into Wadler's rules.

A search for a simpler translation: what about the Loop Facility? That could be ugly..., or is it?

# Our Compilation using the Loop Facility

```
(defmacro lc ((h &rest qs))
  `(loop repeat 1 ,@(lcr h qs)))

(defun lcr (h qs)
 (if (null qs) `(collect ,h)  ;; head
  (let ((q1 (first qs))
        (qr (rest  qs)))
   (if (eq (second q1) '<-)
     (let ((v  (first q1))    ;; generator
           (r  (third q1)))
       `(nconc (loop for ,v in ,r
                     ,@(lcr h qr))))
    `(if ,q1 ,@(lcr h qr))    ;; filter
   ))))
```

# A Translation of `lc`

Compiling

```
(x (x <- i50000) (y <- '(a b c)) (> 0 x))
```

translates to

```
(loop repeat 1
  nconc (loop for x in i50000 nconc
          (loop for y in '(a b c)
             if (> 0 x) collect x)))
```

# *Macro `lc` can Handle Tuples in Generators*

Tuple assignments come for free since the Loop Facility provides it.

Example:

```
(x ((x y) <- '((1 a) (2 b) (3 c))))
```

translates to:

```
(loop repeat 1
 nconc (loop for (x y) in '((1 a) (2 b) (3 c))
        collect x))
```

Gives `(1 2 3)`

# Extended Translation (1)

Generators with:

- Vectors and strings instead of lists: `<--`

```
(x (x <-- "abcdef"))
```

- Range of integers: `<..`

```
(x (x <..  1 10))
```

- Binding operation: `:=`

```
(y (x <..  1 18) (y := (* x x)) (< y 95))
```

# Extended Translation (2)

```
(defun lcer (h qs)
  (if (null qs) `(collect ,h)   ;; head
    (let* ((q1 (first qs))
           (op (second q1))
           (qr (rest  qs)))
      (if (member op '(<- <-- := <..))
        (let ((v (first q1))  ;;generator or binder
              (r (third q1)))
          `(nconc
            (loop for ,v
             ,(if (eq op '<--) 'across
                (if (eq op '<..) 'from 'in))
             ,(if (eq op ':=) `(list ,r) r)
             ,@(if (eq op '<..) `(to ,(fourth q1)))
             ,@(lcer h qr))))
       `(if ,q1 ,@(lcer h qr))    ;; filter
      ))))
```

# *Optimizations*

- Count the number of elements of a List Comprehension: replace `collect` by `count`.
- Verify if a List Comprehension is not empty: replace `collect` by `return`.

# *Compiling into Fundamental Constructs*

Can we do better translations than the Loop Facility?

Could we make the translated code efficient even with low optimization settings?

Let's verify that the Loop Facility is compiled into efficient code.

Let's translate into fundamental Common Lisp constructs: `setq`, `prog`, `go`, ...

# Compiling into Fundamental Constructs

```
;;; h is the head
;;; qs is the list of qualifiers

(defmacro lc2 ((h &rest qs))
   (let ((tail (gensym "tail")))
    `(prog ((,tail (list nil)) rhead)
       (setq rhead ,tail)
       ,(lc2r h qs tail)
       (return (cdr rhead))
     )))
```

# Compiling into Fundamental Constructs

```
(defun lc2r (h qs tail)
  (if (null qs)                          ;; head
      `(block nil (rplacd ,tail (list ,h))
                  (setq ,tail (cdr ,tail)))
    (let ((q1  (first qs))
          (qr  (rest  qs)))
      (if (eq (second q1) '<-)           ;; generator
          (let ((v     (first q1))
                (r     (third q1))
                (lv    (gensym "lv")))
            `(prog (,v (,lv ,r))
              loop
               (if (null ,lv) (go endls))
               (setq ,v (car ,lv))
               ,(lc2r h qr tail)
               (setq ,lv (cdr ,lv))
               (go loop)
              endls
               ))
        `(if ,q1 ,(lc2r h qr tail))  ;; filter
        ))))
```

# *Some Speed Numbers (1)*

```
(x (x <- i5000) (y <- i5000) (< x 0))
```

Optimization O$x$ is ((speed 3) (space $x$) (debug $x$) (safety $x$))

| **Implementation** | **Opt.** | **Time in milliseconds** | | |
|---|---|---|---|---|
| | | **comp** | **lc** | **lc2** |
| Allegro 8.0 | O0 | 818 | 351 | **317** |
| | O1 | 1105 | 421 | 344 |
| | O2 | so | 871 | 861 |
| | O3 | so | 984 | 878 |

# *Some Speed Numbers (2)*

```
(x (x <- i5000) (y <- i5000) (< x 0))
```

Optimization O$x$ is ((speed 3) (space $x$) (debug $x$) (safety $x$))

| **Implementation** | **Opt.** | **Time in milliseconds** | | |
|---|---|---|---|---|
| | | **comp** | **lc** | **lc2** |
| LispWorks 4.4 | O0 | so | 363 | 360 |
| | O1 | so | 363 | **357** |
| | O2 | so | 543 | 363 |
| | O3 | so | 537 | 714 |

# *Some Speed Numbers (3)*

```
(x (x <- i5000) (y <- i5000) (< x 0))
```

Optimization O$x$ is ((speed 3) (space $x$) (debug $x$) (safety $x$))

| **Implementation** | **Opt.** | **Time in milliseconds** | | |
|---|---|---|---|---|
| | | **comp** | **lc** | **lc2** |
| SBCL 0.9.18 | O0 | 673 | 728 | **653** |
| | O1 | 680 | 633 | 683 |
| | O2 | 694 | 670 | 700 |
| | O3 | 804 | 774 | 731 |

# *Some Speed Numbers (4)*

```
(x (x <- i5000) (y <- i5000) (< x 0))
```

Optimization O$x$ is ((speed 3) (space $x$) (debug $x$) (safety $x$))

| **Implementation** | **Opt.** | **Time in milliseconds** | | |
|---|---|---|---|---|
| | | **comp** | **lc** | **lc2** |
| CLisp 2.41 | O0 | so | 7340 | **4786** |
| | O1 | so | 7227 | 5234 |
| | O2 | so | 7407 | 5504 |
| | O3 | so | 8229 | 7350 |

# Conclusion

Common Lisp has List Comprehension.

*It does! It just has this quirky syntax: Loop Facility*

We cannot assume that recursion is translated efficiently for several Common Lisp implementations.

Common Lisp is not Scheme!